Design, Implementation and Cryptanalysis of Modern Symmetric Ciphers

by

Matt Henricksen

Bachelor of Information Technology (1995)
University of Queensland

Thesis submitted in accordance with the regulations for
Degree of Doctor of Philosophy

Information Security Research Centre
Faculty of Information Technology
Queensland University of Technology

June 9, 2005

Keywords

Block Ciphers, Word-based Stream Ciphers, Cipher Design, Cipher Implementation, Cryptanalysis, Key Schedule Classification, Key Agility, Advanced Encryption Standard, RC4, Alpha1, MUGI, Dragon, Correlation Attacks, Divide and Conquer Attacks, Intel Pentium 4

Abstract

The main objective of this thesis is to examine the trade-offs between security and efficiency within symmetric ciphers. This includes the influence that block ciphers have on the new generation of word-based stream ciphers. By incorporating block-cipher like components into their designs, word-based stream ciphers have experienced hundreds-fold improvement in speed over bit-based stream ciphers, without any observable security degradation. The thesis also emphasizes the importance of keying issues in block and stream ciphers, showing that by reusing components of the principal cipher algorithm in the keying algorithm, security can be enhanced without loss of key-agility or expanding footprint in software memory.

Firstly, modern block ciphers from four recent cipher competitions are surveyed and categorized according to criteria that includes the high-level structure of the block cipher, the method in which non-linearity is instilled into each round, and the strength of the key schedule. In assessing the last criterion, a classification by Carter [45] is adopted and modified to improve its consistency.

The classification is used to demonstrate that the key schedule of the Advanced Encryption Standard (AES) [62] is surprisingly flimsy for a national standard. The claim is supported with statistical evidence that shows the key schedule suffers from bit leakage and lacks sufficient diffusion. The thesis contains a replacement key schedule that reuses components from the cipher algorithm, leveraging existing analysis to improve security, and reducing the cipher's implementation footprint while maintaining key agility. The key schedule is analyzed from the perspective of an efficiency-security tradeoff, showing that the new schedule rectifies an imbalance towards efficiency present in the original.

The thesis contains a discussion of the evolution of stream ciphers, focusing on the migration from bit-based to word-based stream ciphers, from which follows a commensurate improvement in design flexibility and software performance. It examines the influence that block ciphers, and in particular the AES, have had upon the development of word-based stream ciphers. The thesis includes a concise literature review of recent styles of cryptanalytic attack upon stream ciphers. Also, claims are refuted that one prominent word-based stream cipher, RC4, suffers from a bias in the *first* byte of each keystream.

The thesis presents a divide and conquer attack against Alpha1, an irregularly clocked bit-based stream cipher with a 128-bit state. The dominating aspect of the divide and conquer attack is a correlation attack on the longest register. The internal state of the remaining registers is determined by utilizing biases in the clocking taps and launching a guess and determine attack. The overall complexity of the attack is 2^{61} operations with text requirements of 35,000 bits and memory requirements of $2^{29.8}$ bits.

MUGI is a 64-bit word-based cipher with a large Non-linear Feedback Shift Register (NLFSR) and an additional non-linear state. In standard benchmarks, MUGI appears to suffer from poor key agility because it is implemented on an architecture for which it is not designed, and because its NLFSR is too large relative to the size of its master key. An unusual feature of its key initialization algorithm is described. A variant of MUGI, entitled MUGI-M, is proposed to enhance key agility, ostensibly without any loss of security.

The thesis presents a new word-based stream cipher called Dragon. This cipher uses a large internal NLFSR in conjunction with a non-linear filter to produce 64 bits of keystream in one round. The non-linear filter looks very much like the round function of a typical modern block cipher. Dragon has a native word size of 32 bits, and uses very simple operations, including addition, exclusive-or and s-boxes. Together these ensure high performance on modern day processors such as the Intel Pentium family.

Finally, a set of guidelines is provided for designing and implementing symmetric ciphers on modern processors, using the Intel Pentium 4 as a case study. Particular attention is given to understanding the architecture of the processor, including features such as its register set and size, the throughput and latencies of its instruction set, and the memory layouts and speeds. General optimization rules are given, including how to choose fast primitives for use within the cipher. The thesis describes design decisions that were made for the Dragon cipher with respect to implementation on the Intel Pentium 4.

Contents

List of Figures

List of Tables

Declaration

Previously Published Material

The following papers have been published or presented, and contain material based on the content of this thesis.

[1] Lauren May, Matt Henricksen, William Millan, Gary Carter, and Ed Dawson. Strengthening the key schedule of the AES. In Lynn Batten and Jennifer Seberry, editors, *Proceedings of Information Security and Privacy - 7th Australasian Conference, ACISP'02*, volume 2384 of *Lecture Notes in Computer Science*, pages 226–240. Springer-Verlag, July 2002.

[2] Ed Dawson, Gary Carter, Helen Gustafson, Matt Henricksen, William Millan, and Leonie Simpson. Evaluation of the MUGI psuedo-random number generator. Technical report, CRYPTREC, Information Technology Promotion Agency (IPA), Tokyo, Japan, 2002.

[3] Ed Dawson, Helen Gustafson, and Matt Henricksen. Analysis of statistical flaws in the RC4 encryption algorithm. In *19th British Combinatorics Conference*, Bangor, Wales, July 2003.

[4] Kevin Chen, Matt Henricksen, Leonie Simpson, William Millan, and Ed Dawson. A complete divide and conquer attack on the Alpha1 stream cipher. In Jong In Lim and Dong Hoon Lee, editors, *Information Security and Cryptology - ICISC 2003, 6th International Conference, Seoul, Korea, November 27-28, 2003, Revised Papers*, volume 2971 of *Lecture Notes in Computer Science*, pages 418–431. Springer, 2004.

[5] Kevin Chen, Matt Henricksen, Leonie Simpson, William Millan, and Ed Dawson. Dragon: A Fast Word Based Cipher. In *Information Security and Cryptology - ICISC '04 - Seventh International Conference*, 2004. To appear in Lecture Notes in Computer Science.

Acknowledgements

Cryptology took me by surprise, consumed my soul and hijacked me from a mundane career in the software industry.

My primary supervisor, Professor Ed Dawson, of the Information Security Research Centre gave the best assistance I could imagine, allowing me to immerse myself as a full-time researcher in his fine institution. Doctors Gary Carter, Lauren May, Bill Millan, Helen Gustafson and Leonie Simpson were pleasures to work with. Joint work with Lauren, Gary and Bill is featured in Chapter 3, with Bill and Leonie in Chapters 5 and 7, and with Helen in Section 4.3. Thanks also to Kevin Chen for being an intelligent stream cipher buddy; our joint work is featured in Chapters 5 and 7. Lyta Penna, of University of Central Queensland, bolstered my confidence and provided remote but fruitful discussion. Her work in [189] inspired me to write Chapter 8. I thank many members of the ISRC for their conversation and ideas.

I could not have finished this thesis without the invaluable aid of the following: Faye Wong, who whispered words of encouragement in my ear for the crazy last year; Tuyet and Minna, lifelong friends who provide me with endless inspiration; Random; my parents, Noel and Carol, for their unbounded support and friendship; and the incomparably beautiful Ye Li, to whom this thesis is dedicated.

Chapter 1

Introduction

Cryptology is the art and science of information obfuscation. It has always played a role in military history, but has become increasingly important in everyday life, following the proliferation of e-commerce, in which on-line transactions worth billions of dollars must be protected from abuse (eavesdropping, repudiation, misrepresentation, and so on).

One of the most well known aspects of cryptology – that of encryption, which this thesis addresses – is founded on very simple ideas that form the basis of childrens' games. Replace each letter in the above paragraph with the letter three positions subsequent in the alphabet, and you are *encrypting* with the Caesar cipher, designed millennia ago to protect the Roman general's battle orders. The paragraph you have encrypted is called the *plaintext*. The resulting paragraph of unreadability:

```
Fubswrorjb lv wkh duw dqg vflhqfh ri lqirupdwlrq reixvfdwlrq. Lw kdv
dowdbv sodbhg d uroh lq plolwdub klvwrub, exw kdv ehfrph ylwdoob
lpsruwdqw lq flyloldq olih, iroorzlqj wkh dqgsurolihudwlrq ri
h-frpphufh, lq wklfk rq-olqh wudqvdfwlrqv wruwk eloolrqv ri grooduv
pxvw eh surwhfwhg iurp dexvh (hdyhvgurssslqj, uhsxgldwlrq,
plvuhsuhvhqwdwlrq, dqg vr rq).
```

is called the *ciphertext*. By possessing the algorithm (which states that letters are replaced by those x positions further in the alphabet) and the key (in this case, $x = 3$), you are able to successfully reverse the process and *decrypt* the ciphertext to the plaintext. Identifying the plaintext without knowledge of the key is called *cryptanalysis*. For the simple algorithm given above, the key has only twenty-six potential values, so the amount of secrecy employed by the algorithm is trivially small.

Commercial cryptography is considerably more advanced than this simple example. In addition to encryption, it incorporates other related techniques, including protection and verification of message integrity, authentication and non-repudiation. It can be partitioned into two fields, which relate to how cryptographic keys are used. In asymmetric cryptography, two keys are used; one is visible to the initiator of the cryptographic process, and is useful for signing and encrypting documents; the other key is publicly available and useful for verifying the author of signatures and decrypting associated documents. Symmetric ciphers use a single key for the dual purpose of encryption and decryption, and for integrity protection, but are generally not used for signature generation or verification.

Symmetric ciphers fall into two categories: block ciphers and stream ciphers. Block ciphers are stateless and given plaintext directly output ciphertext. Stream ciphers maintain state and produce *keystream* which is externally combined with the plaintext to make the ciphertext. Until recently there was little in common between the stream and block ciphers, other than their use of a single secret key. Stream ciphers were fast in hardware and slow in software, and block ciphers were slow in hardware and fast in software. The output unit of stream ciphers was frequently a single bit, while block ciphers produced tens or hundreds of bits at a time. Now from just the metrics of a symmetric cipher — output units in excess of 64 bits, high-speed throughput in software, high-speed throughput in hardware — it is difficult to tell whether they originate from a stream or block cipher.

This thesis deals with techniques of symmetric cipher design that allow efficient implementation in conjunction with a high level of security. It comes at a time when the boundaries between block ciphers and stream ciphers have become blurred, opening up new avenues for design. In this chapter, block ciphers and stream ciphers are introduced in Sections 1.1 and 1.2 respectively. In Section 1.3, the aims and objectives of the thesis are explored. In Section 1.4, the structure of this thesis and its contributions are explained.

1.1 Block Ciphers

A block cipher approximates a random permutation on x-bit blocks of plaintext. As the size of x increases, a random permutation becomes exponentially more impractical to implement, but less vulnerable to statistical analysis. A block cipher represents a compromise between security and efficiency, by joining smaller $m \times n$ non-linear components ($m < x, n \leq x$) together with linear diffusion elements. The block size of the cipher typically ranges from 64 bits (for example, the Data Encryption Standard (DES) [177]) to 256 bits (Rijndael-256 [62]). The general model for a block cipher is to incorporate the linear and non-linear components within a *round function*, and iterate the function many times to produce the ciphertext.

Non-linear components are frequently implemented using substitution boxes (s-boxes) in which each output bit is related to the input bits in a highly non-linear way. The size of an s-box grows exponentially in the number of inputs as $2^m \times n$ bits. Typical sizes for s-boxes include 4×4 bits (for example, Serpent [4]), 8×8 bits (Rijndael [62]), and 8×32 bits (MARS [42]).

The block cipher provides confidentiality by mixing a secret key within each round function. Generally, the key material used within each round – the *round keys* – differ. The total material of the round keys may be produced from a much smaller secret *master key* by a *key schedule*; this expansion of a small amount of secret material to produce a larger amount of pseudo-random material is based upon a similar problem to the one that stream ciphers address.

The strength of a block or stream cipher is frequently measured by the length of its effective key size ek, as compared to the size k of its actual key. The effective key size is determined by the best attack on the cipher. A secure cipher is regarded as one in which ek approximates k, where k is beyond the range of brute-force searching (see Appendix A.1). Due to advances in computing power, traditional key sizes, such as the 56 bits used by the DES, fall within the scope of brute-force attack, and are no longer regarded as secure.

Most contemporary symmetric ciphers use keys with sizes of 128 bits and above. Opinion is divided on what constitutes a secure length: some cryptographers believe that a cipher with an effective key length of 128 bits is secure [61], while others recommend lengths of 256 bits or above [48]. The problem of choosing a key length has little to do with the efficiency of handling it within the block cipher for standard block sizes, but rather of generating the requisite number of random bits for the master key. One problem that arises when insufficient randomness exists is demonstrated in Section 3.3.2.

While there is no argument that the cipher algorithm of a block cipher needs to be efficient as possible to allow a high-throughput, there is some contention over the importance of the efficiency of the accompanying key schedule. One viewpoint is that a slow key schedule protects the cipher by increasing the time needed to launch a brute-force attack. This argument is somewhat facetious considering both the inability of modern processors to execute brute-attacks against modern ciphers with 128-bit keys, and the pressing need for key agility in wireless applications.

The block cipher function does not change over time, so encrypting the same plaintext block multiple times produces identical ciphertext blocks. For example, a 512-bit message P will be passed to a block cipher with a block size of 128 bits in four equal-length sections $P_0 ... P_4$.

$$P = P_0 \parallel P_1 \parallel P_2 \parallel P_3$$

and encrypted with key K as

$$C = E_k(P_0)(= C_0) \| \ E_k(P_1)(= C_1) \| \ E_k(P_2)(= C_2) \| \ E_k(P_3)(= C_3)$$

If $P_0 = P_1$ then $C_0 = C_1$. To obscure this kind of relationship, and reduce the potential for attack through statistical analysis, block ciphers are used in conjunction with *modes of operation*. The default mode of block cipher operation is the null mode Electronic Code Book (ECB), which is implicitly used in the example above. Other standardized modes include:

- Cipher Block Chaining (CBC): in this mode, ciphertexts are chained such that $C_i = C_{i-1} \oplus E_k(P)$, where $C_{-1} = IV$ is a public initialization vector. For $P_x = P_y, C_x \neq C_y$ unless $C_{x-1} = C_{y-1}$. In this way, all ciphertexts depend upon the values of all previous plaintexts. Thus CBC mode has a use additional to confidentiality: it can be used to generate a single digest (a MAC) that verifies the integrity of the preceding ciphertexts.

- Cipher Feedback Chaining (CFB): in this mode, a block cipher emulates a self-synchronizing stream cipher. The size of the plaintext blocks (r) does not need to match the size of the blocks (n) in the cipher algorithm, thus this mode can be used to expedite transmission of encrypted data by using smaller ciphertext blocks. The mode initializes a shift register S with an IV. It produces keystream O_i as: $O_i = E_k(S_i); C_i = P_i \oplus trunc_r(O_i); S_{i+1} = S_i \ll r \parallel C_i$, where $trunc_r$ reduces the input to r bits and $\ll$ is the left shift operator. Because this mode is stateful, identical plaintexts encrypted will produce different ciphertexts unless the same key and initialization vector (IV) and preceding plaintexts are used. As with CBC mode, each ciphertext depends upon all preceding plaintexts.

- Output Feedback (OFB): in this mode, a block cipher emulates a synchronous stream cipher, such as those studied in Chapter 4. As with CFB, the size of the plaintext blocks does not need to match the size of the blocks in the cipher algorithm. However it differs from CFB, in that the keystream is used to replenish the shift register, independently of the plaintext. Using the same notation as for CFB, OFB works as follows: $O_i = E_k(S_i); C_i = P_i \oplus trunc_r(O_i); S_{i+1} = O_i$. When $r < n$, the expected period of the keystream is $2^{\frac{n}{2}}$. When $r = n$, the expected period is 2^{n-1} [191]. Consequently block ciphers in OFB mode may be vulnerable to distinguishing attacks unless $n \geq 2k$ for a master key length of k [181]. OFB mode is stateful so identical plaintexts encrypted produce different ciphertexts unless the same key and IV are used. The ciphertexts are independent of previous plaintext.

Newer modes of operation seek to include additional features such as low-cost integrity. Examples include Integrity Aware CBC (IACBC) [116] and Offset Code Book (OCB) [199]. In [152], tweakable block ciphers are suggested as a way to provide the variability to ciphertexts provided by modes of operations, but within the block cipher, rather than externally. This kind of block cipher possesses two keys, one key for secrecy, and one (not necessarily secret) key for variability. Changing the latter key is inexpensive, and produces a seemingly different and independent block cipher. From this construction, additional modes of operation can be developed. The practicability of tweakable block ciphers is built on the premise that key schedules of block ciphers are expensive compared to the cipher algorithm. This premise is flawed: secure key schedules can be extremely efficient, as demonstrated in Chapter 3.

1.2 Stream Ciphers

The boundary between stream ciphers and block ciphers has all but dissolved, given that OFB and CFB modes of a block cipher emulate stream cipher behavior, and that stream ciphers can now output large blocks of keystream. The primary difference between block and stream ciphers is that the latter are stateful, and the former are not. From a cryptanalytic viewpoint, this means that cryptanalysts attacking stream ciphers have fewer degrees of freedom in manipulating the cipher inputs. However the algebraic attacks of [53], although as yet only successful against bit-based stream ciphers, by their nature pose a danger to all stream ciphers, since each bit of output generates a new equation that can be algebraically solved to identify key bits.

The classical example of a stream cipher is the one-time pad in which keystream bits are generated randomly and independently. Each keystream bit k_i is combined with a corresponding plaintext bit p_i to form the ciphertext $c_i = p_i \oplus k_i$ for $0 \leq i < M$ where M is the message length. Non-random generation of the keystream bits (the pad), or duplication of portions of the pad lead to recovery of the plaintext, more easily if it possesses a high level of redundancy. When used correctly, the one-time pad is unconditionally secure. However it suffers from the requirement that the keystream needs to be as long as the message it encrypts. The goal of a stream cipher is to emulate a one-time pad in producing a keystream that appears unpredictable and random, as if it were the product of a non-deterministic process. Other than the one-time pad, no stream ciphers are unconditionally secure. In deterministically generating lengthy keystream from very small inputs, these stream ciphers need to satisfy computational security requirements, in which retrieval of the key is computationally rather than theoretically infeasible.

A stream cipher consists of a state S, an update function Υ, which modifies the state S, and an output filter f which produces keystream using S. It is considered good practice to use a rekeying algorithm that determines how the initial state S_0 is populated by the master key. If the keystream is generated independently of the ciphertext, the stream cipher is described as *synchronous*. The alternative, self-synchronizing ciphers, incorporate previous ciphertext bits into the Υ function, but are not common.

One limitation of a stream cipher, indeed the same one suffered by block ciphers in ECB mode, is that two or more invocations initialized with the same key produce the same keystream. As with block ciphers, the solution is to introduce into the rekeying algorithm, an initialization vector that may remain public but must change with each invocation of the rekeying algorithm. Particularly for binary additive ciphers, which combine the plaintext with the keystream using exclusive-or, but also for any cipher which uses a involution operation as a combiner, reuse of keystream can lead to exposure of the plaintext without possession of the encrypting key. This is evident in the case of exclusive-or, as an attacker in possession of $C_1 = P_1 \oplus K$ and $C_2 = P_2 \oplus K$ can combine both ciphertexts to acquire $P_1 \oplus P_2$, which can be solved with varying degrees of success depending upon the redundancy inherent in the plaintext [163].

Typical bit-based ciphers use one or more linear feedback shift registers (LFSRs) to contain the state S. LFSRs have predictable properties and allow fast implementation in hardware, if not software. Non-linearity is introduced to the cipher by irregular clocking of one LFSR by another or by using a non-linear filter or combiner to produce the output [214]. In word-based stream ciphers, there is more freedom of design. Some, such as SNOW [67], remain conservative to their origins, some such as Helix [73] resemble block ciphers, and others such as Turing [201] and MUGI [224] retain a shift-register state, to which is appended a large and complex filter function.

1.3 Aims and Objectives

The main objective of this thesis is to examine the trade-offs between security and efficiency within symmetric ciphers. The aspect of security covers both design of ciphers, and their cryptanalysis. It is difficult to design a cipher that is immune to a catalog of cryptanalytic attacks, without awareness of what the attacks involve. Conversely, being well versed in contemporary design points a cryptanalyst to potential weak spots within a cipher design.

The other aspect of this thesis, efficiency, has until recently been neglected. But the practical needs of an industry requiring high throughput in its ciphers means that algorithms that encrypt at kilobits or megabits per second are now unwanted. The industry is looking for gigabytes per second. Bit-based stream ciphers do not provide this.

But in an effort to improve the performance of stream ciphers, their designers adopted a word-based methodology, and, as a result, studied construction of block ciphers. They borrowed well-analyzed components and made them their own. By matching the size of the keystream produced by the output filter to that of a block cipher, the designers acquired the licence for additional complexity, and therefore flexibility in cipher construction, without undue penalization of efficiency. Whereas a stream cipher that produces one bit for each invocation of the update function can afford to perform only a small amount of work, another that produces 160 bits can do a comparatively large amount of work. This is the freedom provided by moving to a word-based paradigm. The result is a family of stream ciphers that encrypts more efficiently than many block ciphers without any loss of security.

This thesis studies the influence that block ciphers have on the new generation of word-based stream ciphers, and proposes a new word-based stream cipher in which the filter function resembles a block cipher round function. The stream cipher is approximately four times faster than the Advanced Encryption Standard (AES) [62] and resists all known cryptanalytic attacks.

The thesis also emphasizes the importance of keying issues in block and stream ciphers. It demonstrates the benefits of reuse of components from the cipher algorithm: security can be enhanced without a decrease in efficiency, either through a loss of key agility, or an expanding footprint in memory. The thesis suggests a replacement algorithm for the key schedule of the AES that has better security properties but maintains key agility, and a new variant of the MUGI stream cipher that improves its key agility without loss of security.

Underlying these designs is a knowledge of how to shape designs to match characteristics of the architectures on which they will be implemented. This thesis explicitly describes factors that cipher designers need to consider to succeed in designing fast and secure ciphers.

1.4 Results

This thesis contains the following contributions. In Chapter 2, the key schedule classification by Carter et al. [45], and its application to the ciphers of the AES competition is reviewed. Some of its shortcomings are illustrated, including its inability to distinguish key schedules in which knowledge of a small number of round keys allows some of the remaining keys within the schedule to be easily discovered. An alternative classification is proposed which remedies the identified problems. It retains the spirit of Carter et al.'s work by partitioning key schedules as those which are flawed (Type I) or robust (Type II).

A simple categorization of block ciphers is developed. This incorporates the alternative key schedule classification, and extends the work of Carter et al. beyond the AES competition to categorize the key schedules of the Japanese CRYPTREC, European NESSIE and Korean ISO/IEC/JTC1/SC27-Korea competition. The categorization also highlights other construction techniques that have been the subject of interest recently, including the method by which the s-boxes are developed.

The classification shows the key schedule of the AES has a Type I key schedule, which is surprising for a modern national standard. In Chapter 3, a statistical package is used to demonstrate that the AES key schedule suffers from bit leakage and lacks sufficient diffusion, which directly aid in some attacks on reduced-round versions of the cipher. The chapter includes an alternative Type II key schedule that inhibits the attacks, using a common construction technique in which the round function of the cipher is also used in the key schedule. This has the effect of reducing the footprint of the cipher in memory, while leveraging cryptographic analysis. The throughput of the new key schedule is compared to those of other block ciphers and it is determined that the new key schedule satisfies both key agility and security requirements.

Recent word-based stream ciphers are reviewed in Chapter 4, indicating how their designs have been influenced by ciphers categorized in Chapter 2. The review includes an analysis of their update and rekeying schedules, and how they affect the performance of the ciphers. This chapter also discusses how the inadequate rekeying schedule of one popular cipher, RC4, causes a bias in the second byte of each keystream, but it contains a refutal of a claim [170] that there are biases in other keystream bytes of the same cipher. The chapter concludes by emphasizing the importance of solid rekeying strategies in stream ciphers, and briefly reviews recent cryptanalytic attack styles upon stream ciphers.

Chapter 5 presents an attack against the proposed cellular stream cipher Alpha1. This is a conventional irregularly clocked bit-based stream cipher with a 128-bit state. The chapter shows how to launch a divide and conquer attack on the cipher, breaking each of its four registers one at a time. Biases in the clocking taps allow verification of guesses on the smallest register's initial state. Following the recovery of this register, the reduced cipher can then be viewed as the underlying sequence produced by the longest register, with additive noise provided by the remaining registers, and bit insertion caused by irregular clocking. A correlation attack can be launched on the longest register, using joint probability based upon Levenshtein distances as the correlation measure. A guess and determine attack recovers the values of the remaining registers. The overall complexity of the attack is 2^{61} operations with text and memory requirements of 35,000 and 2^{30} bits respectively.

Chapter 6 gives an example of a word-based stream cipher that is immune to the kind of attacks to which Alpha1 is vulnerable. MUGI acquires this protection because it possesses a large state that can be processed efficiently using word-based operations, and a non-linear filter that resembles a block cipher function. MUGI suffers key agility problems because it is commonly implemented on an architecture for which it is not designed, and because its NLFSR is too large relative to the size of its master key. By reducing the size of its non-linear feedback shift register (NLFSR) and modifying the rekeying strategy to deal with the smaller state size, MUGI-M is developed. This is an algorithm which exhibits a 200% improvement in rekeying time, without apparent loss of security. This serves the cipher well both on 32- and 64-bit architectures.

Chapter 7 gives the specification for a new cipher called Dragon that has been developed especially for 32-bit architectures, also bearing in mind the importance of a strong and agile key schedule. The cipher uses the minimum number of simple operations to ensure its security, yet maintains a high level of efficiency. The chosen operations are known to be efficient on the Intel Pentium family, which is the most widely used processor family. With a keystream throughput of 6.7 cycles/byte and a key generation setup time of 1,395 cycles, Dragon is competitive with other modern word-based stream ciphers, and much faster than block ciphers with equivalent security estimations. Chapter 7 analyzes the cipher with respect to statistical properties, and demonstrates that any cryptanalytic attack on the cipher is not immediately obvious.

Dragon is used as a case study in Chapter 8, which provides general guidelines for fast cipher designs and implementation, and specific advice for development of ciphers for the Intel Pentium 4. Particular attention is given to the architecture of the processor, including its register set and size, the throughput and latencies of its instruction set, and the memory layouts and speeds.

The following material from this thesis has been published: the majority of the work of Chapter 3 appears in Paper 1; the key schedule irregularity of MUGI from Chapter 6 appears in Paper 2; the material concerning the keystream bias in RC4 from Chapter 4 was presented in Paper 3; and the entirety of the work in Chapters 5 and 7 was published in Papers 4 and 5 respectively.

The contribution of the author to the original work in this thesis is as follows: Chapters 2, 6, 8 and 9 are solely the work of the author. In Chapter 3, the security and performance analysis of the AES with and without the proposed key schedule belongs to the author, as does the metric by which the success of the new key schedule is gauged. In this chapter, the proposed algorithm and Crypt-X analysis are the work of Lauren May. All work in Chapter 4 belongs to the author, except for the statistical analysis of RC4 included in Section 4.3. The cryptanalysis algorithm of Chapter 5 was devised by Leonie Simpson: the author's contribution was to significantly improve the implementation and profiling techniques, without which the cryptanalysis would not be successful. The cipher algorithm and analysis of Chapter 7 was an iterative group effort, in which all parties contributed in equal measure to the design and analysis of the cipher.

Chapter 2

Block Ciphers

A block cipher is a substitution cipher that maps an m-bit plaintext to an m-bit ciphertext using a k-bit master-key. Between the inception of the Data Encryption Standard (DES) [177] and the Advanced Encryption Standard (AES) [62] of the late-nineties, m was usually 64 bits. Nowadays block ciphers with this block size are considered legacy [71] and vulnerable (at computationally great expense) to dictionary attacks (see Section A.1). In recent years, the most common block size of new ciphers is 128 bits, although some 256-bit block ciphers have been proposed (for example, NUSH [145] and SHACAL [89]). In this thesis, in-depth discussion of legacy (64-bit) block ciphers is avoided.

A general model for a block cipher is one that consists of many short rounds that mix the key material with the round's input to produce its outputs. A round is considered weak – given its inputs and outputs, it is not a computationally hard problem to determine the key material used [221]. For this reason, rounds are iterated, in which the output of one round becomes the input of the next. After a pre-determined number of rounds, the ciphertext is emitted and the next block of plaintext processed. The number of iterations in a cipher is a trade-off between security and efficiency. It is simple to create a secure cipher by iterating the weak function a very large number of times, but in an area that is already regarded as overpopulated with cipher algorithms, block ciphers need to be efficient to garner attention. To meet this criteria, the number of rounds, and the operations within them need to be carefully considered.

Each round in a block cipher contains a combination of *linear* and *non-linear* components. Linear components implement *diffusion*, which is the process of making each output bit depend upon as many input bits as possible. Examples of these linear components include fixed rotations, wired permutations and maximum distance separable matrices (MDS). Non-linear components implement *confusion*, and so mix bits to destroy the dependencies of their outputs upon their inputs. Non-linear components are as equally varied as linear components, but the most prominent is the substitution box (s-box). The types of linear and non-linear components used in block ciphers have changed recently, following the success of the block cipher Rijndael [62] in the AES competition. Many variants have followed in its wake, using components and structures (including the MDS) that were rarely seen previously.

Almost universally, as shown in Section 2.2, s-boxes are used as the prominent form of non-linearity. There are many ways to construct s-boxes: one broad categorization includes whether an s-box is random or algebraically constructed, or filtered for strong properties.

Random s-boxes are unlikely to possess strong cryptographic properties, such as optimal resistance against differential and linear cryptanalysis [205]. However, they are also unlikely to be over-defined and thus vulnerable to the XLS attack (excluding 4×4 or smaller s-boxes, which are always over-defined) [56].

While algebraically constructed s-boxes can be generated to achieve specific properties, such as optimal resistance against linear and differential cryptanalysis, their method of construction may have the side-effect of generating unknown or unwanted properties or vulnerabilities. One much emphasized example is the AES s-box which is generated using exponentiation in a Galois field, then altered using a linear transformation to rid it of its simple algebraic expression. Despite this precaution, the s-box suffered from redundancies (although not utilized in an attack) [76] [232], and permitted an expression of the cipher as the basis of an over-defined system of equations [56].

One technique that is favored by the designers of the block cipher MARS [42] and the stream ciphers Turing [201] and Dragon [48] is filtered generation of s-boxes, to acquire strong cryptographic properties such as high non-linearity, without the drawback of simple algebraic expression. In this technique, iterative changes are made to an s-box, until it passes a fitness test based upon its cryptographic properties. The s-box may be changed using random (MARS and Turing) or heuristic methods (Dragon). Generation of heuristic s-boxes is further discussed in Section 7.2.2 and in [168].

Block Cipher Structures

One common block cipher structure is the *Feistel round.* A Feistel round breaks its input block into words, and in the course of each round, uses a function of one or more source words and key words to modify target words within the block. Between rounds the identities of the source and target words alternate.

The Feistel round can be classified on the basis of how many target words each source word modifies. In a Type-n Feistel cipher, one source word modifies n target words. In the most common type of Feistel cipher, the Type-1, such as DES, the input of round i is split into halves (L_{i-1}, R_{i-1}) which are treated (with function f and key k_i) as:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$$

There are two well-known advantages to the Feistel structure. The first is that it facilitates decryption due to the invertibility of the connecting exclusive-or (or other involution) operation, and the swapping of halves at the end of the cipher [27]. Invertibility of the round function f is not required for the algorithm to be used unmodified for both encryption and decryption. All that is required is that the order of the round keys is reversed. Secondly, the Feistel structure reduces the size of the block that needs to be processed at a time (a Type-1 halves the size, a Type-3 quarters it). This leads to gains in efficiency, particularly if the size of the handled block matches the word-size of the processor on which the cipher is implemented. For example, MARS is a Type-3 Feistel cipher on a 128-bit block, meaning that the 32-bit source word matches the word-size of most processors, including the ubiquitous Intel Pentium family. In this case, the interaction within the source word can occur within a single register, and the interaction between target words is restricted to the highly efficient addition and exclusive-or operations.

An alternative to the Feistel network is the Substitution-Permutation Network (SPN), in which each word can be both a source and a target word. In the SPN, three types of components are involved: substitution, permutations, and key addition. One of the most well-known SPNs, and an antecedent to many of the SPN block ciphers devised in the last eight years, is the SQUARE cipher [60]. It has a block length of 128 bits which is conceptually divided into an array of four by four bytes. In each round, each of the sixteen cells in the array is modified by three cells and one key byte. The round consists of four operations: a mixing step (θ), a byte transposition (π), a substitution layer (γ) and a key addition layer (σ). The diffusion elements in the cipher are present in θ and π. The γ operation contains parallel applications of an 8×8 s-box. The key addition σ mixes each cell with a corresponding byte in the round key. This is a common model for most SPNs, although the details of the components differ between ciphers.

In an SPN, all of the words within the block are modified within the round. This means that confusion and diffusion may be implemented more quickly than for Feistel ciphers, and as a result, the SPN may have fewer rounds for an equivalent level of security. However, the round function needs to be constructed carefully to be invertible to allow both encryption and decryption through the same algorithm. An alternative is to use the block cipher only in a stream cipher mode such as OFB. Otherwise the footprint – the code-size and memory requirements of the cipher – grows to accommodate encryption and decryption.

Homogeneity of the Round Function

A block cipher can be categorized according to whether all of its rounds are homogeneous, or whether some rounds differ (at a high level) from others. By this, it is meant that the type and inter-relationships of components in the rounds differ, rather than the values within them.

One common line of reasoning in designing a cipher with heterogenous styles is that a cipher that is more difficult to analyze is also more difficult to attack. Consequently some ciphers have been designed as heterogenous structures in an adhoc approach to gain security. However, the designers of the MARS block cipher reasoned that many techniques of cryptanalysis involve guessing key bits at the top or bottom layers of the cipher, following which these rounds are removed to allow the middle rounds to be subjected to further attack [42]. They observed from this, that the top and bottom rounds of a cipher play a different role to the middle rounds in protecting against the attack; in particular, that the top and bottom rounds are more concerned with fast avalanche of key bits than resistance to cryptanalysis. They theorized that an additional advantage to a heterogenous cipher is that it is likely to be more resilient to new attacks, since to take advantage of a weakness in one homogeneous structure within the cipher also requires propagating it through the others, in effect making a multiple-phase and therefore more difficult attack.

The disadvantage to a heterogenous structure is the increased complexity of analysis. This ultimately played against MARS, when commentators complained its complexity was undue and hindered its analysis [12], [180].

A classification of block ciphers is formulated on the aspects discussed in this section, and used to categorize modern ciphers in Section 2.2. This classification is based upon: (i) whether the high-level structure of the cipher is SPN or Feistel; (ii) whether the cipher uses differing round functions within that structure; and (iii) the method of construction of s-boxes used within the round. A fourth aspect is included in the classification: this pertains to the type of key schedule used by the cipher, and is discussed in the next section. Appendix A contains a survey of attack techniques upon block ciphers.

2.1 Block Cipher Key Schedules

The secrecy in a block cipher is provided by round keys. Some ciphers use more than one key in each round. For example, E2 [183], a Feistel cipher that modifies sixty-four bits of text per round uses two 64-bit keys to provide 128 bits of secret material. A cipher may use whitening, in which key material is added (but not mixed) to the input of the first cipher round, and to the output of the last cipher round, this being an effective way of obscuring the inputs and outputs from attackers.

The total amount of secret key material SK for the round and whitening keys may range from hundreds to thousands of bits. This is derived by the key schedule from a master key. The task of the key schedule is to obscure the relationship between the master key and the round keys, and also between round keys, therefore preventing related-key and slide attacks (see Section A.5). It should avoid generating weak keys (keys for which encryption is the same function as decryption [121]), or too many identical round keys from different master keys [131].

For a cipher with a strong key schedule, an attacker who learns a round key of RK bits should face a task of 2^{RK} complexity in acquiring another round key, and an effort of 2^{K} to retrieve the master key. When the total material $SK \geq K$, brute-force searching, rather than attacking the key schedule or the cipher algorithm, may be the most viable method of attack.

2.1.1 The Carter et al. Key Schedule Classification

One key schedule classification that relates the difficulty that an attacker faces in discovering round and master keys, was devised by Carter et al. [46]. This classification assumes that the attacker possesses a single round key, and divides key schedules into two types. Type 1 key schedules allow the attacker to retrieve *some bits* of other round keys, or of the master key with a minimum of calculation. Type 2 key schedules are more robust. For these key schedules, the attacker needs to perform computationally infeasible tasks to retrieve additional key material using that already held. Each of the two types is further divided into three classes.

The Type 1 key schedule classes incorporate:

- **Type 1A** key schedules, in which the master key is used without modification as round keys (this is the minimum possible key schedule, a *null* key schedule);
- **Type 1B** key schedules, in which the master key is modified in a reversible and trivial way to produce the round key (for example, by rotating each round key by a predefined amount to produce its successor). The values of master key bits are immediately evident in a round key (although the position of the bits may have changed);
- **Type 1C** key schedules, when some trivial calculation may be performed to produce each round key from its predecessor, or from the master key. Given a round key, an attacker performs some calculation (such as exclusive-oring two known values) to retrieve some master key or round key bits.

Type 1 key schedules are particularly vulnerable to self-similarity cryptanalysis, which include related-key and slide attacks (see Section A.5). The Type 2 key schedule classes are:

- **Type 2A** key schedules, in which only a subset of master key bits are used to derive each round key;
- **Type 2B** key schedules, in which all master key bits are used to derive each round key;

- **Type 2C** key schedules, in which each round key is independent, such that the length of the master key equals the total of the lengths of the round keys.

There are some issues concerning this classification, which include key schedules that have a hybrid category due to polymorphic behavior; key schedules that have different categories depending upon the number of round keys already possessed; the similarity of the Types 1B and 1C categories; and the misleading position of the Type 2C category within the classification.

Hybrid key schedules Different round keys within a block cipher can fall within different categories in this classification. But the classification provides no explicit guidance as into which category they should be placed. This leads to inconsistencies in the classification.

As an example, Carter et al. list SQUARE as Type 1A since they note *the first round subkey is the master key itself.* The key schedule of SQUARE's successor Rijndael, when used with a 128-bit key, exhibits identical behavior. But Carter et al. give Rijndael a Type 1B classification because some of the other round keys are derived using a more complex algorithm. In terms of the classification, the key schedules of Rijndael and SQUARE behave in identical ways so this discrepancy should not occur.

The choice of the round key for analysis affects the categorization. Depending upon the number and position of the round keys possessed, Rijndael with a 256-bit master key may slot into any of three categories:

1. an attacker in possession of the first two round keys instantly possesses the master key. In this case, Rijndael is acting as a Type 1A cipher.

2. an attacker in possession of the first round key cannot derive any of the other round keys, since these are partially based on the second round key (the second half of the master key), which is independent of the first. In this case, Rijndael is acting as a Type 2B cipher.

3. an attacker in possession of any other round keys is able to derive some bits from other round keys and possibly the master key, without any major computational effort. In this case, Rijndael is acting as a Type 1C cipher.

In many of the attacks discussed in Appendix A, the first or the last round keys are the first targeted by an attacker. Once discovered, the game may be over for ciphers with Type 1 key schedules, depending upon how many key bits can be retrieved. For ciphers with Type 2 key schedules, the round with the discovered key is peeled off, and the reduced cipher treated as a new attack subject. This suggests that for a key schedule of a cipher that falls into multiple categories, classification is prioritized on the behavior of the round keys that are targeted first. The failing of this approach is the difficulty in determining consistency between attack types in terms of which keys they target. An alternative is to classify the schedule on the basis of the weakest round key, since if it is possible, this is the one that the attacker will target first.

Are two round keys twice the value of one? The classification only considers the scenario for an attacker in possession of one round key. The scenario can alter radically when an attacker discovers a second round key. As an example, consider Serpent and Zodiac [98]. As shown later in this survey, the Carter et al. classification awarded to the key schedules of these ciphers is Type 2B. For a single round key, an attacker can derive no information about other round keys or the master key. But if the attacker can, through brute force searching (or in some other way), discover a second round key, retrieval of subsequent round keys is trivial.

For one round key, Serpent and Zodiac have a Type 2B classification. For two round keys, they exhibit Type 1C behavior. The classification does not distinguish between robust key schedules against an attacker with multiple round keys, and those that are robust only for a single round key. When the combined material in two or more round keys is less than the number of the bits in the master key, this is a serious failing of the classification. When the combined material is greater than the size of the master key, a brute-force attack is more effective, and the number of round keys possessed by the attacker is immaterial.

Similarity of Type 1B and 1C classifications The division between Type 1B and 1C categorizations in Carter et al. scheme seems somewhat artificial. In Type 1B ciphers no calculation is required to retrieve master key bits. In Type 1C ciphers, trivial calculation is required. An attacker is unlikely to find much difference, in terms of complexity, between the two categories relative to the differences between other categories.

The division leads to confusion in the Carter et al. classification. A decryption that leads directly to DEAL's [134] master key is classified as 1B, while simple arithmetic operations that provide Crypton's [150] master key is classified as Type 1C. According to the guidelines given above, the classification of DEAL should be Type 1C.

Type 2C key schedules are not the strongest In [45], Carter states *The strongest key schedules are 2C schedules in which subkey generation is totally independent.* This is consistent with the category's placing at the end of the classification (which is ordered from weakest to strongest). However, it ignores the earlier cryptanalytic result of [121], which indicates that ciphers using independent keys may be broken as easily as Type 1A ciphers: in some circumstances an independent round key can be treated as a master key. Indeed the advice of Kelsey et al. [122] is *Avoid independent round keys...we have shown that this dramatically lowers [a] cipher's resistance to related-key attacks.*

In a fundamental way, Type 2C key schedules are similar to Type 1A key schedules, in that the master key is used directly as round keys. The difference is that in a Type 1A key schedule, the master key is repeated as many times as there are rounds, but a Type 2C master key subsequence is never deliberately repeated.

2.1.2 Repairing the Carter et al. Key Schedule Classification

Despite the issues identified in Section 2.1.1, the scheme by Carter et al. is a useful tool for evaluating the strength of block cipher key schedules. In this section, modifications are made to the classification to rectify the issues described.

In the modified classification, as with the original, Type 1 key schedules do not require laborious effort from the attacker to retrieve new key material. Type 2 key schedules require the attacker to hold more than one round key, or to recommence a new attack in order to acquire new key material. The issues relating to Types 1B, 1C and 2C of Carter's classification are repaired by amalgamating or reorganizing them, to accurately reflect the level of effort an attacker needs to discover new key material. A new category is introduced to reflect the situation that occurs when an attacker with multiple round keys finds it easy to identify new material, but an attacker with a single round key does not. This makes the classification guidelines more explicit, to avoid confusion about into which class to place a key schedule, when different round keys exhibit different behavior.

The new Type 1 key schedule classes are:

- **Type 1A** key schedules, in which the master key is used directly as *each* round key, or in which it is split, without modification, across round keys.

  This category is identical to Carter et al.'s classification, except that it incorporates the case when halves of a n-bit master key are alternated between rounds that use $\frac{n}{2}$-bit round keys. In this case, knowledge of one round key immediately provides knowledge of half the round keys and one half of the master key.

- **Type 1B** key schedules, in which some minimal calculation is required upon any round key to derive parts of other round keys or the master key. This calculation may involve any primitive operations, such as addition or rotation, or a decryption using the cipher's round function. It also includes the case in which the master key is directly used as some (but not all) round keys, and can be easily calculated from other discovered round keys.

  This classification is an amalgamation of Carter et al.'s Type 1B and 1C levels. As indicated in Section 2.1.1, the distinction between these two levels is not of interest to an attacker. It also addresses the case when different round keys fall into different categories of the Carter et al. classification. For example, the polymorphic behavior of Rijndael's key schedule is resolved: because it uses the master key directly as some round keys, and because master key bits are so easily retrievable when given other round keys, Rijndael's key schedule falls into the Type 1B category of this classification.

- **Type 1C** key schedules, in which independent round keys are used, such that the length of the master key equals the total of the lengths of the round keys; or in which the master key is used as some (but not all) round keys, but master or round key bits cannot be easily derived from other round keys.

  This classification subsumes Carter et al.'s Type 2C classification. But as reflected by its positioning in the Type 1 category of the new classification, this type of key schedule is not effective against some kinds of related key attacks.

The Type 2 key schedule classes are:

- **Type 2A** key schedules, in which possession of a single round key does not allow trivial calculation of bits of other round keys or the master key. However, an attacker with multiple round keys may be able to easily calculate other key material, under the proviso that the effort 2^{SK} to acquire the original round keys does not equal or exceed the effort required for a brute-force attack 2^{K}.

  There is no comparable class in Carter et al.'s classification, which concentrates on a possession of a single round key. Thus it has no way to distinguish between the key schedules of the Zodiac and Serpent block ciphers, in which an attacker with one round key gains no headway in identifying new key material.

  The Zodiac cipher supports a 256-bit master key and uses 64-bit round keys. An attacker who holds two round keys totalling 128 bits can easily acquire other round keys without attacking the cipher further. The Serpent cipher also supports a 256-bit master key and uses 128-bit round keys. An attacker who holds two round keys can acquire without effort the 256-bit Serpent master key. However, the total effort to acquire the two 128-bit round keys may have exceeded that needed in a simple brute-force attack on the 256-bit master key. Thus while Carter et al. would award both ciphers a 2B classification, the new classification awards Zodiac a 2A classification and Serpent a 2C classification (which is equivalent to a 2B classification in Carter's scheme).

- **Type 2B** key schedules, in which only a subset of master key bits are used to derive each round key, and in which an attacker faces intractable problems to acquire new key material, given that the key material SK of the held round keys is less than the size of the master key K.

  This class is comparable to Carter's Type 2A classification, with the stipulation that the key schedule is robust against $SK < K$ bits of key material in the attacker's possession.

- **Type 2C** key schedules, in which all master key bits are used to derive each round key, and in which an attacker faces difficult problems to acquire new key material, given that the key material SK of the held round keys is less than the size of the master key K.

This class is comparable to Carter's Type 2B classification, with the stipulation that the key schedule is robust against $SK < K$ bits of key material in the attacker's possession.

In [45], Carter surveyed the AES candidates and classified their key schedules. Table 2.1 shows the differences in classifications for those block ciphers. The new classifications are explained further in the following section.

Cipher	Carter Classification	Henricksen Classification	Notes
Cast-256	2B	2C	Equivalent
Crypton	1C	1B	
DEAL	1C	1B	Incorrectly noted in [45] as Type 1B
DFC	2A	2B	Equivalent
E2	2B	2C	Equivalent
Frog	2B	2C	Equivalent
HPC	1B	1B	Incorrectly noted in [45] as Type 2B
LOKI97	2B	2C	Equivalent
Magenta	1A	1A	Incorrectly noted in [45] as Type 1B
Mars	2B	2C	Equivalent
Rijndael	1B	1B	
RC6	2B	2C	Equivalent
SAFER+	1B	1B	Incorrectly noted in [45] as Type 1C
Serpent	2B	2C	Equivalent
Twofish	2B	2C	Equivalent

Table 2.1: Differences Between Carter et al.'s and Henricksen's Key Schedule Classification of the AES Candidates

2.2 Categorizing Block Ciphers

Prior to 1998, there were only a handful of commonly known and industrially deployed ciphers including DES, IDEA [143], FEAL [172], RC2 [194] and RC5 [196]. The United States' Advanced Encryption Standard and analog European, Japanese and Korean competitions changed all this. Now block ciphers are prolific, and new ciphers are viewed with a degree of cynicism unless distinguished through a high degree of efficiency or an elegant security proof (for example, the Wide-Trail Strategy discussed in Section A.2). In this section, 128-bit block ciphers proposed in the recent competitions are described and classified according to a four-component categorization that includes:

- Cipher Structure: which specifies whether the cipher is constructed using a Feistel structure, an SPN, or a novel design.
- Round Homogeneity: whether, at a high-level, each function looks identical. Detailed differences between functions, such as the values used in s-boxes, are ignored.
- S-box construction: whether the s-boxes (if any are used) are constructed randomly or using algebraic heuristic means; also, whether they are filtered or keyed.
- Key Schedule: which is classified according to the scheme developed in the Section 2.1.2.

2.2.1 Advanced Encryption Standard

The call for ciphers in the AES competition [179] was made in September, 1997. The requirements were for a block cipher with a block size of 128 bits, and a key schedule that supported 128, 192 and 256 bit keys. The block cipher was required to be more efficient in software than the Triple-DES [177] on a majority of platforms.

Of the submissions, fifteen were accepted. These included Cast-256 [3], Crypton, DEAL, Decorrelated Fast Cipher (DFC) [85], E2, Frog [78], Hasty Pudding Cipher (HPC) [207], LOKI97 [39], Magenta [113], MARS, RC6 [197], Rijndael, SAFER+ [157], Serpent and Twofish [205].

Of these fifteen ciphers, seven had significantly new designs. These were DFC, E2, Frog, HPC, Magenta, and Serpent. Of these designs, Frog and HPC were highly unconventional. Frog, LOKI97 and Magenta were broken almost immediately after their presentation [16], [136], [222].

Cast-256, RC6 and SAFER+ were extended from earlier 64-bit ciphers to meet the requirements of the AES. Crypton, Rijndael, and to an extent Twofish, owed much to the SQUARE SPN cipher. DEAL and LOKI97 were strongly influenced by the DES cipher.

In 1999, the fifteen candidates were narrowed down to five finalists [180]. These were MARS, RC6, Rijndael, Serpent and Twofish. From these, Rijndael was chosen as the AES winner and standardized in December of 2001.

All fifteen of the AES candidates are summarized and classified below.

Cast-256 (Feistel, Heterogenous, Filtered, 2C)

Cast-256 is a byte-oriented cipher. It inherits security properties and analysis from the 64-bit Feistel cipher Cast-128 [1].

Cast-128 uses three different round functions. In each round function, a 32-bit data key is mixed with the 32-bit input and the combination rotated using a 5-bit rotation key. The result is split into bytes, each of which is passed through an 8×8 s-box. The four s-box outputs are recombined to form the round output, using mixing from different algebraic groups. The three round functions differ in how the data, key and plaintext are mixed, and how the s-box outputs are recombined. The round functions alternate between rounds.

To upgrade this algorithm for use with a 128-bit block, Cast-256 uses a 48-round Feistel structure that quarters the block into 32-bit sub-blocks. The right-most sub-block is used as input to the round function. This sub-block is replaced by the combination of the function output with the third sub-block. Then the 128-bit block is rotated by thirty-two bits. As with Cast-128, round functions alternate between rounds. However, after twenty-four rounds, the order of alternation is inverted for the remaining twenty-four rounds, to allow the extended Feistel structure to be used for both encryption and decryption without alteration.

The s-boxes for Cast-256 are inherited directly from Cast-128. These are 8×8 s-boxes heuristically constructed from bent functions for strong cryptographic properties.

Cast-256 has no whitening. The key schedule produces 37-bit round keys for each of the 48 rounds. The schedule contains a 256-bit state, which is initialized by the padded master key and altered by constants. The algorithm used by the schedule to update the state is similar to the CAST-256 round function, and is iterated ninety-six times. This non-invertible process chains the round keys, producing a Type 2C key schedule.

Crypton (SPN, Homogenous, Filtered, 1B)

Crypton is a variant of the SQUARE cipher that does not explicitly adhere to the wide-trail design strategy. It is byte-oriented.

Crypton uses a very similar SPN round structure to SQUARE but that replaces the byte-wise MDS used in the mixing step (θ) with bit-based masking and binary additions, presumably for reasons of efficiency. A consequence of this is the branching number of the operation is reduced by one to four. To compensate for the reduced round diffusion, Crypton uses twelve rounds rather than eight rounds used by SQUARE. In alternate rounds, different s-boxes and mixing steps are used, but from an abstract level, Crypton remains an homogenous cipher.

Crypton uses four 8×8 s-boxes, which are variants of a single involution s-box that is constructed randomly from two 4-bit permutations and filtered on the basis of good differential and linear characteristics.

The original key schedule of Crypton had undesirable features since the knowledge of one round key could easily be used to determine half of the remaining round keys. Consequently it was classified by Carter et al. [46] as Type 1C. In the updated version of Crypton [150], the key schedule was amended to remove weak keys and the round function used in the generation of the 128-bit round keys from the master key. It does not use whitening. Round keys for rounds with even/odd numbers respectively are related, as they are determined using only shifting of the words and exclusive-oring with known constants. By our classification, Crypton has a 1B key schedule.

DEAL (Feistel, Homogenous, Filtered, 1B)

DEAL is a wrapper around the DES to bring it into line with the requirements of the AES Competition, and to supercede Triple-DES. Triple-DES was already being used to circumvent the short-comings of DES, but had poor performance and a block size that was vulnerable to dictionary attacks, and it was subject to a related-key attack [121]. DEAL is a Feistel cipher that uses the DES cipher in its entirety as a round function. It is competitive with Triple-DES, which executes a DES encryption three times for a 64-bit block, since DEAL executes one DES encryption for each of its six rounds on a 128-bit block.

DEAL does not use whitening. The DEAL key schedule breaks the master key into 64 bit segments, and generates the round keys by alternating among the segments, masking them with round constants and encrypting them using DES with a fixed key. Each round key is formed by exclusive-oring the result with the previous round key. Because knowledge of the first round key leads to the master key after decryption with a fixed key, Carter et al. classifies DEAL's key schedule as Type 1B. However, this involves some (trivial) calculation, so according to the guidelines used in their classification, DEAL actually has a Type 1C key schedule. According to our classification, this trivial calculation accords a Type 1B status to DEAL.

DFC (Feistel, Homogenous, Random, 2B)

DFC is a simple byte-oriented Feistel cipher belonging to the PEANUT family [85]. The ciphers in this family come with security proofs based on decorrelation theory. DFC is targeted towards 64-bit architectures, which negatively impacts its performance on 32-bit processors.

The round function of DFC has two parameters a (the data) and b (the key). Given the 64-bit input x, the output RF equals $CP((a \times x + b) \bmod (2^{64} + 13)) \bmod 2^{64}$. CP is a permutation that takes a 64-bit input $y = y_l|y_r$. The output of CP is $y_r \oplus RT(y_l)|(y_l \oplus KC) + KD \bmod 2^{64}$ where KC and KD are 32- and 64-bit constants respectively. RT is a random 6×32 s-box. The round function is iterated eight times.

The key schedule generates eight 128-bit round keys using four rounds of the encryption function. Carter et al. note that while the encryption function is non-reversible, the first round key depends on half the master key bits. This invites an exhaustive search on the first round key. Therefore the DFC key schedule is Type 2B.

E2 (Feistel, Homogenous, Algebraic, 2C)

E2 is a word-based cipher that uses a global Feistel structure, but within the Feistel round adopts an SPN layout. It has twelve rounds. The SPN consists of a layer of eight parallel 8×8 s-boxes, followed by a 64-bit wide permutation, followed again by a layer of parallel 8×8 s-boxes, finishing with a byte-based rotation. A single s-box is used repeatedly within this structure. It is generated algebraically as an exponentiation in a Galois field, followed by an affine transformation, in the same manner as Rijndael.

The E2 key schedule generates a 128-bit round key for each round, and four 128-bit whitening keys. The round keys are applied in two 64-bit parts, each before a layer of s-boxes in the round function. The key schedule initializes a 256-bit value from the master key, which is padded if necessary. The schedule iterates two threads, one containing the key-based value, and the other an auxiliary value chained from a constant. During the course of the round, each value is modified four times by an SPN. The outputs of the SPN for the key-based value are incorporated into the auxiliary value prior to its application to the SPN. The outputs of the SPN for the auxiliary value are concatenated to form two 128-bit round keys. The chaining method is non-invertible, so the key schedule is type 2C.

Frog (Adhoc, Homogenous, Random, 2C)

Frog is one of a small number of ciphers surveyed here that does not conform to either the SPN or Feistel structure. Instead it treats the cipher block as a circular array of bytes, and in each of eight rounds, performs four operations on each byte, three of which are based upon byte values in the key. The confusion operations within Frog include exclusive-oring each byte in the array with a corresponding key byte, and performing a byte substitution using a keyed s-box. The diffusion operations include exclusive-oring the output of the s-box into the subsequent byte, and also into another byte, the position of which is determined by a key-dependent permutation. There is little theoretical underpinning to this design.

Frog has no whitening. Key setup involves taking a key of between 40 and 1,200 bits, and concatenating it to make an internal key of 18,432 bits. This is used to encrypt a zero-filled array of 18,432 bits using Frog in CBC mode. The encrypted array is used to populate the array used during the exclusive-or operation, the keyed s-box and the keyed permutation. The key schedule is Type 2C, since the one-way Frog round is used to produce each round key from all bits of the master key.

Hasty Pudding Cipher (Adhoc, Homogenous, Keyed, 1B)

The oddly named Hasty Pudding Cipher is a family of block ciphers with varying block- and key-sizes. The HPC-medium cipher conforms with the AES specifications of a 128-bit block and key sizes of 128, 192 and 256 bits.

HPC operates on 64-bit words using addition, subtraction, exclusive-or, shifting and a key-dependent s-box. The key is not mixed directly with the round, but influences the ciphertext through the key-dependent s-box. The round function has an ad-hoc design and is iterated eight times.

HPC has no whitening. The key schedule sets up the 8×64 key dependent s-box. The s-box is initialized with constants, over which the repeated key is exclusive-ored. The key schedule stirs each value in the s-box three times, using a lengthy non-linear function that mixes three s-box values using addition, subtraction, exclusive-oring, logical-oring, and shifting. This process is non-invertible so the master key can not be deduced from knowledge of any word in the s-box. Also, each word in the s-box depends upon all bits of the master key. For this reason, Carter et al. classify the key schedule as Type 2B. However, since the s-box does not change between rounds, and no explicit key addition process is followed, knowledge of the key in one round permits knowledge of the entire key in any other. The key schedule classification guidelines make clear that this kind of key schedule is Type 1B.

LOKI97 (Feistel, Homogenous, Algebraic, 2C)

Along with DEAL, LOKI97 is one of the more conservative entrants in the AES competition. It is a Type-1 Feistel cipher descended from LOKI89 [40] and LOKI91 [38], both of which bear strong resemblance to the DES. The homogenous round function of LOKI97 contains a keyed permutation, a static permutation, a 64-bit expansion box, and two rows of s-boxes, each of which contains alternating 13×8 and 11×8 s-boxes. No explicit key addition is used within the round. The s-boxes use cubing in a Galois field and are optimized against differential and linear cryptanalysis; however, the diffusion elements do permit such attacks (see Section A.2 and A.3). The keyed permutation obscures which s-box each bit of input enters during the round, and the static permutation uses a latin square to provide fast diffusion.

The key schedule uses an unbalanced Feistel network that leverages the round function and iterates it forty-eight times to produce 16×256-bit round keys. These are used in the keyed permutation, as input into the second row of s-boxes, and as whitening material for the half block that is not processed in the round function. Discovery of a 256-bit round key leads, with trivial computation, to other round keys later in the schedule. But the schedule is not invertible, and does not allow recovery of the master key. Because the effort needed to derive other key material is equivalent to a brute-force search, this key schedule is categorized as Type 2C.

Magenta (Feistel, Homogenous, No s-boxes, 1A)

Magenta is a homogenous Feistel cipher. It contains no s-boxes, its non-linearity coming from a modified Fast Hadamard Transform structure which is applied recursively. It is a simple but slow cipher, since the deepest structure in the recursion uses exponentiation on a pair of bits. In an encryption of a 128-bit block, this exponentiation is executed 2,304 times.

Magenta does not possess a complex key schedule. The 256-bit master key is split into 128-bit segments which are alternated between rounds. Carter et al. categorize this key schedule as Type 1B because knowing one round key does not automatically provide the entire master key. However, their Type 1A category provisions for learning only part of the master key directly. This is the category to which the key schedule of Magenta belongs.

MARS (Feistel, Heterogenous, Filtered, 2C)

MARS is explicitly geared towards fast performance on contemporary 32-bit machines, on which addition, substraction, rotation and multiplication were expected to be fast. By extensively using these efficient and easily analyzed operations, the designers of MARS expected their cipher would carry the same properties. Another distinguishing feature of MARS is its heterogenous structure, in which the top and bottom parts play different roles to the middle parts. However, this heterogenous structure has lent to the impression that MARS is a complex and difficult-to-analyze cipher, which is one reason that it was not selected from the five finalists as the AES [180].

The top and bottom rounds of MARS are "wrapper layers" that add key words, and perform several rounds of unkeyed s-box mixing that provide rapid avalanche of key bits.

The middle rounds of MARS are a cryptographic core that consist of eight rounds of a keyed forward transformation, followed by eight rounds of a mirrored keyed backwards transformation. Each round uses one data word and two keyed words to influence three other data words, using a combination of s-box lookups, multiplications, additions, and data-dependent rotations. When combined with addition, data dependent rotations are highly efficient at defeating linear and differential cryptanalysis. However, only the five least significant bits of the rotation word are used in the rotation. High-bits of multiplication products are used to provide these rotation amounts, since they depend upon the most bits in the operands. Conversely, the most significant bits in the multiplicand and multiplier have the least effect upon the bits in the product; these bits are instead used as inputs into the highly non-linear s-box. In this way, the use of an economical number of simple operations contribute to a strong and secure cipher.

The MARS s-box was created pseudo-randomly using the SHA-1 hash function, and filtered on the basis of five differential and four linear properties. However, Burnett et al. [41] identified a mismatch between the resulting s-box and the claimed properties, and proposed techniques by which better s-boxes could be generated more quickly.

The key schedule of MARS generates 256 bits of whitening material, and 64-bit round keys for each of the sixteen rounds. It expands the initial key (of between 128 and 1,248 bits) to 1,280 bits, using a simple linear transformation involving exclusive-ors and fixed rotations. The expanded array is mixed using seven rounds of a simple Type-1 Feistel network. Carter et al. note that for a 128-bit key, only three rounds of stirring are required for all round keys to become dependent upon all bits of the master key. Finally, weak keys are identified and remedied. Because the round keys are not generated from contiguous sections of the expanded array, knowledge of one round key does not aid in calculation of others. Consequently this is a Type 2C key schedule.

Rijndael (SPN, Homogenous, Algebraic, 1B)

Rijndael was the winner of the AES competition, and is referred to as the AES in the remainder of this thesis. It is a byte-based cipher. One of its strong points is formalized defence through its use of the wide-trail strategy, which it inherited from SQUARE. It is a SPN that bears strong resemblance to SQUARE, differing primarily in its use of a high-diffusion MDS as the mixing component. The combination of the MDS and the repeated use of a single robust 8×8 s-box means provable security against basic forms of differential and linear cryptanalysis. The other point in its favor is efficiency by executing simple operations over a small number of rounds.

Each round in the AES is identical, with the exception of the last, in which the mixing component is removed, to allow reuse of the algorithm for encryption and decryption. Its s-box is constructed algebraically using inversion in a Galois field combined with an affine transformation. This is due to a result by Nyberg [184].

The AES does not use whitening. The key schedule of the AES is Type 1B. In the case where the block size is 128 bits, knowledge of a 128-bit round key may directly provide knowledge of 96 bits of the round key two rounds previous.

Some master key bits are directly used as the round key.

RC6 (Feistel, Homogenous, No s-boxes, 2C)

RC6 is a modification of the popular 64-bit block cipher RC5. It can be viewed as two parallel but intertwined versions of RC5, where rotation amounts depend upon a quadratic equation of variables from both instances.

In the AES submission, RC6 iterates twenty rounds of its Feistel structure. It is optimized for 32-bit processors and uses multiplications in conjunction with rotations, instead of s-boxes, for non-linearity. This parallels MARS, but RC6 is unique in that these operations replace rather than complement s-boxes. At the time of its design, it was expected that this slimline design would lead to an extremely fast cipher, but as with MARS, the newly emerging and now ubiquitous Intel Pentium 4 architecture penalized precisely these operations (see Chapter 8).

The standard version of RC6 uses 22×64-bit round keys, two of which provide whitening. The key schedule of RC6 is identical to that of RC5, except that the core function is iterated at least three times per round key. Each round key is generated from all bits of the master key, and chained to all other round keys, making the key schedule function non-invertible. Consequently, the key schedule of RC6 is Type 2C according to our classification.

SAFER+ (SPN, Homogenous, Algebraic, 1B)

SAFER+ is an upgrade of the 64-bit block SAFER cipher family to meet AES requirements. Despite its authors' claims to the contrary, it is a classic substitution-permutation network. SAFER+ is a byte-oriented cipher.

The SAFER+ round function contains two key addition layers, between which is sandwiched a row of s-boxes. After the second key addition layer a block-wide linear transformation occurs based upon pseudo-hadamard transforms (PHTs). The key additions are not performed uniformly, but alternate between pairs of exclusive-ors and modular additions. Likewise pairs of s-boxes are alternated.

SAFER+ contains two algebraically generated 8×8 s-boxes. One is constructed as $S_1(x) = 45^x$ mod 257 mod 256 and the other by $S_2(x) = \log_{45}(x)$ mod 257 mod 256.

SAFER+ does not use whitening. The key schedule generates 128-bit round keys for eight, twelve or sixteen rounds for key sizes of 128, 192 or 256 bits respectively. As the master key provides the first round key according to Carter et al.'s classification, the cipher should be categorized as Type 1B rather than Type 1C. The remaining round keys are generated iteratively from the previous round key, by rotating within the bytes by three bits, rotating between the bytes by one byte, and combining the result with a constant. As knowledge of any round key allows retrieval of at least fifteen master key bytes, according to the new classification, this is a Type 1B key schedule.

Serpent (SPN, Homogenous, Filtered, 2C)

Serpent attempts to combine a conservative and secure design with efficiency through its use of bit-slicing. Until the attack of [56], it was widely regarded as one of the more secure AES finalists; however, it is not competitive in terms of throughput [180].

The cipher consists of an initial permutation, thirty-two iterations of the round function and a final permutation. The initial and final permutations are bit-based and convert the block to bit-sliced form and vice versa. In each round, each of the thirty-two bit-sliced words undergoes key addition with four bits of the round key, and the result is used as input to a replicated 4×4 s-box. The output of the thirty-two s-boxes is recombined and, in all but the last round, used as input to a linear transformation. In the last round, this transformation is replaced by a final key addition.

The cipher has eight 4×4 s-boxes. Each s-box $S_i, 0 \leq i < 8$ is used in parallel eight times in rounds r such that $r \bmod i = 8$. The s-boxes were designed using an RC4-like algorithm in which entries were swapped until differential and linear criteria were met. The XLS attacks of Courtois and Pieprzyk [56], described further in Section A.6, apply to Serpent because it uses 4×4 s-boxes, which are always over-defined.

Serpent does not use whitening. The key schedule produces 33×128-bit round keys from a master key that is, if necessary, padded to 256 bits. The key schedule produces thirty-three intermediate keys, based upon linear combinations of parts of the current and previous intermediate keys, the round key number, and a constant. The intermediate keys are placed through s-boxes in bit-slice mode to produce the round keys. Because each 32-bit word in the round key depends on between two and four 32-bit words exclusive-ored in the previous round key, a single round key does not give any information about other round keys or the master key. Knowledge of any eight consecutive 32-bit round key words gives knowledge of all round keys and the master key; however this is equivalent to a brute-force search, so the key schedule of Serpent has a Type 2C classification.

Twofish (Feistel, Homogenous, Keyed, 2C)

Twofish [205] is a word-based Type-1 Feistel cipher that uses sixteen homogenous rounds. For diffusion, Twofish uses PHTs inspired by the SAFER family, and an MDS matrix inherited from SQUARE. Twofish distinguishes itself from most of the other AES candidates through its use of four 8×8 keyed s-boxes.

The keyed s-boxes complicate analysis of Twofish and may be one of the contributing factors in its failure to win the competition. But the cipher is highly regarded and was one of the five finalists [180].

The key schedule of Twofish is an SPN that uses randomly selected fixed 4×4 s-boxes in conjunction with a MDS matrix, and addition and rotation operations. It produces 256 bits of whitening material, 64-bit round keys for sixteen rounds of the round function and material for the keyed s-boxes. It is non-invertible and, due to its lengthy keying process, complicated to analyze. Its classification is Type 2C.

2.2.2 Japanese IPA CRYPTREC

In June 2000, the Japanese Information Technology Promotion Agency (IPA) conducted a call for a wide range of cryptographic techniques including legacy and contemporary block-ciphers [108]. In total they received six entrants to the 128-bit block cipher section, including the AES candidates MARS and RC6.

Also submitted were Camellia, CipherUnicorn-A [219], Hierocrypt-3 [186], and SC2000 [211]. SEED [141], although not formally submitted, was a target for examination. The AES cipher was added as a candidate in 2001. Camellia is the successor to the AES candidate E2, to which SEED also bears obvious resemblances. Hierocrypt-3 is derived from the AES block cipher. SC2000 takes many of its operations from the AES and Serpent. CipherUnicorn-A is a new and complicated design. None of these ciphers have been seriously compromised by attacks.

In [109], the CRYPTREC committee recommended the ciphers AES, Camellia, CipherUnicorn-A, Hierocrypt-3, and SC2000 for use in constructing Japanese e-government systems. RC6 was removed from consideration due to intellectual property rights issues.

Camellia (Feistel, Heterogenous, Algebraic, 1B)

Camellia is a revamp of the failed AES candidate E2, and inherits some features from the legacy NESSIE candidate MISTY-1 [160]. The cipher is an eighteen round Feistel cipher in which each round resembles an SPN. While the SPN of E2 contains two layers of s-boxes, Camellia's SPN contains only one, which is compensated for by an increased number of rounds.

The SPN permutation and byte rotation in Camellia are derived from E2. However, after each group of six SPNs, a MISTY-like linear function (denoted FL) and its inverse are applied in parallel to the full block. The FL and FL^{-1} functions are intended to add non-regularity to the cipher to provide security against unforseen attacks. They are simple, containing logical ands and ors, exclusive-ors and one-bit rotations. The last of these disrupts byte-oriented cryptanalysis, a feature that the cipher shares with Twofish. Camellia has a more ad-hoc approach to heterogeneity than MARS.

As with E2 and the AES, the s-boxes are generated algebraically using exponentiation in a Galois field, and disrupted using affine transformations. Camellia uses four different 8×8 s-boxes, three of which are simple modifications of the fourth, using rotation on either the s-box input or output.

Camellia uses pre- and post-whitening, of 128 bits each. The key schedule uses two 128-bit variables K_L and K_R, which are initialized with the master key, and padded if necessary. Two additional variables K_A and K_B are generated by iterating the round function on $K_L \oplus K_R$ six times using constants as keys. Each round key is created from one of the four variables rotated by a fixed amount, thus Camellia has a Type 1B key schedule.

CipherUnicorn-A (Adhoc, Homogenous, Algebraic, 1B)

CipherUnicorn-A has a 128-bit block size and permissible key sizes of 128, 192 and 256 bits. It is a homogenous cipher with a complicated Feistel-like round function in that the round is broken into two computations. At the end of the round, one word from the second computation is exclusive-ored with the result of the first. This forms the input to the next round.

The first computation contains ten sub-rounds. The second computation contains six sub-rounds. In each sub-round, half of the computation's words form input to s-boxes, which modify the remainder of the words. In alternate sub-rounds, the roles of the source and target words are swapped. The second computation influences the source words of the first computation's final two sub-rounds.

CipherUnicorn-A uses four 8×8 s-boxes constructed algebraically in the same way as for the AES.

CipherUnicorn-A uses pre- and post-whitening totalling 256 bits of key material. The key schedule for CipherUnicorn-A uses a network of 64-bit MT functions. This function is based upon a network of s-boxes, logical operators, multiplication and rotation, and produces a thirty-two bit round key. The input to the MT function is derived solely from the round key four rounds earlier. Consequently CipherUnicorn-A has a Type 1B key schedule.

Hierocrypt-3 (Nested SPN, Homogenous, Algebraic, 2C)

Hierocrypt-3 is a cipher in the style of SQUARE and the AES, but uses a 2-tiered nested SPN structure, which its authors claim benefits efficiency. The cipher is deceptively simple, since the only components used in the round function are s-boxes, MDS matrices, and key addition. Each round in the top level of the SPN consists of a layer of 32×32 virtual s-boxes (denoted xs-boxes) and an MDS, which is implemented as network of exclusive-ors between the sixteen bytes in the block. The second level of the SPN is accessed through the xs-box layer, which is implemented as two layers of 8×8 s-boxes, between which is sandwiched a 4×4 MDS matrix on 8-bit data. The s-boxes are identical and are algebraically generated by exponentiation in a Galois field, and modified using pre- and post-linear transformations.

The Hierocrypt round key schedule generates 256-bit round keys for each of the six to eight rounds, and for a 128-bit post-whitening. Each round key is split into halves, which are used at the start and finish of each second level round respectively. The iterated key generation algorithm is Type 2C, and consists of generating intermediate keys using byte-level permutations in conjunction with a reduced round function. The intermediate keys are further modified by another reduced round function to produce the round keys. The intermediate keys are combined in a non-invertible way so cannot be discovered from a round key and used to calculate other key material.

SC2000 (SPN + Feistel, Homogenous, Algebraic, 2C)

SC2000 has a unique modified Feistel structure. The round structure is described as $I - B - I - R \times R$. This structure is iterated six or seven times, depending upon the length of the master key, and followed with $I - B - I$ post-whitening. The I function is a key addition layer that operates on 128-bit words. The B function is an SPN-type layer of bit-sliced 4×4 s-boxes influenced by Serpent. The R function is a Feistel structure in which the rightmost 64 bits are used as an input to a function F that modifies the leftmost 64 bits. Although there are many different parts to this structure, the round itself is homogenous.

The R function is sub-divided into three layers. The first layer is a row of twelve s-boxes, of which eight are 5×5 and four are 6×6 s-boxes. The second layer is a linear transformation implemented by two parallel 32×32 MDS matrices. In the third layer, the two halves of the 64-bit block are mixed using logical operations.

There are three s-boxes used in SC2000: S_4, S_5 and S_6. These are generated by a power mapping in a Galois field followed by an affine transformation.

SC2000 does not use key whitening. The key schedule generates between fourteen and sixteen 128-bit round keys, depending upon the master key length. It is a three stage process. In the first stage, the master key is padded to eight 32-bit words, by duplicating as many as necessary of the master key words. In the second stage, the padded key is expanded to a working key of twelve words, using an S_4 s-box layer and an MDS matrix. Each working key word depends upon only two of eight master key words. In the third stage, the working key words are chosen in pseudo-random (but fixed) order and combined using rotation, modular addition, subtraction and binary addition, such that each round key depends upon every master key bit. The generation of the working key using pairs of master key words ensures that knowledge of one round key does not provide knowledge of any other or of the master key. This key schedule belongs to the 2C category.

SEED (Feistel, Homogenous, Algebraic, 2C)

SEED [141] is a traditional Feistel cipher with a 128-bit block and a 128-bit master key. The round function is split into halves, one of which is applied to a nested SPN function twice, the other being applied once. The results of each half are combined inside the Feistel round, using modular addition. The nested SPN consists of a layer of 8×8 s-boxes, followed by a permutation.

The s-boxes are generated algebraically, as a power mapping in a Galois field, followed by a linear transformation.

The key schedule generates 64-bit round keys for each of the sixteen rounds.

There is no pre- or post-whitening. Round keys are generated using the non-invertible round SPN. The round keys are split into halves, which are independent, so that one half of the master key will never influence one half of each round key. However, since both halves are applied within a round, all master key bits affect each round key bit, leading to a Type 2C classification.

2.2.3 NESSIE

The NESSIE (New European Scheme for Signatures, Integrity and Encryption) competition commenced on January 1, 2000 [71] as a European analog to the Advanced Encryption Standard. As with the CRYPTREC competition, its scope was much broader than the AES competition, incorporating hash functions, stream ciphers and asymmetric primitives. The block cipher competition was divided into three components: 64-bit (legacy) block ciphers, 128-bit block ciphers, and variable-length block ciphers. The entrants in the last two categories included the AES candidate RC6, the CRYPTREC candidates Camellia, Hierocrypt-3 and SC2000, and new ciphers Anubis [9], Grand-Cru [36], Noekeon [61], NUSH, Q [164], SAFER++ [157] and SHACAL.

Of the new ciphers, Anubis, Grand-Cru and Noekeon were very strongly influenced by the AES or its predecessors. Q was designed using components from the AES cipher and the AES candidate Serpent. NUSH bore more than a little resemblance to the AES finalist RC6. SHACAL was the application of the well-established hash-function SHA-1 as a block cipher, with its inputs inverted. SHACAL-2, a variant of SHACAL that is based upon SHA-256, was submitted after the competition commenced. All of these ciphers suffer from weak Type 1 key schedules.

The NESSIE competition took a different approach to the AES competition, endorsing multiple ciphers [182]. These were the AES block cipher (which was not officially submitted), Camellia, and SHACAL-2. SAFER++ and RC6 were eliminated after the first round, the former because of perceived poor security, and the latter because of intellectual property rights issues, as per the CRYPTREC competition. Noekeon, NUSH and Q were broken [181].

Anubis (SPN, Homogenous, Filtered, 1C)

Anubis is a very close relative to the AES, and appears to have been proposed in response to the criticism of the AES key schedule [72] and its algebraic s-box.

Like the AES, Anubis is an homogenous SPN that adheres to the Wide-Trail Strategy. However, it acquires some of the benefits of Feistel structures by using operations that are involutions. This means that encryption and decryption follow the same algorithm, and only the order of round keys changes. The order and composition of the operations in the round functions are similar to those of the AES, with the exception that the permutation π is a transposition, as for SQUARE, rather than the ShiftRow used by the AES.

The 8×8 s-box is not constructed algebraically, as for the AES, but is instead randomly generated and filtered to meet certain conditions, including non-linear order and optimal resistance against differential and linear cryptanalysis.

The Anubis key schedule uses the wide-trail strategy to maximize diffusion. The master key is masked by the s-box and used as the pre-whitening key. This is Type 1B key schedule behavior. However, it is subsequently iterated using a modified round function, in which a ShiftColumn operation replaces the transposition. During each iteration, a round key is extracted from the result using a non-reversible Vandermonde matrix. This is Type 2C key schedule behavior. According to the guidelines, this polymorphic behavior places Anubis in the Type 1C category.

Grand Cru (SPN, Homogenous, Filtered, 1B)

Grand Cru is a derivative of the AES based on the premise that embedding several sub-ciphers, each with their own independent key set, within a single cipher makes it more robust against an attack that would break a cipher with a single key set. This is a defeatist view of thinking in line with that which says encrypting a plaintext with several ciphers leads to a superior ciphertext.

Grand Cru is very similar to the AES, except that the permutation is keyed and operates on columns and bits, as well as rows. Each round also contains a keyed byte-wise rotation which is keyed from a different key set to the permutation. As with the AES, Grand Cru is an homogenous SPN. It uses the same algebraic s-box as does the AES.

The key schedule takes a 128-bit master key, partitions it into quarters, and from each quarter produces a separate 128-bit key. No key bit in the master key is used to produce any two of these keys. The first key is expanded for use with pre- and post-whitening, the second key expanded for the round key addition, the third key expanded for the keyed permutation and the fourth key expanded for the keyed byte-wise rotation. The derivation of the four keys from the master key, and the subsequent key expansions to produce the round keys are all based upon the original AES key schedule. The Grand Cru key schedule suffers the same weaknesses as does the AES key schedule. For this reason, Grand Cru's key schedule is placed in the Type 1B category.

Noekeon (SPN, Homogenous, Algebraic, 1A/1B)

Noekeon is a block cipher with a block size and key size of 128 bits. Like many of the other NESSIE candidates, it is a homogenous SPN with many similarities to the AES. In this case, the commonalities come not directly from the AES, but instead from its ancestors Baseking and 3-Way [57]. Unlike the AES, Noekeon is a bit-sliced cipher that can be implemented using only bit-wise boolean and shift operations. As with Anubis, the operations in Noekeon are involutions so that no change is required to the algorithm between encryption and decryption.

Noekeon uses a bit-sliced 4×4 s-box that it applies to each 32-bit word in the block (the γ operation). The s-box has an alternative representation using only logical operations. The round also uses a linear mapping (θ) implemented using exclusive-ors and byte-wise rotations that mix key material, and two linear shift operations (π_1) and (π_2), which are each others' inverse. Round constants are added in each round.

Noekeon has two modes for its key schedule. The first, *direct-key* mode uses the master key as each round key. In this mode, Noekeon has a Type 1A key schedule that makes it vulnerable to related key attacks. In its second mode, the cipher algorithm is used as the key schedule, with the master key as the plaintext and a null-string as the working key. The resulting ciphertext is the new working key, which is used as each round's key. In this mode, Noekeon has a Type 1B key schedule.

NUSH (Adhoc, Homogenous, No s-boxes, 1A)

NUSH is a Russian cipher primitive used to construct a block cipher, hash function, stream cipher, and even asymmetric digital signature schemes. For the block cipher implementation of NUSH, the block size is flexible. It may be insulting to RC6 to comment on the similarities of their designs; the former possesses a more robust design and is accompanied by an intelligent analysis from its designers.

The round function is implemented using only addition, exclusive-or and rotation. For a 128-bit block, it operates on four 32-bit words a, b, c and d. The round function consists of four sub-rounds, each of which updates a and c using all four words according to $c = (c + k + b) \ggg s; a = a + (c \oplus d)$, where k is the round key, and s is a sub-round dependent constant. When the sub-round is completed, the words are cycled. The round function is iterated seventeen times for a 128-bit block.

The key schedule is simplistic and involves generating round keys and pre- and post-whitening keys by combining the master key words with known constants using modular addition. Since the constants are known, and add no security to the schedule, this is equivalent to using the master key as every round key. Consequently the key schedule is Type 1A.

Q (SPN, Homogenous, Filtered and Algebraic, 1B)

Q is a cipher that inherits strongly from the AES cipher and Serpent. Because Q lacks strong diffusion components found in Serpent and the AES, it is vulnerable to differential and linear cryptanalysis in its full eight or nine rounds [25], [119]. The structure of the Q round is $K_1 - S_1 - K_2 - S_2 - K_3 - P - S_3$ where K_i is keying layer, S_i is a layer of parallel s-boxes, and P is the AES ShiftRow diffusion operation.

There are two types of s-box mechanism deployed in Q. S_1 is the AES s-box. S_2 and S_3 are bit-sliced s-boxes, the former of which is taken from Serpent, and the latter of which is slightly modified from S_2.

Q uses pre- and post-whitening keys. The first two keying layers K_1 and K_2 in the round functions use working keys that do not vary between rounds. K_3 is a keying layer that uses an individual round key for each round. The key schedule of Q uses a reduced version of the cipher algorithm, with constants as the working keys. The first output of the algorithm is discarded; subsequent ciphertexts are used to produce pre- and post- whitening keys, the fixed working keys and eight or nine round keys (depending upon the master key length). In [25], Biham et al. note that recovery of any round key easily gives knowledge of any other round key and in fact the master key. Thus the key schedule is classified as Type 1B.

SAFER++ (SPN, Homogenous, Algebraic, 1B)

SAFER++ [156] is a relative of the AES candidate SAFER+. It has many similarities with SAFER+. Primarily the overall structure is the same: it is an homogenous SPN with two key addition layers sandwiching a row of s-boxes. Following the second row of key additions is a block-wide linear transformation. In SAFER++, this is based upon 4-point rather than 2-point Pseudo-Hadamard Transforms. The designers reduced the number of rounds to seven (for a 128-bit master key) due to their belief that the mixing had improved; however [190] asserts that the branch number of the linear transform is only five, which is poor for the large block size.

SAFER++ contains two algebraically generated 8×8 s-boxes, which are identical to those used in SAFER+.

SAFER++ uses a single 128-bit post-whitening key. Its key schedule generates fifteen or twenty-three 128-bit round keys for a 128-bit or 256-bit master key respectively. The key schedule is similar to that for SAFER+: round keys are simply rotations of an expanded key, in which one byte is omitted, combined with a pre-determined constant. However, for a 256-bit master key, odd and even round keys depend upon different halves of the master key. Since knowledge of one round key trivially allows knowledge of most of another, and in fact, of the master key, the SAFER++ key schedule is classified as Type 1B.

SHACAL (Adhoc, Heterogenous, No s-boxes, 1B)

SHACAL [89] is a straight-forward derivation of the SHA-1 hash function as a block cipher with a 160-bit block and a 512-bit key.

SHA-1 processes a message in 512-bit blocks and combines the blocks with 160-bit chaining variables, the end result of which is a 160-bit digest. By using the compression function of SHA-1, inserting the secret key as the message, and initializing the chaining variables with the 160-bit plaintext, the block cipher SHACAL results.

During each of eighty rounds, SHACAL applies a non-linear function to one of the chaining variables. Each non-linear function contains only simple logical operations (no s-boxes) and varies according to the round number: there are three functions in total, which lends the cipher a heterogenous appearance. Although the overall structure remains the same, these changes cause attackers difficulty in covering the whole cipher.

The key schedule expands the 512-bit master key to an expanded key of 2,560 bits. This is broken into eighty 32-bit round keys. The master key provides the first sixteen round keys. The remaining round keys are derived from the combination (using exclusive-or) of four previous round keys. Knowledge of any 512 consecutive round key bits, which is equivalent to the effort of a brute-force search, provides the complete expanded key. The key schedule is Type 1B.

SHACAL-2, a variant of SHACAL with a 256-bit block was introduced in the second phase of NESSIE. SHACAL-2 is based on the standardized hash function SHA-256, so its key schedule is slightly different. Again the master key provides the first sixteen round keys. Each subsequent round key depends upon four previous round keys (two of which are processed by the ω function). The ω functions are not invertible; however, knowing four round keys allows immediate calculation of some subsequent round keys, so SHACAL-2 also has a Type 1B key schedule.

2.2.4 ISO/IEC/JTC1/SC27-Korea

The cryptographic competition in Korea was low key in comparison to the European, Japanese, and American competitions. It received three entrants, one of which, SEED, was resubmitted from CRYPTREC. The remaining two ciphers, Xenon [215] and Zodiac [216], were submitted by a single designer, but Zodiac was quickly broken.

Xenon (Feistel, Homogenous, No s-boxes, 2A)

Xenon is a Feistel cipher with a simple structure. Within the Feistel round, the 64-bit block is broken into 32-bit blocks, to each of which is applied a function that uses byte rotation, multiplication, binary addition and modular addition. The result of each sub-block is then combined with the other sub-block using the exclusive-or operator, analogous to the SEED algorithm.

The key schedule generates one 64-bit pre-whitening, one 64-bit post-whitening and sixteen 64-bit round keys. The schedule contains a data-pad and a key-pad. These are initialized by alternate 32-bit words of the master key masked with constants. To generate a round key, an initial permutation is applied to the data pad, which is encrypted using two rounds of the round function, with the key pad acting as the key. The round key is produced by adding constants to the data pad. The data-pad is overwritten by the key-pad, which in turn is overwritten by the round key. Then the next round key is produced.

The round function is not reversible, so knowing one round key does not allow retrieval of the bits of other round keys or master key bits. However, knowing two consecutive 64-bit round keys provides the contents of both the data-pad and key-pad which allows retrieval of all subsequent (but not prior) round keys. The master key is not directly used as a round key, so Xenon is classified as owning a Type 2A key schedule.

Zodiac (Feistel, Homogenous, Algebraic, 2A)

Zodiac is a 16-round Feistel cipher with an extremely simple three-layered round function. In the first layer, each byte is exclusive-ored with one of its neighbors. The second layer is a simple permutation. The third layer consists of a layer of alternating s-boxes. Key addition takes part externally to the round function, and is combined with the round function's input.

Zodiac uses two algebraically constructed 8×8 s-boxes. One is generated by an inversion in a Galois field, and the other by an exponentiation in a modular number system.

Zodiac uses pre- and post-whitening. The key schedule of Zodiac is similar to that of Xenon. The difference occurs because each schedule uses the round function of their respective cipher algorithms. As with Xenon, the key schedule of Zodiac is classed as Type 2A.

2.3 Summary

This chapter evaluates the key schedule classification scheme proposed by Carter et al. in [46]. This classification divides key schedules into two partitions: those in which an attacker with a single round key can analyze the key schedule to acquire further key material; and those for which the attacker needs to re-launch an attack. While the tool is useful for evaluating the robustness of key schedules against attackers who have discovered a single round key, it suffers from some deficiencies. These include the inability to distinguish the robustness of the key schedule against an attacker in possession of one round key, and against another who has multiple round keys. Also some of the categories in the classification are either misleading or not useful.

A new classification is proposed which is strongly influenced by the classification of Carter et al. This classification also contains two partitions that reflect the robustness of the key schedule against an attacker in possession of round keys. As with the first classification, the Type 1 category is reserved for key schedules in which round keys and master keys are strongly related. A new category is introduced in the Type 2 partition: a Type 2A key schedule is robust against an attacker with a single round key, but allows an attacker with multiple round keys to acquire further information. Also the 1B and 1C categories of the Carter et al. classification are amalgamated due to their similarity; the Type 1A category is expanded to include cases where portions of master keys may be used as round keys; and the Type 2C category of the Carter et al. classification is demoted to a Type 1C following the cryptanalytic result of Kelsey et al [121] which demonstrates that key schedules in this category demonstrate similar behavior to key schedules in the Type 1A schedule.

The new classification of key schedule formed part of a broader classification of block ciphers, which also incorporated aspects of the cipher algorithm, including the overall structure, the homogeneity of the rounds, and the methods of s-box construction. Twenty-nine ciphers from the Advanced Encryption Standard, Japanese CRYPTREC, European NESSIE and Korean ISO/IEC/JTC1/SC27-Korea competitions were surveyed. Table 2.2 contains a summary of the properties that are used in this survey. Almost all of these ciphers are word-based; the exceptions, all from the AES competition, are DEAL, LOKI97, Magenta, and (because of its initial and final permutations) Serpent. Full versions of the block ciphers marked with * in the table are vulnerable to serious attacks.

It is interesting to note a couple of trends from this table. Following the success of the AES block cipher, many of the ciphers in subsequent competitions are either variants or strongly influenced by its design. This means that they use SPN structures and unfortunately have Type 1 key schedules. The ciphers of the CRYPTREC competition use algebraic s-boxes generated in the same way as the AES s-box. But the results of [56] and [176] caused the designers of some ciphers in the more recent NESSIE competition to use randomly selected, filtered s-boxes, avoiding the simple algebraic expression possessed by the AES s-box.

Heterogeneous structures are intended to complicate cryptanalysis. None of the ciphers using heterogenous structures have been broken, but there are too few of these to draw definite conclusions. There appears to be no other pattern inherent in this classification that indicates which ciphers have been broken. Intuitively, ciphers with adhoc structures are more likely to broken since they are less studied, but the difficulty of analyzing new structures may act as a deterrent to effective cryptanalysis. Also, there is no clear pattern to indicate that ciphers with Type 1 key schedules are especially vulnerable to complete breaks. But as evidence in the next chapter indicates, they nevertheless decrease the strength of these ciphers.

Cipher	Cipher Structure	Heterogenous Cipher	S-boxes	Key Schedule Class
Advanced Encryption Standard				
Cast-256	Feistel-1	√	Filtered	2C
Crypton	SPN		Filtered	1B
DEAL	Feistel-1		Filtered	1B
DFC	Feistel-1		Random	2B
E2	Feistel-1		Algebraic	2C
Frog*	Adhoc		Random	2C
HPC	Adhoc		Keyed	1B
LOKI97*	Feistel-1		Algebraic	2C
Magenta*	Feistel-1		None	1A
Mars	Feistel-3	√	Filtered	2C
Rijndael	SPN		Algebraic	1B
RC6	Feistel-2		None	2C
SAFER+	SPN		Algebraic	1B
Serpent	SPN		Filtered	2C
Twofish	Feistel-1		Keyed	2C
Japanese IPA CRYPTREC				
Camellia	Feistel-1	√	Algebraic	1B
CipherUnicorn-A	Adhoc		Algebraic	1B
Hierocrypt-3	SPN		Algebraic	2C
SC2000	Feistel-1 / SPN		Algebraic	2C
SEED	Feistel-1		Algebraic	2C
NESSIE				
Anubis	SPN		Filtered	1C
Grand-Cru	SPN		Filtered	1B
Noekeon*	SPN		Algebraic	1A/1B
NUSH*	Adhoc		None	1A
Q*	SPN		Algebraic	1B
SAFER++	SPN		Algebraic	1B
SHACAL	Adhoc	√	None	1B
SHACAL-2	Adhoc	√	None	1B
ISO/IEC/JTC1/SC27-Korea				
Xenon	Feistel-1		None	2A
Zodiac*	Feistel-1		Algebraic	2A

Table 2.2: Summary of Properties of Contemporary Block Ciphers

Chapter 3

An Improved Key Schedule for the AES

The Advanced Encryption Standard (AES) [62] is the American standard for block ciphers, and has also been adopted de facto worldwide so its robustness against attacks is essential. To date, no practical attacks succeeding against full-round AES have been published. However, the key schedule of the AES is categorized by both Carter et al. [45] and the scheme developed in Chapter 2 of this work, as Type 1B. Consequently, the AES may be vulnerable to attacks that: target the key schedule directly to identify round keys or the master key; or indirectly exploit the key schedule to launch an attack on the cipher algorithm. Some cryptographers have gone as far as to deride research on block cipher key schedules [27], [61]. This is despite the fact that published block ciphers are vulnerable to known attacks that exploit the weaknesses of their key schedules [14], [59], [130]. Weak key schedules also affect the security of ciphers used in hashing mode [129], [132]. The nature of block cipher attacks is one of exploiting the weakest parts of the cipher system.

To avoid the potential vulnerabilities of the AES key schedule, we propose a different approach to its design that puts its security on an equal standing with that of the cipher algorithm. Although not as fast as the original on Intel Pentium processors, the new AES key schedule is nevertheless efficient and in constrained environments may even be faster, due to the smaller code footprint. There is frequently a trade-off between speed and security; however, there is no reason to choose a fast key schedule that possesses exploitable weaknesses, when there are reasonable compromises that offer greatly increased security at a slight reduction in speed.

The analysis of weak key schedules has led to guidelines for robust key schedule design that borrows from well-known and accepted design principles for block cipher algorithms. Our design follows these principles.

In Section 3.1, we review the principles that aid the production of strong key schedules. In Section 3.2, we review serious security weaknesses in the key schedule of AES which assist published attacks on reduced-round versions. These key schedule weaknesses may be used directly as targets of attack or to extend other attacks. A more secure key schedule for the AES cipher is presented and analysed in Section 3.3. The throughput of the new key schedule is diminished compared to the original, but justification is given, in terms of speed and security, as to why the new version of the key schedule is more suitable for the AES standard than the original. This is performed in the context of the speeds of the key schedules of the other short-listed AES cipher candidates, which are still in use by parties concerned that selecting a single AES candidate leads to a single point of failure. Finally, Section 3.4 summarizes the contribution of this chapter.

3.1 Block Cipher Key Schedules

Most block ciphers' key schedules expand a short master key into a longer expanded key, which is used to generate round keys for each round in the block cipher. With the use of Type 2 key schedules, the round keys appear to bear no relation either to other round keys or to the master key. Consequently an attacker who discovers a round key will not find any shortcuts to discovering the master key. This strategy also prevents related-key attacks [14], since Type 2 related master keys do not generate strongly related round keys.

Knudsen [131] listed four necessary but not sufficient properties for secure Feistel ciphers. Two of these, *no simple relations* and *all keys are equally good*, are achievable with strong key schedules. The remaining two properties relate to protection against differential and linear cryptanalysis, which are not directly related to key schedules.

Strong key schedules should also have the following properties:

Non-invertibility A block cipher acts as a one-way function when its master key is unknown. Using components from a cipher algorithm in the associated key schedule is a common practice for providing the schedule with non-invertibility [6], [61], [149], [192], [193], [204]. Reuse of components improves the ease of implementation and reduces code size in software.

Freedom From Bit Leakage Reduction of bit leakage between round keys and the master key increases the complexity of some attacks on block ciphers, by forcing the attacker to do more work to retrieve additional round keys. Examples include differential and linear cryptanalysis of the Data Encryption Standard (DES) [26], [158] and the AES attacks in [72] where the authors summarize that "*Some of our attacks make use of the relations between expanded key bytes and would have a higher complexity if these relations did not exist.*" Using master key bits directly in round keys is the degenerate case of bit leakage. Leakage of information between adjacent round keys is directly prevented by non-invertibility in the key schedule, so the two properties are closely coupled.

Efficient Implementation The cipher algorithm and the key schedule should complement each other in both security and implementation facets. It is advantageous that the execution time of a key schedule be of the same order of speed as the cipher itself, as is the case with all the short-listed AES candidates. This is particularly important for use in wireless paradigms or when the cipher is used as a hash function.

3.2 The Advanced Encryption Standard

The AES block cipher [62] allows master keys of 128, 192 and 256 bits. For these key sizes, the AES iterates ten, twelve and fourteen rounds of the round function respectively. Each round uses a single 128-bit round key.

The round function is based upon the SQUARE round function [60], and the theoretical underpinning provided by the wide-trail strategy, as discussed in Section A.2. It operates on a 128-bit block conceptually divided into a four by four array of bytes.

In the round function, the array undergoes four operations:

- Byte Substitution (γ), a non-linear operation in which each byte is independently transformed by the AES 8×8 s-box.
- Shift Row (π), a linear operation in which each row (indexed from zero) of the array is shifted to the left byte-wise, by the row number.
- Mix Column (θ), a linear operation in which each 32-bit column is used as input to a Maximum Distance Separable (MDS) matrix. The 32-bit output of the MDS directly replaces its input.
- Key Addition (σ), in which each byte is combined with a corresponding round key byte using the exclusive-or operation.

The round function can be summarized as:

$$b = \sigma(k) \cdot \theta \cdot \pi \cdot \gamma(a)$$

where a is the input, b is the output, and k is the round key.

In addition to n iterations of the round function, there is a pre-round key addition to the plaintext. From the master key, the key schedule is required to generate $n + 1$ round keys. In the next section, the key schedule is analysed in detail, to highlight weaknesses that may be introduced to the cipher algorithm through the key schedule.

3.2.1 Description of the AES Key Schedule

The AES key schedule is based on 32-bit words, with the master key used to directly supply the initial words. The remaining words in the *expanded key* are generated through an iterated process. Groups of four adjacent 32-bit words in the expanded key are concatenated to produce the 128-bit round keys. Table 3.1 shows the key schedule algorithm for all three permitted key sizes. In this algorithm, MK is the master key, RK_i is the round key for round i, $W[i]$ is a 32-bit word, $\gamma(x)$ is the parallel application of four s-boxes to the bytes in the 32-bit words, $\lll$ is a byte rotation, % is a modulus reduction operation and C_r is a predefined round constant.

```
128-bit key:
  $W[0..3] = RK_0 = MK$
  $rs = 1, \#rk = 10$
192-bit:
  $W[0..5] = MK$
  $W[6] \quad = W[2] \oplus W[5]$
  $W[7] \quad = W[3] \oplus W[6]$

  $RK_0 = W[0..3], RK_1 = W[4..7]$
  $rs = 2, \#rk = 12$
256-bit:
  $W[0..7] = MK$

  $RK_0 = W[0..3], RK_1 = W[4..7]$
  $rs = 2, \#rk = 14$

for $(r = rs$ to $\#rk)$ {
  128-bit key, 192/256-bit key (r % 2 = 0):
    $W[4r] \quad = W[4r-4] \oplus \gamma(W[4r-1] \lll 8) \oplus C_r$
  256-bit key (r % 2 != 0):
    $W[4r] \quad = W[4r-4] \oplus \gamma(W[4r-1])$
  192-bit key (r % 2 != 0):
    $W[4r] \quad = W[4r-4] \oplus W[4r-1]$

  all cases:
    $W[4r+1] = W[4r-3] \oplus W[4r]$

  128/256-bit key, 192-bit key (r % 2 = 0):
    $W[4r+2] = W[4r-2] \oplus W[4r+1]$
  192-bit key (r % 2 != 0):
    $W[4r+2] = W[4r-2] \oplus \gamma(W[4r+1] \lll 8) \oplus C_i$

  all cases:
    $W[4r+3] = W[4r-1] \oplus W[4r+2]$

    $RK_r = W[4r] \parallel W[4r+1] \parallel W[4r+2] \parallel W[4r+3]$
}
```

Table 3.1: The AES Key Schedule

3.2.2 Previous Cryptanalysis

This section overviews previous cryptanalysis of the AES in which the attacks exploit its key schedule. Other types of cryptanalysis on AES, to which its key schedule is not relevant (for example, XLS), are described in generic terms in Appendix A. No attacks on full-round AES have been found and published

The most prominent attack on reduced-round AES is the integral (formerly SQUARE) attack [139], as described in Appendix A. This is a chosen-plaintext attack that exploits the byte-oriented structure of AES to recover the last round key. Because of the bit-leakage of the key schedule, all other round keys, and indeed the master key can be recovered from knowledge of this key. The basic attack uses a three round characteristic to attack four rounds with a complexity of 2^9 texts and 2^9 operations [62]. The three round characteristic can be used to launch an attack on six rounds, by guessing four bytes each of the first and sixth round key, and an additional byte of the fifth round key. Integral cryptanalysis relies on a key distinguishing feature to determine the accuracy of key byte guesses. This feature does not apply until at least the fourth round; therefore, for ciphers owning a Type 2 key schedule, the attack would not automatically provide any of the first three round keys, or the master key.

In [72], Ferguson et al. note that for a 192-bit key, the characteristic can be used to launch an attack on seven rounds, by guessing the sixteen bytes of the seventh round key. This leads to a complexity of 2^{200}. However, they find that bit leakage provide two bytes of the sixth round key and a byte of the fifth round key, based on the seventh round key guess. This gives a complexity of 2^{176}, which is quicker than a brute force attack. If the key schedule of the AES were Type 2, this attack would not successfully apply to seven rounds of the cipher.

In [72], Ferguson et al. also identify *key splitting* but are unable to use it to launch an attack. By guessing the fourteen bytes through which one half of the expanded key interacts with the other, the master key can be *split* into halves, each of which controls half the round keys. This suggests a meet-in-the-middle attack, but the non-linearity of the key schedule has prevented its discovery.

In [72], Ferguson et al. describe a 9-round related-key attack against AES with a 256-bit master key. This attack is a variant of integral cryptanalysis, and uses 256 related master keys for which the fourth round keys differ in a single byte. Plaintext differences are used to cancel out differences in the earlier round keys, so that three bytes in the sixth round key balance. This allows the attacker to guess key bytes of the last three rounds, and to use the balance in the sixth round key as verification of the guesses. By guessing twenty-seven bytes of the master key, bit leakage allows acquisition of sixty-six bytes of the expanded key. If the key schedule were Type 2, the differences between the texts could not be tracked throughout the nine rounds; insufficient key bytes could be guessed to lead to a more efficient attack than brute force. Ferguson et al. note the small number of non-linear elements and slow diffusion in the key schedule structure when compared to the cipher algorithm.

3.2.3 Our Analysis

The overriding security concern with the AES key schedule is the fact that, given knowledge of a round key (or part of a round key), other round keys are partially or fully derivable. In fact, the AES cipher is the only one of the five finalists in the AES competition to be categorized with a Type 1 key schedule.

We now explicitly define the bit leakage in the AES key schedule. From the key schedule algorithm given in Table 3.1, it can be observed that each $W[i]$ value is related to previous values $W[k], k < i$. Given a 128-bit key, one example shows that knowledge of half a round key (for example, $W[41]$ and $W[42]$) immediately determines a quarter of the previous round key ($W[38] = W[41] \oplus W[42]$). The iterative nature of the key schedule is used to enhance the efficiency of the implementation, but in this case, it is too simplistic, leading to the bit leakage problem. Additionally, all master key bits are not involved in the generation of round key bits until at least $W[6]$ (in the case of 128-bit keys). It is clear that the key schedule does not satisfy the properties of "non-invertibility" or "freedom from bit leakage" defined in Section 3.1.

Having defined the problem we wish to avoid, we outline our approach to the new design. In general the cipher designer strives to obtain rapid mixing of input bits (confusion), and also to ensure that each input bit affects each output bit (diffusion). Unlike the key schedule, the AES cipher algorithm achieves these properties elegantly by the fourth round.

To measure the confusion and diffusion properties of the key schedule proposal against those of the original key schedule, we use two basic statistical tests available in the CryptX [87] statistical package.

The frequency test measures the bit mixing property, a basic measure which is fundamental in achieving bit confusion. The result of this test is a single

Round	Freq (p)	SAC (D*)
2	0.0000	96.083
3	0.0048	20.687
4	0.7560	1.183

Table 3.2: CryptX Statistical Results for the AES Cipher Algorithm

Round Key	Freq (p)	SAC (D*)
1	0.0000	125.053
2	0.0000	105.433
3	0.0000	72.563
4	0.0000	46.858
5	0.0593	31.840
6	0.0000	28.057
7	0.0000	28.153
8	0.0034	28.237
9	0.0000	28.161
10	0.0110	28.215

Table 3.3: CryptX Statistical Results for the AES Key Schedule

probability (p) value where a small p indicates a significant result. A p value greater than 0.01/0.001 indicates that bit mixing is satisfied at the 1%/0.1% critical level.

The Strict Avalanche Criterion (SAC) test measures the bit diffusion property. This test checks that a one-bit change in the input block produces, on average, changes to half the bits in the output block, which is a good measure of bit diffusion. The resulting test statistic is the Kolmogorov-Smirnoff statistic denoted by D*. See [87] for more details. A D* value less than 1.628/1.949 indicates that bit diffusion is satisfied at the 1%/0.1% critical level.

Test results for the AES encryption algorithm are detailed in Table 3.2, indicating that both the frequency and SAC tests are satisfied after four rounds. The reason that the SAC test is not satisfied until the fourth round of AES encryption is that the MixColumn operation is removed from the final round. The AES authors state that the MixColumn function was omitted from the final round "*in order to make the cipher and its inverse more similar in structure*". If the MixColumn function is included in the final round, then the SAC test is satisfied after three rounds, as shown in Table 3.4.

In comparison to the AES cipher algorithm, the key schedule's test results,

Round	Freq (p)	SAC (D*)
2	0.0000	21.113
3	0.2663	1.282
4	0.3110	1.347

Table 3.4: CryptX Statistical Results for the AES Cipher Algorithm including Final Round MixColumn

detailed in Table 3.3, show that the key schedule is less successful in achieving rapid bit diffusion and confusion. Row i, $1 \leq i \leq 10$, gives the test results after applying the frequency test and the SAC test to the Round i key. The majority of round keys do not gain complete bit mixing. The process does not satisfy the SAC test for any of the round keys.

3.2.4 AES Implementation Metrics

The AES key schedule is very fast. Two metrics for expressing this are the number of clock cycles required to generate the round keys and the ratio of the number of blocks that can be encrypted by the cipher algorithm in the time it takes to execute the key schedule. As it is a relative measure, the second metric is a more successful method to demonstrate key agility, which is important when an algorithm needs to be frequently rekeyed.

Table 3.5 lists both metrics for the five AES short-listed block ciphers benchmarked with 128-bit keys. These metrics are taken from two third-party implementers [11], [80], who both use Intel Pentium processors to profile the implementations. The first column under each reference in Table 3.5 shows the number of cycles to complete the key schedule, while the second column shows the number of cycles to encrypt a block. The third column under each reference is the ratio of key setup time to single block encryption time. In this table, AES is shown as the cipher with the fastest key schedule, and also with the best keying to encryption ratio. Given the deficiencies explained previously, we believe that it is too fast. The other ciphers are classified by both Carter et al. and our scheme in Section 2.2, as possessing Type 2 key schedules. These have been designed with a better balance between key setup and encryption time, and consequently between speed and security.

	Reference [11]			*Reference [80]*		
	Key	Encrypt	K:E	Key	Encrypt	K:E
MARS	6934	656	**7.5**	2118	364	**5.8**
RC6	2278	318	**7.2**	1697	269	**6.3**
AES	1289	805	**1.6**	215	362	**0.6**
Serpent	6944	1261	**5.5**	1300	953	**1.4**
Twofish	9263	780	**11.9**	8520	366	**23.3**

Table 3.5: Bench-marking the Key Schedules of the AES Finalists

3.3 A New AES Key Schedule Proposal

Efficient bit mixing and bit diffusion techniques have already been developed for the AES cipher algorithm (as is shown in Table 3.2), so it seems logical to include these techniques in the production of a strong key schedule.

A proposed AES key schedule is detailed in Table 3.6.

```
128-bit key:
  $\#rk = 10$
192-bit key:
  $\#rk = 12$
256-bit key:
  $\#rk = 14$

for $r = 0$ to $\#rk$
  for $j = 0$ to 15
    128-bit key:
      $a_j = b_j = \mathrm{MK}_j \oplus S[16r + j]$
    192-bit key:
      $a_j = \mathrm{MK}_j \oplus S[16r + j] \oplus S[MK_{j+8}]$
      $b_j = \mathrm{MK}_{j+8} \oplus S[16r + j] \oplus S[MK_j]$
    256-bit key:
      $a_j = \mathrm{MK}_j \oplus S[16r + j] \oplus S[MK_{j+16}]$
      $b_j = \mathrm{MK}_{j+16} \oplus S[16r + j] \oplus S[MK_j]$
  for $i = 0$ to 2
    $a = \sigma(b) \cdot \theta \cdot \pi \cdot \gamma(a)$
  $KR_r = a$
```

Table 3.6: Proposed Key Schedule for the AES

The schedule is required to produce eleven, thirteen, or fifteen round keys for master keys MR of 128, 192 and 256 bits respectively (the number of round keys is denoted by $\#rk$).

The master key is not used directly as inputs to the AES round function. Instead, it is used to populate two 128-bit values a and b, which are dependent on the master key bytes and round number, so differ for each round key.

For any given round key, each byte $a_j, 0 \leq j \leq 15$ in a is initialized from the combination of the corresponding master key byte MK_j, and a constant that is non-linearly dependent upon the byte index j and the round key number r. The non-linearity is generated through the use of the AES s-box. When the master key has 192 or 256 bits, an extra key byte, masked by the s-box, is added to the a value. This allows more than 128 bits to be incorporated into the value.

Each byte $b_j, 0 \leq j \leq 15$ in a 128-bit value b is initialized in almost the same way as for a_j. The position of the corresponding master key byte is offset by a constant dependent on the length of the key. These offsets are zero, eight, and sixteen bytes for 128-, 192- and 256-bit master keys respectively. The offset for the additional masked master key byte is zero.

Round key $RK_r, 0 \leq r \leq 15$ is populated with the ciphertext of AES cipher algorithm, reduced to three rounds, using a as the plaintext and b as the key.

The addition of the s-box constants at the start of each round key generation not only isolates each round key from the others, but also breaks up possible weak keys (for example, if all the master key bytes were identical).

Table 3.7 gives CryptX test results for generating a single round key, where the number of AES encryption rounds used in the generation is given in the first column. Results are included for 128-, 192-, and 256-bit master keys. These results indicate that, for each round key generated using the proposed key schedule, complete bit mixing and bit diffusion is reached after three rounds, independently of the chosen key size. Increasing the number of rounds will slow the key schedule without necessarily increasing the security. Using fewer than three rounds will not achieve the necessary levels of mixing and diffusion.

The most important achievement of the proposed key schedule is that the frequency and SAC tests are satisfied, thus ensuring complete bit confusion and diffusion is achieved for every round key. This is in contrast to the AES key schedule, which does not satisfy the frequency test in the majority of round keys generated, and does not satisfy the SAC test for any round key.

3.3.1 Implementation of the Proposed Key Schedule

For a 128-bit master key, the proposed key schedule executes thirty-three cipher encryption rounds to generate the eleven round keys, which are used in ten rounds of encryption. So for any implementation, there is a theoretical key schedule setup to encryption ratio of 3.3. Respectively for 192- and 256-bit keys, this falls to 3.25 and 3.21. This is still faster than most of the other short-listed AES candidates. These figures are shown in Table 3.8, along with benchmarks and the key encryption setup to encryption ratio for the standardized key schedule.

3.3.2 Security Analysis of the Proposed Key Schedule

The major problem with the standardized AES key schedule is bit leakage. In our proposal, as each round key is generated independently, there is minimal bit leakage between round keys, and the master key is not used directly as a round key. Non-invertibility is achieved by encrypting the master key using a well-analysed function, with the key being used both as the data block and the key block. According to the classifications of Carter et al. and that of Chapter 2, the proposed key schedule is categorized as the strongest (for Carter et al., Type 2B; for the classification of Chapter 2, Type 2C) which exhibits much stronger security properties than the Type 1B category to which the standardized schedule belongs.

In contrast to the current AES key schedule, even if an entire 128-bit round key is known, it is infeasible to invert the three-round function and retrieve the master key. It is not possible to obtain round key bits from one round using material purely from another.

Related-key attacks As evidenced by Table 3.7, there is a high diffusion of each master key bit across each round key. This is useful in preventing related-key attacks, since altering a single bit in the master key changes approximately half the bits in each round key. Consequently it is difficult for an attacker to coerce the relationship between two master keys to exist as relationships of any form between the corresponding round keys. In the standardized key schedule, the master keys are used directly in the pre-round addition, and for 192- and 256-bit keys, also in the first round, providing an attacker with a foothold to launching a related key attack. Since the master key is not used directly in our proposal, this opportunity does not exist.

Round	Freq (p)	SAC (D*)
128-bit master key		
2	0.1557	15.775
3	0.8757	1.212
4	0.3498	1.689
192-bit master key		
2	0.5593	11.891
3	0.2002	1.268
4	0.2041	1.155
256-bit master key		
2	0.8900	21.158
3	0.6766	1.196
4	0.9029	1.189

Table 3.7: CryptX Statistical Results for Proposed Key Schedule

	Reference [11]	*Reference [80]*	Our proposal
128-bit master key			
Key Setup	1289	215	-
Encrypt block	805	362	-
K:E	**1.60**	**0.59**	**3.30**
192-bit master key			
Key Setup	2000	215	-
Encrypt block	981	428	-
K:E	**2.04**	**0.50**	**3.25**
256-bit master key			
Key Setup	2591	288	-
Encrypt block	1155	503	-
K:E	**2.24**	**0.57**	**3.21**

Table 3.8: Comparison of the Speed of the AES and Proposed Key Schedules

Additionally the strong non-linearity of the proposed key schedule and the addition of constants to each of the master key bytes aids in preventing conventional related-key attacks [14].

The primary attack on the cipher algorithm is integral cryptanalysis [62]. The attack can be applied to the key schedule as a differential related-key attack. In this attack, a single plaintext is repeatedly encrypted under a set of 256 master keys, a Λ-set [60], where a single byte of each text differs such that the binary sum of the texts is zero. An example of this is given on the standardized key schedule in [72]. However, we could find no way to practically exploit this with the new key schedule, due to its strong non-linearity. Guessing bytes in a 128-bit master key gives an equivalent number of bytes in each round key, rather than giving free bytes as per the attack in [72]. For 192- and 256-bit master keys, two master key bytes must be guessed to determine each round key byte.

Differential and Linear Cryptanalysis Conventional techniques applied to three rounds of the AES cipher algorithm, such as differential cryptanalysis and linear cryptanalysis, do not hold for direct attacks upon the proposed key schedule. These attacks typically require collections of chosen- or known-plaintexts. The analogs of these texts, as keys, are not available under the conditions of related-key attacks.

The minimum assumption for a key-schedule attack is that, given a known difference between round keys, there is a mapping to an exploitable feature within the cipher algorithm. We could find no such mapping, so it is difficult to determine how conventional differential or linear cryptanalysis could weaken the proposed key schedule.

Weak Keys It is unavoidable that a key schedule that determines a 128-bit round key from a 192- or 256-bit master key generates key collisions that produce identical 128-bit round keys. The best that can be hoped is that the key schedule does not generate large classes of weak keys, from which key collisions can be engineered. Since every round key in the proposed key schedule satisfies the SAC test, this means that if one bit of the master key is changed, approximately half the bits of the round key will change. This is the best diffusion possible, so any collisions that do occur are the one-off result of two randomly-chosen master keys, and do not form a class of weak keys.

Aiding attacks on the cipher algorithm Many attacks on the cipher algorithm use weaknesses in the key schedule to reduce the complexity of the attack. A prime example of this is the integral attack on reduced-round AES cipher versions which, when combined with the bit leakage property of the key schedule, recovers all the round keys including the master key. Since our proposed key schedule does not allow bit leakage, its adoption would prevent this extension of the attack.

Related Cipher Attacks Appendix A describes the new class of attacks by Wu [226] under the umbrella name of the "Related Cipher Attack". This attack works on the same principle as slide attacks [32]: by gaining the text produced by the cipher just a few rounds (round m) from the end of the cipher (round n), the rounds between m and n can be treated as a reduced version of the cipher, which is more easily broken, exposing the round keys.

Slide attacks rely on homogenous round functions to acquire the ciphertext at round m and round n, using 'slid pairs'. Whereas, related cipher attacks rely on using two versions of the cipher, one of which has m rounds, and the other n rounds. The AES is a perfect example of this type of cipher, since the number of the rounds differs slightly with key lengths.

However, the original key schedule of the AES avoids the related-cipher attack by combining, with each round key, a constant that is dependent upon the length of the key. Thus the changing constant ensures that the round key for the mth round is different between the two cipher instances, and no 'slid pair' analog will be obtained.

In [226], Wu asks the reader to contemplate using two versions of the AES with the modified key schedule, one of which uses a 192-bit master key (and therefore twelve rounds), and the other a 256-bit master key (with fourteen rounds). His attack relies on concatenating a 64-bit value three times to form the 192-bit master key, and on concatenating the same value again to form the 256-bit master key. Concatenating a key of length k for which $\gcd(k, 64) \neq k$ will not work. This is because the initialization stage of the key schedule differs between the two master key lengths, in that it offsets the location of the byte of the master key chosen to form each a and b value by 64 bits.

If the resulting two rounds can be broken with less than 2^{64} effort, the cipher can be broken with less than the designed bit security of the key. The attack can be defeated by changing the constants that control the offsets by which master key bytes are selected. If the constants are co-prime, concatenating the 64-bit value to form a master key will still produce different round keys under each cipher. However this attack is valid only under very constrained conditions, so the proposal comes with the recommendation that all bits of the master key should be randomly generated. This is a common operational parameter for use with all block ciphers.

Algebraic Attacks Courtois and Pierprzyk [56] hint that ciphers that possess key schedules that are similar to the cipher algorithm may be targeted by a specific form of XLS (see Section A.6), but this theory is never publicly developed. We are unable to find any evidence that our key schedule contributes to a reduction in complexity of the XLS attack on the AES block cipher.

We believe the proposed key schedule to be safe from conventional methods of cryptanalysis.

3.4 Summary

In this chapter, we described and analyzed the AES key schedule. We also presented and justified an alternative and strengthened key schedule, based upon the standard information principles of confusion and diffusion, which still permeate cryptographic design.

Block cipher key schedules have not received the same focus as cipher algorithm design in the past, but are nonetheless vital to the overall security of the cipher system. A weak schedule can provide a means through which an otherwise secure cipher system is attacked.

We demonstrated that the current key schedule does not satisfy the bit frequency mixing test for the majority of round keys and does not satisfy the avalanche (bit diffusion) test for any of the round keys. This indicates poor pseudo-randomness properties in the key schedule. There is also a high level of bit leakage between round keys which is exploited in some theoretical attacks on reduced round AES.

In contrast, for the proposed key schedule, every round key is independent from each other round key, preventing bit leakage. Each round key also satisfies the frequency and SAC tests indicating good pseudo-randomness properties.

The key schedule proposal is between approximately one and a half to six times slower than the original, depending upon the key length and the implementation used for comparison. But we believe this decrease in speed is justified when held in the context of improved security. Using the new proposal, the round keys for AES can be created in the time it takes to encrypt three AES blocks. This ratio is faster than three of the five AES finalists (the exceptions being the AES standard and Serpent). Since none of the candidates were disqualified on the basis of key agility, we do not believe our proposal is unduly slow, or prevents its use in hash functions or applications with high key turn-over.

The primary goal of a symmetric block cipher is to provide security, with its speed of implementation a very important secondary goal. This proposal achieves increased security by limiting the extent to which previously published attacks can exploit the key schedule, at a modest increase in initial key setup times.

In the two and a half years since the publication of the proposal, the only attack suggested to date has been the Related Cipher Attack by Wu [226]. This attack relies on owning multiple instances of the cipher, each with a different master key length, and each with a master key constructed from a repeated 64-bit seed. This is an unrealistic model of attack, and as the master key is not constructed randomly, certainly not consistent with how the cipher should be used. No other attacks on the AES cipher with the modified key schedule have been forthcoming.

Chapter 4

Stream Ciphers

Traditionally the realm of stream cipher design has focussed on bit-based linear feedback shift registers (LFSRs), which are well studied and satisfy common statistical criteria. Non-linearity is instilled into the keystream by irregular clocking, or a non-linear filter or combiner. Some stream ciphers include multiple mechanisms, such as LILI-128 [214] which uses both irregular clocking and a non-linear filter.

Bit-based stream ciphers are for the most part, blazingly fast in hardware. They are also excruciatingly slow in software when compared to block ciphers. For example, the Advanced Encryption Standard (AES) [62] outputs blocks of 128 bits at the cost of 14 cycles/byte on the Intel Pentium 3 [151], while LILI-128 outputs single bits at 580 cycles/byte on the same machine [189].

There is a tendency in some cryptographers to promote pro-stream cipher sophistry along the lines of software performance and security being mutually exclusive, or the forerunner in one arena (hardware speed) not needing to compete in others (software speed). Aside from slowing the uptake of symmetric cryptography, this is misleading for two reasons. Military domains have ready access to hardware; commercial companies with casual users of cryptography and finite budgets do not. Personal computers are ubiquitous but seldom equipped with cryptographic accelerators. Secondly, there are increasingly many ciphers that are good in software *and* hardware and not yet insecure. New (non-experimental) ciphers need to justify themselves in the three areas of security, hardware performance and software performance.

There is one piece of very telling evidence against slow but secure bit-based stream ciphers in the public domain. Inarguably the most used stream cipher is RC4 [5]. It is word-based rather than bit-based, and is faster in software than in hardware. Because it is also significantly faster than most block ciphers (at seven cycles/byte), it is deployed more frequently, despite a trail of literature ([74] [155] [200] [220]) acting as a beacon to its deficiencies. Slower ciphers with a better track record of security are eschewed in favour of speed.

The balance is being redressed with a new generation of word-based stream ciphers, in which the boundaries with block ciphers are blurred. The size of the blocks that stream ciphers generate are by and large unconstrained (for example, RC4 outputs blocks of eight bits, but Turing [201] outputs blocks of 160 bits). Design methodologies have to some extent remained similar; for example building LFSRs over GF(2^8) or GF(2^{32}) rather than GF(2), but many traditional cryptanalytic methods do not apply without substantial modification.

Word-based stream ciphers based on block ciphers may be naturally faster than block ciphers. Block cipher cryptanalysis takes the form of dynamic analysis, in which an attacker can manipulate the inputs to the block cipher regularly. However, a stream cipher takes a key and an initialization vector and produces a long length of keystream. A cryptanalyst attacks stream ciphers using static analysis, in which the cipher output is passively examined. This strategy appears to be less successful in attacking ciphers than dynamic analysis, so directs design of stream ciphers towards complex key initializations but very lightweight update functions [73], which have an intrinsic efficiency advantage over the complex or highly iterated round functions of block ciphers.

Until recently, there have been only a handful of word-based stream ciphers, including SEAL [198], SOBER [93] and RC4. In Section 4.1 of this chapter, we survey recent word-based ciphers, describing in each case the influence that block ciphers have played upon their their development. In Section 4.2, we provide a brief description of stream cipher attack techniques as they apply to word-based stream ciphers. In Section 4.3 we add to the literature analyzing the RC4 block cipher by refuting recent claims that the first byte in each keystream is biased.

4.1 Modern Word Based Stream Ciphers

This section describes the design strategies for a selection of word-based ciphers, with a focus on how block ciphers have influenced them. It examines how these strategies affect the performance and security of the ciphers. The ciphers include HC-256 [227], Helix [73], Hiji-Bij-Bij [203], MUGI [224], Rabbit [34], RC4 [5], Scream [88], SNOW [67], and Turing [201]. The SOBER [94] family shares much in common with Turing, so is not discussed here. Discussion of BMGL [92] which is in reality a filtered block cipher running in an OFB-like mode of operation is deferred to Section 4.2.

Word-based stream ciphers are allowed, by virtue of generating large keystream blocks, increased complexity and consequently flexibility in design. This has resulted in a wide range of styles in modern stream ciphers, when compared to the LFSR bit-based ciphers of the past.

Some stream cipher designers have elected to choose conservative strategies, and as a result, ciphers like MUGI, SNOW and Turing retain a close resemblance to bit-based LFSR ciphers. SNOW and Turing both use a single LFSR, whereas MUGI uses an NLFSR.

Ciphers like Helix and Scream eschew shift registers in favor of a state modified by a block-like non-linear component. Their update functions resemble the rounds of block ciphers. The key initialization of Helix uses a Feistel structure to chain key words. Scream iterates its round function in the same way as does a block cipher.

Ciphers based upon dynamic permutations bear little resemblance either to LFSR-based stream ciphers or block ciphers. Due to their simplicity, they have the potential to run extremely quickly. They use indices and counters that point to locations in a state table. The indices select the inputs for the update function, and a monotonic counter selects a target word for modification. The use of a counter ensures that all words in the table are modified in a given time-frame. RC4 is a dynamic permutation. HC-256 is not a guaranteed permutation, but is strongly influenced by RC4.

Other ciphers have unique design strategies that borrow components and structures from other stream ciphers, or from block ciphers. Two examples are Hiji-Bij-Bij, which is based upon cellular automata, and Rabbit, which uses chaotic mapping for non-linearity.

Update Functions of many word-based stream ciphers borrow heavily from the round function of the AES cipher. In particular, they may borrow the AES s-box to provide non-linearity or the MixColumn operation, with its MDS matrix that provides efficient diffusion.

That the AES algorithm was successful in winning a well-scrutinized competition gives impetus to borrowing components for new stream ciphers. That it comes with a proof of security against two standard and powerful methods of block cipher cryptanalysis also gives it important credentials. But the stream ciphers surveyed here, which borrow components from the AES, do not follow the Wide-Trail Strategy that underlies the success of the AES algorithm. Borrowing components from the AES for use in stream ciphers may be considered a cheap design technique that improperly leverages analysis while removing it from its proper context. When it occurs, it should be accompanied by fresh analysis that justifies the inclusion of those components within the new cipher.

There is work by Fuller et al. [76] and Youssef et al. [232] that show redundancy within algebraically generated s-boxes such as the AES s-box, although the cryptographic significance is not clear. Also, the work by Courtois et al. [56] on using the s-box to produce over-defined equations (see Section 4.2.9) calls into question the wisdom of basing too many cryptographic primitives on a single technique for providing non-linearity. Ciphers like Turing and Dragon (see Chapter 7) construct their own s-boxes using well-established boolean theory; this originality is to be commended.

Rekeying Strategies are frequently overlooked in block and stream ciphers (see Section 4.2.8 for more on related-key attacks on stream ciphers). However, almost all of the candidates surveyed here adopt a sensible keying strategy that utilizes both a master key and an IV to protect against related-key attacks. The exceptions are RC4 (for which keying issues are well documented), Rabbit and Hiji-Bij-Bij, all of which omit incorporation of an IV. This means their keys must be more carefully managed to prevent resynchronization attacks.

In some ciphers, the rekeying strategy is partitioned, such that in the first phase, the key is added to the state and processed, and in the second phase, the IV is injected and mixed. It can be argued that as the IV is publicly available, it requires less mixing. When the cipher is rekeyed with an IV, the keying strategy omits the first phase, improving IV-keying agility. Ciphers that follow this two-phase rekeying strategy include Helix, MUGI, Scream and Turing.

All but two ciphers (RC4 and Turing) make use of the update function to mix the state during the rekeying strategy. As with block ciphers, reuse of the update function provides the advantages of a smaller implementation footprint and leverage of existing security analysis. There are few reasons not to reuse it – it is designed to be optimal in terms of confusion and diffusion, which are quantities equally required in the key initialization algorithm. However, comparative to the state size, the output generated by the update function is small. This is necessary, otherwise the keystream provides too much insight into the internal state.

Consequently the larger the state to be keyed, the more inefficient this process becomes, which may affect key agility. Some ciphers use a modified update function which incorporates a larger output filter to populate the state in a shorter time. One example is Dragon (see Chapter 7), which during keystream generation, uses an output filter of 64 bits. During key initialization, the output filter generates 128 bits to populate the 1,024 bit state. As a result its key agility is better than most of the ciphers surveyed here. The key agility of a cipher that is most penalized by a large state belongs to HC-256, which needs to iterate the update function 4,096 times to mix each element in its tables twice. Another example of poor key agility occurs in MUGI, which is scrutinized more closely in Chapter 6.

The update function can be reused within the rekeying strategy in a number of ways. Ciphers like Helix, MUGI and Scream use the update function to generate the contents for the initial state. Other ciphers populate the state by a different means, but use the update function at the end of the process as a simple and effective way to mix it; Hiji-Bij-Bij and Rabbit populate the state directly with the key prior to mixing, whereas HC-256 applies a more complicated method involving SHA-2 functions and chaining. Chaining, also used by Scream in its rekeying strategy, is an effective way to rob an attacker of control over the initial state. This is because small changes in the master key are amplified in the internal state by the chaining; however, non-linear mixing needs to occur after the chaining has taken place, to prevent the attacker from utilizing the amplification. An ideal way to do this is to invoke the update function on the state.

4.1.1 HC-256

HC-256 obtains high throughput by trading memory for speed, and utilizing the super-scalar features of Intel Pentium family.

HC-256 is a simple cipher that operates on a 2,048 element table of 32-bit words. During the update function, the table is logically split into halves, P and Q. The location of the counter i determines the roles of P and Q. When i points to an element in P, Q is treated as a series of 8×32 s-boxes. The keystream is formed by combining the values of four words from Q indexed by the bytes of a word P_x. The index x is calculated using the values of four words in P and one in Q. All indices are calculated using binary and modular addition, and fixed rotations. When the counter iterates to the first element of Q, the roles of P and Q in the update function are swapped.

Block cipher aspects of the HC-256 update function include the treatment of half of the state table as a series of s-boxes, and using primitive operations to provide diffusion across the table elements in selecting indices.

The key initialization of HC-256 uses a 256-bit master key and an additional 256-bit IV. These are concatenated to form the first sixteen words of a 2,560 word array W. The remaining words W_i are generated using chaining, in which the SHA-2 functions are applied to previous words in the array. The initialization is concluded by iterating the update function 4,096 times, with the resulting keystream being discarded. Consequently the key initialization modifies each element of the large table twice, causing key agility issues. IV rekeying is also slow, as the master key and IV are combined before the update function is called for the first time.

The throughput of HC-256 is 4.1 cycles/byte on the Pentium 4, which makes it the second fastest of the ciphers surveyed here. But the state size of HC-256 is 65,536 bits, coupled with an additional 16,384 bits for key setup. This prohibits its use in constrained environments.

4.1.2 Helix

Helix's update function is similar to a block cipher's iterated round function. The update function consists of interwoven helices that operate upon five strands of the 160-bit state.

Each helix contains ten iterations of a very simple round function operating on a target (T) and source (S) word: $T = T \otimes S; S \lll C_r$ where the $\otimes$ operator is either binary addition ($\oplus$) or modular addition ($+$) in GF(2^{32}); C_r is a round dependent constant. There are two special round function instances: key injection $T = (T + K) \otimes S$, and the plaintext injection $T = (T \oplus P) \otimes S$. Key injection uses two round keys: one comes from a working key, the other from the combination of a partial working key, the IV, and a round constant. At the end of the round function, the source S and target T words cycle right by one strand.

Helix outputs 32-bit blocks of keystream and is unique among the ciphers surveyed here in that it provides a MAC for integrity. When the final keystream word has been emitted, the MAC is generated by modifying the first helix strand with a constant, and updating the state eight times, discarding the keystream. Four further keystream words are produced and combined to produce a 128-bit MAC tag. The rationale of its authors in providing a MAC for a small cost is that significant vulnerabilities are introduced by encrypting without authenticating, but traditionally, authentication has been operationally expensive. However, injecting the plaintext into the state has lead to a differential-style attack by Muller [175], which is discussed further in Section 4.2.7.

The round function of Helix is inspired by block ciphers like RC5 [196] and RC6 [197]. It contains no s-boxes, instead relying upon the weak non-linearity of the $\boxplus$ addition relative to the $\oplus$ operator, amplified through many iterations of the round function. The inclusion of a MAC is inspired by block cipher modes of operation that provide integrity for almost no cost. One example is Integrity-Aware Cipher-Block Chaining (IACBC) [116].

Helix has a complex rekeying strategy to defeat dynamic analysis. The internal state is populated using a master key padded to 256 bits. Together the master key, and an encoding of its length are used to populate the state. As with Turing, this encoding serves to ensure that different length keys do not generate equivalent states. The eight-word working key is produced by iterating the update function eight times in a block-cipher Feistel-type structure, so that later words are chained to all of their predecessors. The update function is then iterated a further eight times, with the resultant keystream being discarded. This is an efficient strategy that provides high key agility. Helix does not use the IV in the key initialization algorithm; instead, it incorporates it directly into the keystream generation algorithm, so that IV rekeying is exceptionally fast. The strategy of injecting the IV late is a weakness that aids in the attack by Muller [175].

Helix is optimized for 32-bit platforms, through its use of simple operations that are readily parallelized on super-scalar architectures. Its small state is wholly containable within the registers of the Intel Pentium family, although overheads may generate some register pressure. This means that Helix is easily capable of encrypting at seven cycles/byte on modern platforms.

4.1.3 Hiji-Bij-Bij

Hiji-Bij-Bij [203] is in some ways a conventional word-based stream cipher, but replaces LFSRs with non-chaotic cellular automata (CA). It has a linear state of 512 bits, representing two 256-bit cellular automata and a 128-bit non-linear state (NLS). The rationale for using CA is that the shift of two sequences obtained from a CA is exponential, rather than linear, in their length. Hiji-Bij-Bij is unusual in that it can be used in synchronous or self-synchronous mode.

Hiji-Bij-Bij's update function has two stages. The first stage invokes the AES s-box on each byte in the 128-bit NLS, which it folds into a 32-bit variable. It adds this variable into each 32-bit block of the NLS and rotates each block by a constant. It transposes the NLS (by interpreting it as a 4×32 matrix) and reapplies the AES s-box to each byte. This component of the update function resembles a block cipher SPN layer, using the structure S-P-S, where S is the s-box layer, and P is the permutation (in this case, the transposition of the NLS). One example of a block cipher using this structure is E2 (see Section 2.2).

The second stage modifies the cellular automata using rules 90 and 150. Each rule involves exclusive-oring the contents of the automata with two copies of itself (shifted one bit to the left, and one bit to the right respectively). A rule-dependent constant is added to each automata.

The cipher emits the 128-bit keystream word, which consists of a combination of the NLS words with fixed automata words. The automata words not used in the keystream are folded into the NLS.

Hiji-Bij-Bij has a very simple rekeying strategy. It takes a key of 128 or 256 bits and initializes the linear state as $K\|\overline{K}\|\overline{K}\|K$ or $K\|\overline{K}$ respectively, where $\overline{K}$ is the complement of K. The key strategy compresses the key to 64-bit K_{64} by exclusive-oring key segments, and initializes the non-linear state as $K_{64}\|\overline{K_{64}}$. The complex update function is iterated sixteen times, causing key agility problems, and the last four keystream blocks are concatenated and combined with the automata linear state. No IV is incorporated into the scheme.

The designer of Hiji-Bij-Bij acknowledges that cellular automata are much less efficient than LFSRs, but judges the trade-off towards security as necessary. Given that the cipher is one of the slowest surveyed, and that it was demolished by a guess and determine attack ([127], see Section 4.2.4), this viewpoint seems incorrect.

4.1.4 MUGI

MUGI borrows heavily from the stream cipher PANAMA [58], in that its two components – a 1,024-bit buffer based upon a non-linear feedback shift register, and a 192-bit non-linear internal state – provide feedback to each other. If this feedback is ignored, the cipher bears strong resemblance to a traditional LFSR based cipher with a heavy filter function. Another way to view the cipher is as a single 19-stage NLFSR in which each stage consists of 64 bits.

The update function of MUGI includes a target-heavy Feistel primitive that contains two invocations of a modified 64-bit AES round, in which the RowShift operation is removed, and the outputs of the two MixColumn operations are permuted. The function provides a high-degree of non-linearity to the cipher. The buffer is clocked like a standard sixteen-stage LFSR, with additional feedback coming from stage one of the internal state, and two modified intermediate stages from within the buffer. The 64-bit keystream word comes from one of the three stages of the internal state.

MUGI borrows many of its security claims from the provable security of the AES block function. This gives it immunity to highly correlated linear approximations, related-key attacks based upon integral cryptanalysis, and variants of differential and linear cryptanalysis. However, the complex structure of MUGI is difficult to analyze.

MUGI uses a key of 128 bits, and an IV of 128 bits. It has a complicated rekeying strategy that includes iterating the update function forty-eight times for full rekeying, and thirty-two times for IV-rekeying. This causes serious key agility problems, which are addressed in Chapter 6.

MUGI's performance is poor on 32-bit architectures, principally because it is designed for 64-bit architectures. This makes it the slowest of the ciphers surveyed here.

4.1.5 Rabbit

Rabbit [34] is a very simple cipher based upon chaotic maps. It aims to combine the random properties of real-valued chaotic maps with the speed and precision of integer representations. It has an internal state of 513 bits. This includes eight 32-bit variables, eight 32-bit counters and one counter carry bit. The keystream output consists of eight 16-bit words, where each word is a simple linear mix of one half of each of two state variables.

The update function of Rabbit consists of modifying each variable $x_i, 0 \leq i < 8$ according to the equation $x_{i+1} = g_i + (g_{i-1} \lll a_i) + (g_{i-2} \lll b_i)$, where g is the quadratic function $g_i = ((x_i + c_i)^2 \oplus ((x_i + c_i)^2 \gg 32)) \bmod 2^{32}$. Counters c_i are updated according to $c_i = c_i + a_i + \phi_i \bmod 2^{32}$ where a_i is a constant and ϕ_i is a carry bit based upon counter additions. This approach has no analog in block ciphers, and overlooks lessons learned by the designers of the RC6 and MARS block ciphers about the inefficiencies of variable rotation and multiplication on the Intel Pentium 4. The keystream output consists of eight 16-bit words, where each word is a simple linear mix of different halves of two state variables.

The key initialization algorithm of Rabbit divides the 128-bit key supplied by the user and places each 16-bit fragment directly into one half of each of two state variables and one half of each of two counters. It iterates the internal state four times and updates each counter by combining it linearly with the contents of a single state variable. Rabbit is highly key-agile due to a lightweight rekeying scheme. However, the initially published version does not permit IVs.

On the Pentium III, Rabbit is the fastest of the surveyed ciphers. It uses primitive operations which account for its efficiency. However, in one update function, it performs sixteen multiplication operations. On the Pentium 4, and pre-Pentium Intel processors, it will perform more slowly than the stated 3.7 cycles/byte.

4.1.6 RC4

RC4 is the most widely used stream cipher in software. Its popularity is due to its speed, size and simplicity. It has been incorporated into mainstream cryptographic protocols including Secure Sockets Layer (SSL) [187] and Wireless Equivalent Privacy (WEP) [100]. RC4 is the oldest stream cipher presented in this section, predating most block and stream ciphers, so has little in common with other word-based ciphers. It has a parameterized word size, which is typically eight bits. This represents a respectable trade-off between efficiency and security. RC4 has an internal table S of 2^n bytes, a counter i and an index j. The size of the keystream block matches the parameter n.

RC4's update function in shown in Figure 4.1. In the function, the counter i is incremented modulo 2^n, and table element indexed by i is added to the j index, also modulo 2^n. The elements referenced by the indices are swapped, and added to form the index of the element returned as the keystream word.

```
i = (i + 1) mod 256
j = (j + S[i]) mod 256

swap(S[i], S[j])
z = S[S[i] + S [j]] mod 256
```

Figure 4.1: RC4 Update function

RC4's key initialization algorithm does not incorporate the use of an IV. This has lead to related-key and distinguishing attacks upon RC4, as described in the next section. The problematic RC4 key initialization algorithm is shown in Figure 4.2. It is very similar to the update function, but not identical, so that object code size almost doubles by its inclusion. Each element in the table is initialized with the value of its index. In each round, the value of the index j is incremented by the value of the element to which the counter i points, and the value of the key byte indexed by i modulo the length l of the key in bytes. The value of the j^{th} element is swapped with the value of the element referenced by the counter i, which is incremented.

```
for i = 0 to 255
    S[i] = i

j = 0
for i = 0 to 255
    j = (j + S[i] + K[i mod l]) mod 256
    swap(S[i], S[j])
i = j = 0
```

Figure 4.2: RC4 Key Initialization Algorithm

During the initialization phase, the counter i touches each element in the S table once. Assuming that the values of the j index are uniformly distributed, it covers approximately 63% of the elements in S, the remainder of which are swapped only once. As a result, Roos [200] observed that for at least 1 in every

256 RC4 keys, part of the key is strongly correlated with the first byte of the keystream. This reduces exhaustive search of the RC4 key by up to 5 bits. Shortly after, Wagner [220] discovered another class of weak keys. Grosul and Wallach [86] described a related-key attack for keys of length greater than two kilobytes. A related-key attack on RC4 by Fluhrer, Mantin and Shamir [74] is discussed in Section 4.2.8.

RC4 has been subjected to many other attempts at cryptanalysis. Golic [82] and Fluhrer and McGrew [75] reported small correlations between bytes in RC4's keystream. Mantin and Shamir [155] discovered a bias in the second byte of the RC4 keystream (discussed in Section 4.2.2). Mihalejevic [167] devised a time-memory tradeoff attack on RC4 with 2^{52} data, 2^{40} memory and 2^{76} processing time, after a one-off preprocessing time of 2^{80} operations. This is discussed further in Section 4.2.1.

Because RC4 has a small state that fits into the L1 cache of Pentium processors, and uses simple byte-based operations, it is one of the fastest stream ciphers available, operating at under 10 cycles/byte on most architectures. However, it is an unusual symmetric cipher in that it is inherently serial and does not lend itself to parallel application in either hardware or software. Also its key initialization algorithm features no IV, meaning that it needs to be executed in totality upon rekeying.

4.1.7 Scream

Scream is a family of word-based stream ciphers based upon the word-based stream cipher SEAL [198]. The Scream family – consisting of Scream-0, Scream, and Scream-F – aims to avoid security flaws in SEAL, while maintaining a very high throughput in software for 32-bit processors.

The internal state in Scream consists of 3×128-bit blocks and a masking table W, which uses 16×128-bit elements for a total of 2,432 bits. Scream outputs 128-bit blocks of keystream. Although Scream uses a 128-bit master key, the stated design strength is 64 bits. This approach contrasts with the consensus used by cryptographers, in which the security of a cipher equals the length of the master key.

In each cipher of the Scream family, approximations to the non-linear component are disguised by linear masking aspects. The designers show how the masking can be removed after observing about 2^{72} bits of keystream, and how this can be used to form the basis of a distinguishing attack [88]. The designers describe a low-diffusion attack using fixed s-boxes, in which output bytes of the non-linear function F depend on only six out of eight bytes. This attack style is discussed further in Section 4.2.3.

At the heart of Scream's update function F are two 64-bit modified AES sub-functions G_1 and G_2, each of which operate upon two 128-bit registers, x and y. Each sub-function consists of the AES ByteSub, RowShift and MixColumn operations. In Scream, and Scream-F, the ByteSub s-boxes are key dependent; in Scream-0, the AES s-box is used.

The non-linear function F is a hybrid Feistel ladder/SP network that is iterated as per a block cipher, each time producing 128 bits of keystream. During each iteration, x is modified by the function $x = F(x + y) + z$ over GF(2), and y is rotated by a few bytes to protect against low diffusion attacks. After iteration of sixteen rounds, y and z, and one element of the masking table W are modified using the F function. The update function outputs 128 bits of keystream based on a simple linear combination of x and a single element of W. The influence of block ciphers upon the update function is evident. The function strongly resembles the round function of a block cipher, including the use of the Feistel ladder, the incorporation of the AES components, and the iteration over sixteen rounds.

Scream uses a very simple rekeying strategy in which it chains elements of the masking table W. It modifies the 128-bit master key by a constant and passes it through the non-linear function F five times (denoted F^5). It uses the result as the first element of the table. Each subsequent element in the 16-element table is derived from the result of F^4 on the previous element in the table. Consequently, this process iterates the F function sixty-five times. The supplied 128-bit IV modifies the linear masking table further and invokes F an additional fourteen times. By introducing the IV late into the rekeying scheme, IV agility is improved, but the subsequent invocations of the F function protect the cipher from related-IV attacks.

Rekeying using a master key involves seventy-nine iterations of the non-linear function, including fourteen iterations that mix the IV. As a result, Scream suffers from the loss of key-agility.

The update function of Scream appears to be very fast in software on newer Intel platforms, primarily through the use of look-up tables along the lines of an AES implementation [178]. Additionally, Scream has a state slightly larger than two kilobytes. This implies it will be quite slow on sub-32-bit platforms, and also on the Pentium 4, which has a small L1 cache.

4.1.8 SNOW

SNOW is the most traditional of the designs surveyed here, in that it uses a simple 16-stage LFSR coupled with a modified finite summation generator. However, the LFSR operates in GF(2^{32}) rather than the bit-based GF(2). This allows SNOW to be fast in software and both fast and compact in hardware. The total state size of the cipher, including both the LFSR and two 32-bit registers is 576 bits. SNOW outputs 32-bit keystream blocks.

During each invocation of SNOW's update function, the LFSR is regularly clocked, and the results of the summation generator are modified by rotation and a virtual 32×32 s-box. The 8×8 s-box components of the virtual s-box are algebraically constructed using a non-linear mapping of a polynomial base combined with a bit-based permutation. SNOW-2, which remedies several problems with the original cipher, constructs the 32-bit s-box using the AES s-box and a MixColumn operation, albeit one that uses a different polynomial to the AES cipher.

SNOW uses a key of 128 or 256 bits and an optional 64-bit IV. The key initialization algorithm places the key into consecutive LFSR stages (in groups of four or eight, depending upon key sizes) until all sixteen stages have key material. It modifies between one quarter and half of the stages, depending upon key length, by a small constant, and exclusive-ors the IV (if present) into two non-adjacent stages. The update function is invoked thirty-two times if an IV is supplied, or sixty-four times if it is not. Because the master key and IV are combined prior to the iterations of the update function, IV-rekeying is slow in comparison to Helix and Turing.

Hawkes and Rose [95] broke SNOW using a guess and determine attack with a complexity of 2^{100} keystream bits and 2^{224} operations. They achieved this by exploiting the single stage input to the FSM and the particular feedback polynomial. Additionally, Coppersmith, Halevi and Jutla [49] discovered a distinguishing attack based on a correlation to the FSM. This requires 2^{100} keystream bits and 2^{100} operations.

The designers of SNOW responded to the attack by defining SNOW-2 [68]. The FSM takes two input words from the LFSR, which uses a new feedback polynomial. The size of the IV has increased to 128 bits, and the AES s-box is used to change the internal state of the FSM.

SNOW and SNOW-2 are designed to be fast on modern 32-bit processors, and to be usable in constrained devices where memory is at a premium. They use relatively cheap operations, have modest state sizes, and provide high throughput in software.

4.1.9 Turing

Turing is heavily based upon the NESSIE candidate SOBER-t32 [94]. The design of Turing inherits much of the analysis performed on SOBER-t32 but addresses the criticisms found in [43], [44], [66] and [69]. The state of Turing is wholly contained within a regularly clocked LFSR consisting of 17×32-bit stages. It is modified by a keyed non-linear filter that uses many elements from block ciphers. Turing targets 32-bit architectures but outputs 160-bit keystream blocks. For this reason, the LFSR is clocked five times within each update function.

Turing's non-linear filter contains two five-input pseudo-hadamard transforms (PHTs), separated by a row of five 32×32 s-boxes. The first PHT is provided with inputs from five stages in the LFSR, which is then clocked. The PHT output is applied to the s-boxes; their outputs are in turn used by the second PHT. Following execution of the second PHT, the LFSR is clocked four times to provide whitening material to the filter, and to involve more than half of the LFSR stages within the production of the resulting 160-bit keystream block. Like Scream, Turing uses keyed s-boxes: its virtual 32×32 s-boxes are composed from random 8×8 and heuristically constructed 8×32 s-boxes. This avoids algebraic attacks while maintaining good statistical properties. The influence from block ciphers is clear: the use of PHTs within the filter function is inspired by the SAFER [157] block cipher, although the former uses 5-input rather than 2-input PHTs; also keyed s-boxes were popularized by the AES Twofish [205] candidate.

The design criteria for Turing apparently stop guess and determine attacks, through the optimization of a full positive difference set to provide the set of LFSR taps. The design of the non-linear filter is claimed to limit the correlation of low-order approximations, thus prohibiting Courtois' algebraic attack [51] and the linear masking attack [49].

Turing's rekeying strategy makes much use of block cipher components, including keyed s-boxes and PHTs. Permissible keys range in size from 32 to 256 bits, with an accompanying IV of not more than 352 bits. Together the key and IV may total as many as 384 bits. In the rekeying strategy, the user provided key is altered at the byte and word levels, by applications of fixed s-boxes and PHTs respectively. These mixed key words are placed in the LFSR and the key-dependent s-boxes. The IV is mixed using the fixed s-boxes and prepended to the mixed key words in the LFSR. A constant modified by the lengths of the IV and master key is appended to the LFSR, and each of the words are chained using addition. Each word is applied to, and overwritten by the application of the 32×32 s-box. Finally the LFSR is modified by a 17-PHT. The final mixing stage that combines the key and IV makes it difficult to launch a related-IV attack, when compared to the lightweight mixing scheme employed by Helix. Non-linear key loading thwarts related-key attacks, and the 17-PHT used in the rekeying strategy causes small changes in related IVs to be hugely magnified.

The key initialization algorithm processes the master key first, followed by the initialization vector, and then a mixing stage than combines both together. This means that master key and IV rekeying are comparatively efficient.

Turing is one of the slower stream ciphers described here because of its complexity, despite the fact that the large size of the key stream output allows a more complex function for a given cycle-to-byte ratio.

4.1.10 Summary

Modern word-based ciphers have a wide range of design strategies. This includes basing ciphers upon LFSRs, or upon block-cipher-like update functions. Other ciphers are constructed as dynamic permutations, or using novel concepts that bear little relation to either block or stream ciphers.

LFSR-based ciphers reviewed in this section include MUGI, SNOW and Turing. MUGI uses a non-autonomous non-linear 1,024 bit LFSR in conjunction with a 192-bit non-linear state. Its filter function is based upon a modified AES round. SNOW and SNOW-2 are conventional word-based stream ciphers that use 512-bit regularly clocked LFSRs in conjunction with a finite state machine (FSM). The AES s-box is used to modify the FSM in SNOW-2. Turing is based upon a 544-bit LFSR and a block-cipher like non-linear function that makes extensive use of keyed s-boxes and PHTs.

RC4 is a dynamic permutation. It is one of the few ciphers listed here that bears little resemblance to a block cipher. Although not a dynamic permutation, HC-256 strongly resembles RC4, and contains two tables. It iterates methodically through the tables, treating one as a series of 8×32 s-boxes. The rekeying strategy is influenced by hash functions, through its use of SHA-256 functions.

Scream has strong ties to the AES block cipher. Its internal state is modified by a 16-round Feistel ladder containing a modified AES round function. Helix also draws strongly from the block cipher paradigm. Its 160-bit state is updated by a block cipher style function that incorporates primitive operations such as addition and rotation, but excludes s-boxes.

Hiji-Bij-Bij is based upon additive cellular automata. It contains a 512-bit state based on the automata and a separate 128-bit NLS. It makes extensive use of the AES s-box in an SPN update sub-function on the NLS.

Rabbit is based on chaotic maps. Its 513-bit FSM is updated by a function constructed from addition, multiplication and rotation, as per the RC6 block cipher, but its construction is unlike any block cipher.

Comparison of Ciphers

Table 4.1 compares the key features of each of the stream ciphers included in this section. Traditional style stream ciphers are based on LFSRs. However, in some ciphers, the feedback may be modified by a non-linear filter, in which case the cipher is classified as a NLFSR. Other ciphers eschew LFSRs in favour of finite state machines (FSM) or Dynamic Permutations (DP). For comparative purposes, the Dragon cipher discussed in Chapter 7 is also included here.

The table is ordered by speed, in terms of the number of cycles required to generate one byte of keystream (the slowest ciphers are towards the end of the table). The table also shows the key and IV sizes of each cipher, along with the block size of the output filter and the memory used by the cipher state. It indicates the key agility of the cipher in terms of full- and IV-only rekeying. It shows whether the cipher is flexible enough to be readily used on both general purpose (32-bit) machines and in constrained environments such as on smart cards. The table indicates whether the cipher shares code between key initialization and keystream generation routines, thus reducing the cipher's footprint.

Cipher	Class	Key (bits)	IV (bits)	Output (bits)	Memory (bits)	Speed (cycles)	Key Agile	IV Agile	Flexible Usage	Code Reuse
Rabbit	FSM	128	-	128	513	3.7	Yes	-	Yes	Yes
HC-256	FSM	256	256	32	81,920	4.1	No	No	No	Yes
Scream	FSM	128	128	128	2,432	4.9	No	Yes	Yes	Yes
SNOW	LFSR	128 – 256	128	32	576	5.5	No	No	Yes	Yes
Dragon	NLFSR	256	256	64	1,088	6.7	Yes	No	Yes	Yes
Helix	FSM	≤ 256	128	32	160	7.0	Yes	Yes	Yes	Yes
RC4	DP	128	-	8	2,048	7.2	Yes	-	Yes	No
Turing	LFSR	≤ 256	≤ 352	160	544	9.2	Yes	Yes	Yes	No
Hiji-Bij-Bij	FSM	128	-	128	640	≥ 16	No	-	Yes	Yes
MUGI	NLFSR	128	128	64	1,216	25.2	No	No	No	Yes

Table 4.1: Summary of Modern Word Based Stream Ciphers

4.2 Survey of Attacks on Stream Ciphers

Of the word-based ciphers reviewed in Section 4.1, Hiji-Bij-Bij and Helix were broken. The organizers of the NESSIE competition declined to accept any of the stream cipher candidates for standardization on the basis that all suffered security flaws [181].

Here we perform a brief review of the techniques used to identify flaws in these stream ciphers. Those techniques which have not been successfully applied to any ciphers reviewed in this chapter, include correlation attacks, discussed in Section 4.2.6; and algebraic XL attacks, discussed in Section 4.2.9.

Attacks which have been successfully applied to ciphers, but without recovery of key material, include statistical attacks and distinguishers in Section 4.2.2, and linear masking attacks, which are used to generate distinguishers, in Section 4.2.3.

The attacks which have been most successful include time-memory-data trade-off attacks, discussed in Section 4.2.1; divide and conquer attacks, discussed in Section 4.2.5; guess and determine attacks, discussed in Section 4.2.4; and related-key attacks, discussed in Section 4.2.8.

4.2.1 Time-Memory-Data Trade-Off Attacks

Time-memory-data trade-off attacks [30] rely on pre-computation to reduce the effort required for a key recovery attack on a keystream. The attack comprises two steps. The first, the preprocessing step, involves the attacker calculating a table of keys or internal states and corresponding keystream prefixes. The table is ordered by prefix. The second step involves observing keystreams, using a sliding window, and attempting to match each against a prefix in the table. If the match is successful, then with some likelihood the internal state is known by reading the opposing entry in the table. The parameters in an attack are time (T), memory (M), and amount of data (D). Generally, $T \times M^2 \times D^2 = S^2$ where S is the state space of the cipher, and $D^2 \leq T$ [30]. The pre-computation time P is equal to $S \div D$.

Particularly valuable to time-memory-data trade-off attacks is sampling resistance [31], which measures the difficulty of finding states that produce rare but recognizable output sequences. For ciphers with low sampling resistance, time-memory-data attacks require many fewer disk probes to store the state-sequences pairs.

RC4 is well recognized for its low sampling resistance [154]. Mihalejevic used the low sampling resistance of RC4 to derive a time-memory-data attack [167], by finding a bias given the three-byte master key prefix of [0, 0, 253]. He calculated that, with a probability of 0.05, master keys with this prefix produced keystream output prefixed with [0, 0]. By storing only output sequences with this prefix, an adversary is able to reduce expensive disk probing in a time-memory-data trade-off attack. This accelerates the attack by a factor of 2^{18}, allowing an attack with 2^{52} data, 2^{40} memory and 2^{76} processing time, after a one-off preprocessing time of 2^{80} operations.

Another time-memory-data trade-off attack is applicable to BMGL [92]. BMGL is a simple construction based upon the AES block cipher running in Key Feedback Mode (KFM). In KFM, the ciphertext $x_i = f(p_i, k_i)$ is used as the key k_{i+1} in the next iteration. The keystream c_i is produced from x_i using *hardcore functions*, in which the output is provably indistinguishable from random assuming that the underlying function f (in this case, the AES round function) is secure. However, KFM mode is a dual to OFB, and as such, is vulnerable to the same time-memory-data trade-off attacks that occur when the block size of BMGL is less than double its key size [181].

4.2.2 Statistical Attacks and Distinguishers

A stream cipher should generate a pseudo-random keystream that does not exhibit any statistical anomalies or biases. One example of an attack that exploits statistical properties is the Berlekamp-Massey algorithm [13]. This algorithm is successful with keystreams that have low linear complexity, defined as the number of stages in the shortest LFSR that can generate the keystream. For a cipher of linear complexity l, the Berlekamp-Massey algorithm can construct an equivalent LFSR, from which the initial state is trivially recoverable, given only $2l$ bits of keystream. Secure ciphers should have linear complexities that exceed the length of the maximum keystream produced under a single key.

It is easy to verify the basic statistical properties of a stream cipher by applying a statistical package such as CRYPT-X [87] to a sufficient number of keystreams generated using random keys. However, it is not possible at this stage to automatically determine whether the statistical properties of the cipher lead to the development of distinguishers, which differentiate the cipher's output from a random keystream. Distinguishers for iterated block ciphers can usually be converted to key recovery attacks, but this does not necessarily apply to stream ciphers unless the bias caused by the distinguisher is particularly strong.

One famous example of a distinguishing attack on a stream cipher is the attack on RC4 by Mantin and Shamir [155], who discovered an unconditional bias in the second byte of RC4 keystream, which contains zero twice as many times as would be expected in a random source of bytes. For n = 8, zero occurs with probability $\frac{1}{128}$. From this, Mantin and Shamir devised a distinguisher, which with 256 output words from unrelated keys, differentiates RC4 from a random keystream. A related attack, also in [155], recovers the second plaintext word of a broadcast message encrypted under multiple keys. Both of these attacks are due to RC4's simplistic key initialization algorithm. Distinguishing attacks on RC4 are further explored in Section 4.3, and a specific type of distinguisher applied to a class of stream ciphers is discussed in Section 4.2.3.

Almost all of the stream cipher entrants to the NESSIE competition — Leviathan, LILI-128, RC4, SNOW, SOBER-t16 and SOBER-t32 — are vulnerable to distinguishing attacks [181]. BMGL, the exception, was broken quickly by other means, so may also be vulnerable.

4.2.3 Linear Masking Attacks

Coppersmith, Halevi and Jutla [49] describe a generic attack technique that uses linear masking.

The technique applies to stream ciphers composed of a non-linear component that looks like a block-cipher primitive, and a linear process such as an LFSR. Potential ciphers to which the attack may apply include SEAL, PANAMA, and of the ciphers surveyed in this chapter, MUGI, Scream and SNOW. In these ciphers the non-linear component decorrelates distant states, while the linear process masks correlated local states.

The attack works by identifying approximations to the non-linear state that exhibit some bias. The linear process masks this bias, so the attack involves finding some combination of consecutive steps such that the masking vanishes. Coppersmith et al. [49] apply this to SNOW. They cause linear masking to vanish over six consecutive steps by identifying and applying to the observed bits, polynomials that are divisible by the LFSR feedback polynomial. The probability of the non-linear approximation is $2^{-8.3}$; applied to six consecutive steps, it becomes

$2^{-49.8}$. The workload for the distinguishing attack is around 2^{100} operations and requires 2^{95} texts. It does not pose a serious threat to the security of SNOW.

The authors describe a variant of this technique, termed the low diffusion attack. This technique involves identifying dependencies between input and output bits of the non-linear function over a single step. They guess some bits, iterate the function, and check that the dependent bits have consistent states, using the above technique to remove the effect of the linear masking. They apply this to Scream-0 using 2^{43} output bytes and 2^{80} operations. This is a form of guess and determine attack, which is discussed in the next section.

4.2.4 Guess and Determine Attacks

In a guess and determine attack, the cryptanalyst acquires an amount of keystream and makes assumptions about values within the internal state of the cipher that was used to produce it. These assumptions are treated as correct, and in a successful attack, allow the cryptanalyst to calculate other values within the state, with a complexity less than that required for a brute-force search. Once the entire state has been deduced, this candidate state is used to produce an amount of keystream that is cross-checked against the known keystream. If the two keystreams mismatch, the candidate state is rejected, and the assumptions altered, before the attack is attempted once more.

Guess and determine attacks have proven effective against the SNOW, which was attacked by Hawkes and Rose [95] with a complexity of 2^{224} operations and 2^{95} bits of keystream. The attack involved guessing values for the two 32-bit registers within SNOW's FSM at two different points in time, and using these guesses to deduce the contents of its 512-bit LFSR. Hiji-Bij-Bij was attacked in both its synchronous and self-synchronous modes by Klima [127] with, in the former case, a complexity of 2^{140} (for a 256-bit master key) and thirty-four consecutive keystream blocks. A guess and determine attack was launched against SOBER-t32, by De Canniére [43], with an operational complexity of 2^{304}, exceeding the brute-force complexity of 2^{255}.

A guess and determine attack is presented on registers two and three of the Alpha1 bit-based stream cipher in Section 5.3.4. A guess-and-determine attack on Helix is discussed in Section 4.2.7.

4.2.5 Divide and Conquer Attacks

A divide and conquer attack partitions a stream cipher into components. By attacking and finding a solution to the most vulnerable component first, the cipher can be reduced, and the attack reiterated until the entire state is deduced. A divide and conquer attack may incorporate many other techniques of attack. For one example of this attack against a bit-based stream cipher, see Chapter 5.

For bit-based ciphers that use n LFSRs, a successful attack may have complexity $\sum_{i=1}^{n}(2^{L_i}-1)$, compared to a brute-force complexity of $\prod_{i=1}^{n}(2^{L_i}-1)$. With word-based stream ciphers, the attacks become more difficult, particularly in the case where large homogenous NLFSRs or LFSRs are used in conjunction with a small number of internal state registers. The components in the cipher may not be independent, which means isolating them for individual attacks becomes problematic. When the size of one of the components is larger than the design strength of the cipher, a divide and conquer attack is not likely to lead to a complexity that is better than exhaustive search. One example of a word-based stream cipher exhibiting immunity in both of these ways is MUGI. The properties that give it this immunity are discussed further in Chapter 6.

4.2.6 Correlation Attacks

Stream ciphers with multiple components that are strongly or weakly correlated to each other may be vulnerable to correlation attacks. An example of a correlation attack on a bit-based cipher composed of four LFSRs can be seen in Chapter 5, in which one of the registers is only weakly correlated to the keystream. However, correlation attacks have proved ineffective on word-based stream ciphers to date. This may be because the components of many of these ciphers are not autonomous, leading to difficulties in determining correlation measures.

4.2.7 Block Cipher Style Attacks

In general, block cipher style attacks are applied only as analysis tools to block cipher-like components within stream ciphers, to gain confidence about their confusion and diffusion properties. See, for example, the analysis of MUGI in [223] and [63]. This is because of the difference in the attack models for the cipher. An attacker can use a chosen-plaintext to control intermediate values during a block cipher encryption, but it provides no control over the internals of a synchronous stream cipher.

Self-synchronous stream ciphers may be more vulnerable to block-cipher style attacks. One example is Helix, which was subjected to a differential attack by Muller [175]. Firstly, as a self-synchronous cipher, Helix uses plaintext to generate a MAC. Secondly, as noted in Section 4.1, the nonce (IV) is injected into the cipher state during the keystream generation, rather than the rekeying phase. Together, these form the cipher's undoing. By encrypting two messages P and P' with a difference in word i, such that $\Delta P_{i-1} = P_{i-1} \oplus P'_{i-1}$, a distinguisher can be formed, by observing the difference $\Delta' C_i$, with a probability that depends on the characteristic $\Delta \to \Delta'$. The characteristic can be used in conjunction with a guess and determine attack that requires the cryptanalyst to guess 96 bits of internal state. The attack provides the remaining 64 bits of the internal state with a complexity of 2^{99} and a text requirement of 2^{18} bits. Note that the attack scenario involves encrypting both messages with the same key and nonce, which is forbidden under the standard usage of stream ciphers. Muller converts this to an attack in which the difference is induced in the nonce, rather than the plaintext, with a complexity of 2^{120} bits, for any key length. Although the specification of Helix requires that the key is changed after encryption of each 2^{67} bits, this is a successful certificational attack on key sizes of 128 bits and greater.

4.2.8 Related-Key Attacks

Many cryptographic protocols use stream ciphers over unreliable channels, or impose limits on the amounts of ciphertext encrypted with a single key. As a result, the protocol needs a strategy by which the key of the stream cipher can be quickly and effectively changed. One strategy for inexpensively rekeying stream ciphers concatenates an IV and the master key to form the cipher key. Changing the IV, which is public, also changes the cipher key, without the need for generating new secret material.

However, the need for resynchronization plays into an attacker's hands, particularly if the IVs are related in some way (which is often the case; for example, counters or consecutive frame numbers may be used). If the rekeying strategy relates the inputs to the internal state with insufficient non-linearity, the cipher may become prone to a related-key attack. In a related-key attack, an attacker can control, in different invocations of the cipher, differences in the internal state through selections of the master key or IV. This is analogous to differential cryptanalysis in block ciphers.

One notorious example of a related-key attack that affected practical implementations of a cryptographic protocol is due to Fluhrer, Mantin and Shamir [74]. They describe a related key attack on RC4 due to an invariance property of the Key Initialization Algorithm. The attack is possible because both the designer of RC4, and the authors of the 802.11 WEP protocol in which it is used, neglected to design robust key initialization routines.

The attack is performed as follows. Given the first x words of the RC4 key, and the initial output word, an attacker is able to compute the first x stages in the Key Initialization Algorithm, and with a probability of more than 5%, deduce a new target keyword. Provided with the first output word for the keystream of around sixty related keys (in which each differs only by an initial and known prefix), the attacker can calculate the entire initial state for any key. Furthermore, the complexity of the attack increases linearly rather than exponentially with the length of the RC4 key.

As a result of the Fluhrer et al. attack [74], Stubblefield, Ioannid and Rubin devised a practical related-key attack on the WEP protocol using around 2^{22} packets [217]. Many vendors of the WEP protocol constructed the RC4 key by concatenating a 24-bit counter and a secret key, incrementing the counter once for each RC4 key. The counter provided attackers with a readily available pool of strongly related keys. The attack was further facilitated by known-plaintext in the form of encapsulating headers, and poor key management strategies in which passphrases were used directly as secret keys.

The attack is mitigated by discarding the first 256 bytes of RC4 output or by hashing the master key with a strong hash function such as MD5 prior to creating the RC4 initial state [195]. To prevent this attack, Housely and Whiting produced a rekeying scheme specifically for RC4 that is less heavyweight than the MD5 hash function [99]. That these strategies were required is undesirable since they both increase the footprint (and complexity) and reduce the key agility and efficiency of the implementation.

4.2.9 Algebraic Attacks

A class of algebraic attacks known as the XL attacks caused a commotion in the stream cipher world when it was claimed they broke, with low complexity, popular stream ciphers, including the summation generator [147], LILI-128 [64] and Toyocrypt [51]. Because they are a new class of attacks, in which research progress is rapidly being made, they are discussed at length here.

Algebraic attacks upon stream ciphers are known-plaintext attacks in which the initial register state of n bits $(k_0, ..., k_{n-1})$ is recovered by solving a system of multivariate equations. For bit-based ciphers, the system of multivariate equations is derived as:

$$\begin{cases} b_0 &= f(k_0, \ldots, k_{n-1}) \\ b_1 &= f(L(k_0, \ldots, k_{n-1})) \\ b_2 &= f(L^2(k_0, \ldots, k_{n-1})) \\ & \vdots \end{cases}$$

where f is a highly-non linear filter function, L is the linear connection function, and b_i is the i^{th} consecutive output bit.

Generally one keystream bit yields one equation. Most attacks (including fast algebraic attacks, discussed in Section 4.2.9) use m consecutive keystream bits to solve the system of equations. However, some attacks [54] relax the restriction that the keystream bits must be consecutive.

The attacks are possible because:

- there exist efficient algorithms for solving systems of nonlinear multivariate equations of low degree [209].
- due to the linear nature of the connection polynomials in LFSR-based stream ciphers, a multivariate equation of low degree in certain state bits retains its low degree in the initial state bits.

Attack Methodology

Basic algebraic attacks consist of three steps.

Step 1 — Identify a System of Equations

This step is a pre-computation in which equations are identified that relate bits of initial state to bits of keystream. This stage needs only to be performed once for multiple keystreams produced by a single algorithm.

Step 2 — Substitute Keystream Into System of Equations

In this step, key bits observed from the keystream are substituted into the system of equations obtained in the previous step.

Generally, one multivariate equation is obtained for each observed key bit, and becomes part of a over-defined system of equations.

Step 3 — Solve System of Equations

In this step, the system of equations is solved to determine the initial cipher state and consequently the secret key K. The solution can be identified if there are a sufficient number of independent low-degree equations within the system.

The complexity of this step is exponential in the degree of the equations. The fast algebraic attacks discussed later in this document focus on reducing the degree of the equations to optimize this step.

Solving the System

There are a number of methods for generating over-defined systems of equations, in which the number of equations is sufficiently large compared to the number of variables. These methods include linearization [7] and the XL algorithm [51].

Linearization

The linearization technique applies to a system of non-linear equations with m equations, in which there are T different terms, such that $m \leq T$. By replacing every monomial with a new variable, the system is transformed to have T unknowns.

This algorithm is applicable when there are $\binom{n}{d}$ keystream bits available, where n is the number of variables within the equation and d is the degree of the equations. The system can be solved using Strassen's algorithm in about $7 \cdot T^{log_2 7}$ operations [7].

Using a variant – the relinearization algorithm of [126] – the system can be solved with about one-fifth that number of equations.

The XL algorithm

When there are fewer than $\binom{n}{d}$ bits of keystream, linearization is not applicable. However, the XL algorithm can succeed in finding a solution to the system at the cost of a higher computation complexity.

The XL algorithm works by multiplying all equations with all possible monomials of degree less than some D.

Given a system of m equations, $l_i(x_0, ..., x_{n-1}) = 0$ with $i = 1...m$, the XL algorithm generates a new system of equations $\prod_{j=1}^{k} x_{i_j} \cdot l_i$ such that $k \leq D - d$. Consequently the total degree of each of these equations is less than D. This step may allow the number of obtained equations to equal or exceed the number of variables. In this case the linearization step can be applied, in which each monomial is considered a new variable. The resulting system can be solved using Berlekamp's algorithm.

The XL algorithm does not guarantee that all new linear equations are independent. Consequently there has some been some debate on the validity of early algebraic attacks on stream and block ciphers (for example, [173]).

Applying Algebraic Attacks

Conventional algebraic attacks can be launched on filter functions f when

- the function f has a low algebraic degree; or
- the function f can be approximated with a probability close to 1 by a function h with a low algebraic degree

In [51], Courtois attacks the Japanese stream cipher Toyocrypt using a low-order approximation to its combining function f. Other than one monomial of degree 17, and another of 64, the highest terms in the filter function of Toyocrypt have degree 4. Thus Courtois' approximation succeeds because the higher-order terms almost always have the value zero. He claims that the operational complexity of this attack is 2^{92} operations.

Courtois and Meier [54] introduce faster attacks in which additional possibilities for successful key retrieval are considered:

- the function f has a multiple fg of low algebraic degree where g is some non-zero multivariate polynomial
- the function f has a multiple fg that can be approximated by a function of low algebraic degree with a probability of close to 1.

In this paper, Courtois and Meier improve upon the Toyocrypt attack of [53] by factoring the polynomial f to obtain degree 3 equations. They claim the attack succeeds with 2^{49} CPU clocks. They also attack LILI-128 by approximating the degree-6 boolean combiner with less complex equations of degree 4. They claim this allows an attack on LILI-128 with an operational complexity of 2^{96} operations. LILI-128 is an irregularly clocked cipher but they circumvent the clocking by guessing the internal state of the 39-bit clock-control register, increasing the computational complexity by a factor of 2^{39}.

Courtois and Meier generalize the result of the attack upon LILI-128 by showing that for any stream cipher with linear feedback, for a filter of k variables, it is possible to generate at least one equation with degree $\frac{k}{2}$. They warn against designing linear feedback stream ciphers in which the feedback uses only a small subset of state bits: these ciphers will be vulnerable to fast algebraic attacks, discussed in the next section, irrespective of their immunity to other conventional attacks.

In [52], Courtois describes breaking stream ciphers with stateful combining functions. He shows the complexity of such attacks decreases rapidly if the cipher outputs several bits during each clocking cycle. He demonstrates this new result on modified (but unused) versions of LILI-128, Bluetooth E0, and the word-based stream cipher SNOW.

Fast Algebraic Attacks

A new class of fast algebraic attacks was introduced by Courtois in [53] and improved upon by Armknecht in [8].

Fast algebraic attacks introduce a new step in to the methodology of the attack. This is:

Step 1a - Degree Reduction within the System of Equations

This step involves generating linear combinations of the equations within the system, which cancels out terms of high degree. Since the complexity of Step 3 is exponential in the highest degree within the system, Step 1a contributes substantially to a reduction in the time taken to perform Step 3.

The “fast” aspect of the attack comes from considering $\binom{n}{d}$ consecutive equations (and consequently consecutive keystream bits), and identifying a linear dependency that does not depend on output bits. The recursive structure within these equations allows the system to be solved using only a single dependency on $\binom{n}{e}$ successive windows of $\binom{n}{d}$ equations, rather than using $\binom{n}{e}$ dependencies.

Courtois [53] estimates the complexity of the fast algebraic attack to be $O(DE)$ where D is the size of the linear combination and E is the size of the system of equations.

However, in [96], Hawkes et al. claim that [53]'s complexity analysis is wrong. The error stems from the substitution of the keystream into the penultimate system of equations. Hawkes et al. argue that while the time complexity is $O(DE)$ when bitwise operations of the substitution are run in parallel, the true complexity is closer to $O(DE^2)$.

Further Improvements

In [96], Hawkes and Rose note that in previous papers, the substitution technique is inefficient because the equations reuse the same values of $g_t(z_t)$ when computing $g'_t(z'_t)$ and all use the same linear combination. Hawkes and Rose show how to reduce the complexity of this step to $2E \cdot D \cdot \log_2 D$ using a Discrete Fourier Transform. They also show how to reduce the complexity of step 1a (in the fast algebraic attack) from D^2 to $D(\log D)^2$.

In [8], Armknect improves the efficiency of the fast algebraic attack by introducing a new parallelizable pre-computation step that examines the structure of the filter and connection polynomial prior to deriving the system of equations. He presents empirical evidence to demonstrate the improvement of the attack: the conventional method takes 18 days to break a version of E0 with a key size of 25 bits, while with the additional pre-computational phase, the attack takes 14 hours. However, the improvement degrades as the key size increases.

4.3 A Note on the Biases in the RC4 Keystream

RC4 has been subjected to many attempts at cryptanalysis related to its key initialization algorithm. In the most significant of these, Mantin and Shamir [155] discovered a bias in the second byte of the RC4 keystream, which for separate, randomly keyed samples, has the value of zero twice as frequently as expected. Inspired by this result, Mironov [170] sought to determine whether other biases existed within the keystream, and the length of the prefix in which they occurred. The methodology that he used was to model both the Key Initialization Algorithm and Keystream Generation Algorithm as random shuffles. This involved idealizing the algorithms (described in Section 4.1.6), in which changes to j are assumed to be random. Consequently j is independent of index i and state S. Mironov justifies the idealization by claiming that Rivest's design principles conform to the randomization of j. Yet this is a flimsy justification; it is a cipher's specification that is analysed, not its design principles.

The idealization models the Key Initialization Algorithm as the exchange shuffle P_t, which is shown in Figure 4.3.

```
S = identity permutation

for i = 0 to t-1 do
    swap(S[i mod n], S[random(n)])
```

Figure 4.3: Idealized RC4 Key Initialization Algorithm — P_t shuffle [170]

Using this model, Mironov devises two distinguishers of the exchange shuffle from random, including a distinguisher that differentiates the permutation's sign — the parity of the transpositions that compose it — from random with 56% probability; and a position distinguisher, in which the likelihood of swapping the i^{th} and j^{th} elements is not uniform, but is dependent upon i, j and n. He uses these distinguishers to show that, following a single invocation of the Key Initialization Algorithm, the internal state of RC4 is not random.

Mironov claims that because the shuffling algorithm used in the Key Initialization Algorithm and Key Generation stages is imperfect, there is a strong bias in the first byte of output. This bias, he states, is unrelated to the second byte bias, and has a variation of around 0.6% from the mean value. Figure 4.4, from the Mironov paper, depicts this claim.

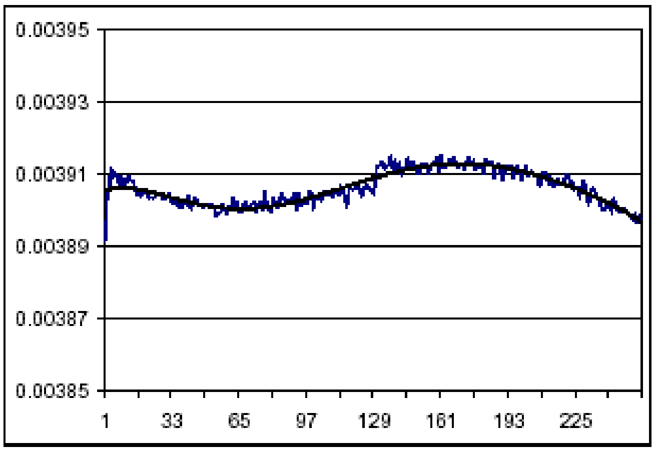

Figure 4.4: Claimed Bias in First Byte of RC4 output [170]

This graph appears to be Mironov's main support in the claim of a bias in the first byte of RC4, yet it does not illustrate any application of a uniform distribution, is not clearly explained, and is not supported by any concrete data or statistical results.

Mironov furthers the claim by stating that this bias persists throughout all of the bytes up to (and including) the 768^{th} byte, although by this stage, he says it is very difficult to detect. From this observation he constructs a theoretical distinguisher against non-idealized RC4 with the first 1,700 bytes of output. To prevent against this potential attack, he recommends discarding the first 3,072 bytes of keystream output.

Our analysis

In order to examine the possible bias in the output bytes of RC4, we carried out a statistical analysis, using the frequency test, on the first ten byte positions in the output stream of $100,000$ different RC4 keystreams. The only weakness that was identified was a strong bias in the second byte of the keystream [155].

If no byte bias exists, then we would expect an equal number of each of the 2^8 byte patterns to appear. As we analysed $100,000$ keystreams, then for unbiased output, the expected count e_i for each byte pattern is $\frac{100{,}000}{256} = 390.625$. A χ^2 test of goodness-of-fit to a uniform distribution was applied, using the null hypothesis that there are no biases in the keystream patterns.

The χ^2 statistic is calculated as

$$\chi^2 = \sum_{i=0}^{255} \frac{(o_i - e_i)^2}{e_i}, \tag{4.1}$$

using 255 degrees of freedom, where o_i is the observed frequency of ones in the keystream. Using standard tables, χ^2 values can be converted to *p-values.* A p-value beneath a chosen significance level indicates that the null hypothesis has failed, and that there are biases in the keystream. We chose a significance level of 0.001: a p-value less than this indicates a bias in the corresponding byte pattern. The equivalent χ^2 statistic for this threshold is 330.535: if the statistic exceeds this value, the null hypothesis has failed.

Table 4.2 summarizes the statistic (χ^2 with 255 degrees of freedom) and p-value for the first ten output bytes. These results strongly support a bias in the second byte position, but not in any other position.

Position	χ^2 **Statistic**	**p-value**
1	217	0.9589
2	644	0.0000
3	216	0.9628
4	304	0.0179
5	288	0.0780
6	300	0.0283
7	287	0.0834
8	274	0.1997
9	235	0.8075
10	251	0.5615

Table 4.2: Uniformity Test on Byte Positions in RC4 Keystream

As mentioned in Section 4.2.1, Mihalejevic identified a bias in the RC4 keystream given a three-byte prefix in the master key of $[0, 0, 255]$. To detect this bias in the above analysis, the sample of keystreams generated would need to be increased significantly. This $[0, 0, 255]$ pattern in the first three bytes would be expected to occur once in every $16,777,216$ ($= 2^{24}$) randomly generated keystreams and is unlikely to account for the second byte bias detected in the sampling methods applied here, or also for any first byte bias claimed by Mironov [170].

We were unable to find any evidence for Mironov's claim of a bias in the first keystream byte of RC4, so believe that his advice to discard 3,072 bytes of keystream output is unnecessarily conservative. Instead we recommend following the advice of [195], which is intended to avoid the WEP attack of [217]. This advice includes discarding the first 256 output bytes of the RC4 generator prior to commencing encryption.

4.4 Summary

In this chapter, we reviewed nine modern word-based stream ciphers. All of these ciphers, with the exception of RC4, were influenced by block ciphers in some way. In particular, Helix and Scream were notable for their block-function-like update functions, the latter incorporating a modified AES round.

Many of the ciphers reused the update function in their rekeying strategies, enabling a reduction in the cipher footprint. Those that did not were RC4 and Turing, although RC4's rekeying strategy bears a strong resemblance to its update function. Almost all of the ciphers incorporated an initialization vector into the rekeying strategy. The exceptions were Hiji-Bij-Bij, RC4 and Rabbit, which require careful key management strategies.

Most of the word-based stream ciphers reviewed in this section outperform block ciphers on the Intel Pentium family. An exception is MUGI, which is targeted towards 64-bit architectures, so performs poorly on 32-bit architectures such as the Intel Pentium family. Also Hiji-Bij-Bij has a complex update function so suffers from poor throughput.

Of the attacks presented in this chapter, the most successful on synchronous word-based stream ciphers are guess and determine attacks, time-memory-data trade-off attacks, and distinguishing attacks. The first and second of these can be prevented by designing the stream cipher with a state size that is significantly larger than the key, a strategy that also appears to defeat divide and conquer attacks. But as shown in Section 4.1 and Chapter 6, this affects key agility, and a compromise between security and efficiency is required.

RC4 is vulnerable to a distinguishing attack, due to a bias that occurs in the second byte of its keystream [155]. Following this discovery, Mironov [170] sought to determine whether other biases existed within the keystream, and the length of the prefix in which they occurred. He claimed that additional biases exists in the keystream, including a prominent bias in the first keystream byte. The attendant advice is to discard 3,072 bytes of keystream following the initialization.

Using statistical methods, in which we analyzed 100,000 independently-keyed 10-byte keystreams, we were unable to find any evidence for Mironov's claim of a bias in the first RC4 keystream byte. We endorse the advice of Mantin and Shamir, and subsequently Rivest, in discarding the first 256 output bytes of the RC4 generator. This has less of a performance penalty upon the cipher than does the advice of Mironov.

Chapter 5

Cryptanalysis of the Alpha1 Stream Cipher

Alpha1 [140] was proposed in 2001 as a stream cipher algorithm for mobile and wireless devices. It was designed to remedy the weaknesses of A5, a GSM cipher algorithm that came in two flavours, a strong version A5/1, a weaker export-friendly version A5/2. Although the A5 algorithm was never released publicly, the details were allegedly leaked in 1994; following this, it was broken by Golic in [81]. In 1999, the algorithm was verified by Briceno et al. [37] who performed a physical reverse-engineering exercise on a cellphone that implemented A5/1. Biryukov and Shamir [30] superceded Golic's cryptanalysis; together with Briceno's work, they reinforced the lesson that secrecy does not equal security.

Both A5 and Alpha1 are based on linear feedback shift registers (LFSRs) and use irregular clocking driven by feedback data. A5 uses three irregularly clocked LFSRs, with a total internal state and key size of 64 bits. Alpha1 attempted to remedy the small internal state size of A5 through the use of an additional LFSR, and a total key and state size of 128 bits. One of Alpha1's four registers is regularly clocked; the other three registers are irregularly clocked. The contents of two stages from each of these irregularly clocked registers are used to control the clocking according to a majority function.

Several weaknesses and inaccuracies in the Alpha1 specification are noted in [171], which also outlines a theoretical attack on $R1$, the shortest register of Alpha1. Wu [225] uses a similar approach to recover the initial state of this register, but leaves recovery of the remaining three registers (totalling 99 bits) as an open problem. This chapter solves that problem, by presenting a divide and conquer attack on Alpha1 that begins with a slight improvement of Wu's attack, and continues the divide and conquer approach to recover the initial states of the remaining three registers.

This chapter is organized as follows: Section 5.1 contains a description of Alpha1. Weaknesses and previous attacks are outlined in Section 5.2. The new divide and conquer attack is outlined in Section 5.3. Finally, some concluding remarks on the security provided by Alpha1 are given in Section 5.4.

5.1 Description of Alpha1

Alpha1 uses four binary LFSRs of lengths 29, 31, 33, and 35, respectively. These registers are denoted $R1$, $R2$, $R3$, and $R4$, as shown in Figure 5.1. For clarity, clocking information is omitted.

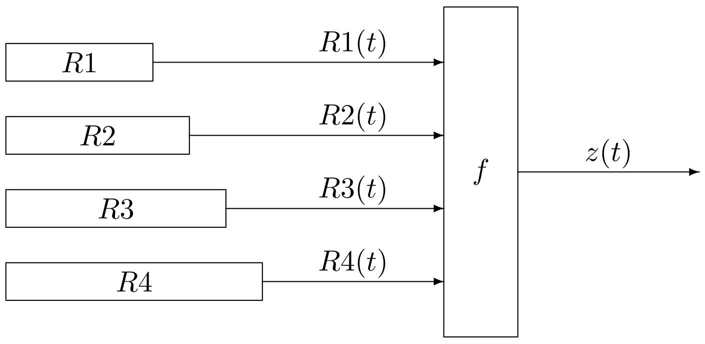

Figure 5.1: Alpha1 Stream Cipher

The LFSRs have the following feedback polynomials:

$$\begin{aligned}
f_1(x) &= x^{29} \oplus x^{27} \oplus x^{24} \oplus x^{8} \oplus 1 \\
f_2(x) &= x^{31} \oplus x^{28} \oplus x^{23} \oplus x^{18} \oplus 1 \\
f_3(x) &= x^{33} \oplus x^{28} \oplus x^{24} \oplus x^{4} \oplus 1 \\
f_4(x) &= x^{35} \oplus x^{30} \oplus x^{22} \oplus x^{11} \oplus x^{6} \oplus 1
\end{aligned}$$

At time t, denote the output of Ri as $Ri(t)$ and the output keystream of Alpha as $z(t)$. The keystream bit is a function of the output bit of each of the four registers.

$$\begin{aligned} z(t) &= f(R1(t), R2(t), R3(t), R4(t)) \\ &= R1(t) \oplus R2(t) \oplus R3(t) \oplus R4(t) \oplus (R2(t) \text{ AND } R3(t)) \\ &= R1(t) \oplus R4(t) \oplus (R2(t) \text{ OR } R3(t)) \end{aligned}$$

Let Ri_j denote the jth stage of LFSR i. $R1$ is regularly clocked and the other three LFSRs are irregularly clocked in a stop/go fashion (the output bits of the LFSRs occur one or more times in the LFSR output sequence). Each of $R2$, $R3$ and $R4$ has two clocking taps. These six clocking taps are divided into two groups, each containing one tap from each of the three irregularly clocked LFSRs:

$$\begin{aligned} \text{Group 1}: &\quad R2_{10}, R3_{22}, R4_{11} \\ \text{Group 2}: &\quad R2_{21}, R3_{10}, R4_{24} \end{aligned}$$

For each group, the majority bit is calculated. For group 1, the majority bit is calculated as:

$$\text{Maj}_1(R2, R3, R4) = (R2_{10} + R3_{22} + R4_{11}) \gg 1$$

where $\gg$ denotes the right shift operation.

For group 2, the majority bit is calculated as:

$$\text{Maj}_2(R2, R3, R4) = (R2_{21} + R3_{10} + R4_{24}) \gg 1$$

Register Ri ($i = 2, 3, 4$) is clocked when both of its clocking taps agree with the majority bit of the respective groups. For example, $R2$ is clocked only if $R2_{10}$ agrees with the majority bit of group 1 and $R2_{21}$ agrees with the majority bit of group 2. For each keystream bit produced, between two and four of the Alpha1 registers are clocked, including $R1$, which is always clocked.

5.2 Known Weaknesses and Previous Analysis

In [171], Mitchell notes two weaknesses and an inaccuracy in the specification of Alpha1. Firstly, the feedback polynomial $f_4(x)$ is not irreducible. Secondly, the combining function f can be approximated by a linear function. Also, as he points out, the authors of Alpha1 miscalculated the clocking probability.

The feedback polynomial $f_4(x)$ is not irreducible, which can be seen by its even number of terms. This polynomial can be factored into four smaller degree polynomials, one of degree 22, one of degree 11, and two of degree 1. As $f_4(x)$ is not primitive, the sequence generated by $f_4(x)$ is not of maximal length. For example, if $R4$ is initialized as $11\dots11$, then the $R4$ output sequence will also be $11\dots11$. This implies Alpha1 has at least 2^{93} weak keys, because in this case, the contribution from $R4$ to both the clocking and the combining function is constant, and the Alpha1 output keystream depends only on $R1$, $R2$ and $R3$.

The authors of Alpha1 claimed that $R2$, $R3$ and $R4$ will each be clocked with a probability of $\frac{7}{13}$. In [171] it is noted that if the inputs to the six clocking taps are randomly distributed, then $R2$, $R3$ and $R4$ will each be clocked with a probability of $\frac{9}{16}$.

The combining function $f(R1(t), R2(t), R3(t), R4(t))$ is closely approximated by a linear combination of the output from registers $R1$ and $R4$. The output function

$$z(t) = R1(t) \oplus R4(t) \oplus 1$$

holds with probability equal to 0.75. Together the correct clocking probability and the combining function bias can be exploited in a known-plaintext attack outlined in [171] to recover the initial state of $R1$. The attack involves exhaustively searching through all 2^{29} initial states of $R1$, to produce a length of candidate keystream from which a correlation measure is tested. Note that this attack is not implemented, therefore the complexity of testing each guess is estimated. Let k denote the complexity of testing each $R1$ candidate state, then the complexity of this attack on $R1$ is $2^{29} \cdot k$. Once the initial state of $R1$ is recovered, this leaves the initial states of the other three registers (totalling 99 bits) to be obtained through exhaustive search. Thus applying the attack to recover the initial state of all four registers has the complexity $2^{29} \cdot k + 2^{99}$.

Wu [225] shows that the clocking taps are biased, and uses an attack similar to that described in [171], to recover the initial state of $R1$. The keystream can be rewritten as the regularly clocked $R1$ output sequence plus the sub-keystream of the remaining registers:

$$z(t) = R1(t) \oplus \mathbf{z}(t)$$

By guessing the initial state of $R1$, the corresponding $\mathbf{z}(t)$ can be produced. Using the biased clocking, the distributions of digraphs 00, 01, 10, 11 in $\mathbf{z}(t)$ are calculated to be $\frac{19}{64}$, $\frac{13}{64}$, $\frac{13}{64}$, and $\frac{19}{64}$, respectively. If the distribution of digraphs in $\mathbf{z}(t)$ agrees with the calculated values above, the guessed initial state of $R1$ is deemed correct. It is claimed that this attack has complexity 2^{29} and requires about 3,000 bits of known keystream for a success rate of about 90%. Note that the estimated complexity of 2^{29} ignores the work required to test each candidate. As the algorithm iterates through 3,000 bits of keystream for each of the 2^{29} possible $R1$ initial states, we estimate a complexity of $2^{29} \cdot 3,000 \approx 2^{40.6}$ operations. We express the complexity in terms of operations to permit a comparison with our attack presented in Section 5.3.

5.3 Divide and Conquer Attack on Alpha1

In this section, we outline our divide and conquer attack against Alpha1 which sequentially recovers the initial states of all four registers. Firstly, the initial state of $R1$ is recovered using a slightly improved version of the attack presented in [225]. Secondly, we recover the initial state of $R4$ using a probabilistic correlation attack. Thirdly, we exhaustively search through the initial states of $R2$, and reconstruct the initial state of $R3$.

5.3.1 Recovery of $R1$

In [225], the initial state of $R1$ is recovered by measuring the distribution of digraphs in $\mathbf{z}(t)$ for each candidate $R1$ initial state, and comparing that with the expected distribution for the correct initial state. Note that the digraphs can be divided into two groups, one with identical bit values (00 and 11) and the other with different bit values (01 and 10). For the correct initial state of $R1$, the distribution is symmetric: both 00 and 11 occur with the same probability ($\frac{19}{64}$) and 01 and 10 both occur with probability $\frac{13}{64}$. We can combine these probabilities and detect the correct initial state of $R1$ by calculating the binary derivative $d'(t)$ of $\mathbf{z}(t)$:

$$d'(t) = \mathbf{z}(t-1) \oplus \mathbf{z}(t) \text{ for } t \geq 1$$

This is a similar approach to that taken by Mitchell in [171]. The binary derivative measures the number of bits in a binary sequence that differ from the previous bit value. The digraphs with identical bit values have a binary derivative of 0; the remaining digraphs have a binary derivative of 1. Based on the probabilities calculated in [225], the bias in the binary derivative is derived for the correct $R1$ initial state. The distribution of zeros and ones in the binary derivative for the correct state is $\frac{38}{64}$ and $\frac{26}{64}$, respectively. For an incorrectly guessed initial state, there is no discernable bias.

Having observed keystream $\{z(t)\}_{t=0}^{n}$ of Alpha1, we use the following algorithm to recover the initial state of Alpha1:

Output: initial register of $R1$

1. Guess $\hat{R1}$, a candidate initial state for $R1$, and produce the $\hat{R1}$ output sequence $\{\hat{R1(t)}\}_{t=0}^{n}$
2. Calculate $\{\mathbf{z}(t)\}_{t=0}^{n}$ by exclusive-oring $\{\hat{R1(t)}\}_{t=0}^{n}$ to $\{z(t)\}_{t=0}^{n}$
3. Calculate the binary derivative $\{d'(t)\}_{t=1}^{n}$ of $\{\mathbf{z}(t)\}_{t=0}^{n}$
4. Calculate the sample distribution for $\{d'(t)\}_{t=1}^{n}$. If the proportion is less than $\frac{26}{64}+c$ (where c is a one-sided threshold based upon keystream length), then put the guessed $R1$ state into the list of candidate initial states.

This attack requires exhaustive search over all 2^{29} states of $R1$ and the production of n bits of output keystream for each state. The attack therefore has an operational complexity of $2^{29} \cdot n$ and text requirements of n bits.

Experimental Results

To efficiently recover $R1$, optimal values for the required keystream length n and the threshold value c have to be determined. If the threshold value is too small, then the correct candidate may be inadvertently discarded. If it is too high, then a large number of false positives may result.

We experimented with five different threshold values $\frac{2}{128} \leq e \leq \frac{6}{128}$ (in steps of $\frac{1}{128}$) and six keystream lengths $512 \leq n \leq 5,120$. The attack was run 2^{16} times for each threshold value and keystream length. The results of the experiment are shown in Table 5.1. The proportion of cases for which the true initial state was successfully identified and the average number of false positives are recorded in columns 3 and 4 respectively. For example, for $c = \frac{3}{128}$ and $n = 2,048$ respectively, the attack successfully identified the correct initial case in 96.8% of cases with around eight false positives. Increasing the length of keystream to

$n = 3,072$ found correct initial states in 98.8% of cases, with no false positives being identified. Generalizing this, an increase in the keystream length allows a corresponding decrease in the threshold without generating extra false positives. However, the keystream length is linearly related to the time that the attack requires to determine the correct initial state, so a trade-off between accuracy and efficiency exists.

threshold c	keystream length n	success rate	false positives
$\frac{2}{128}$	512	0.750	2^{17}
	768	0.788	2^{12}
	1,024	0.816	$2^{8.0}$
	2,048	0.895	0
	3,072	0.935	0
	4,096	0.960	0
$\frac{3}{128}$	512	0.840	2^{19}
	768	0.878	2^{15}
	1,024	0.910	2^{11}
	2,048	0.968	$2^{3.0}$
	3,072	0.988	0
	4,096	0.995	0
$\frac{4}{128}$	512	0.900	2^{20}
	768	0.938	2^{17}
	1,024	0.961	2^{14}
	2,048	0.993	$2^{4.0}$
	3,072	0.999	0
	4,096	1.000	0
$\frac{5}{128}$	512	0.944	2^{22}
	768	0.973	2^{19}
	1,024	0.986	2^{17}
	2,048	0.999	$2^{8.1}$
	3,072	1.000	1
	4,096	1.000	0
$\frac{6}{128}$	512	0.971	2^{23}
	768	0.989	2^{21}
	1,024	0.996	2^{20}
	2,048	1.000	2^{13}
	3,072	1.000	$2^{5.8}$
	4,096	1.000	$2^{4.5}$

Table 5.1: Table of Success Rates in Recovering Register 1 of Alpha1

For the parameters $c = \frac{3}{128}$ and $n = 2,048$, it takes around thirty minutes to run the attack on a Pentium 4 3.2 GHz desktop with a success rate of 96.8%.

The complexity of the attack is $2^{29} \cdot n = 2^{40}$ for $n = 2,048$. The memory required is equal to $3n$. This is a marked improvement over the attack presented in [225], which took several hours on a Pentium 4, with a success rate of 90% for $n = 3,000$. Note that the attack described in [225] calculates the distribution across four digraphs for every initial $R1$ state. Our attack on $R1$ only needs to calculate a distribution with two states, and is therefore more efficient than that of [225].

The running time of our attack can be further reduced by running a two stage process. In the first stage of the process, use a small length of keystream with a higher threshold to remove most of the incorrect candidates, while maintaining a larger than desired number of false positives. This stage can be quickly completed due to the short length of the keystream. The second stage is a more precise analysis of the much reduced pool of candidates. The keystream length is increased, so that false positives from the first stage can be quickly identified; however, the pool is much smaller so the time taken is smaller than that for the unoptimized attack. Using parameters of $n = 1,024$ and $c = \frac{56}{128}$ for the first stage, and $n = 2,048$ and $c = \frac{55}{128}$ reduces the time required to recover $R1$ to about 20 minutes, with a complexity of just over 2^{39} and a success rate of 93%. This is almost a ten-fold improvement over the attack of [225], but uses less keystream and has a greater success rate.

5.3.2 Reduced Version of Alpha1

As the initial state of $R1$ has already been recovered in Section 5.3.1, in the remainder of this section we consider a *reduced* version of Alpha1 with $R1$ removed, as shown in Figure 5.2.

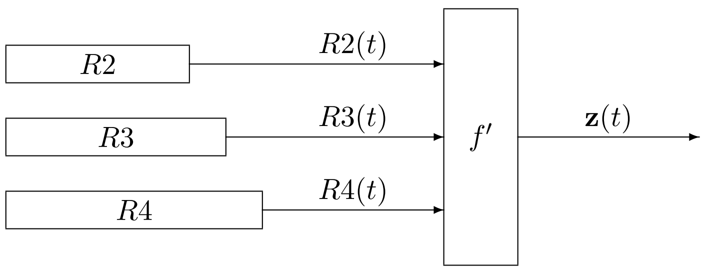

Figure 5.2: Reduced Version of Alpha1 Stream Cipher ($R1$ Removed)

Denote the keystream for reduced Alpha1 as $\mathbf{z}(t)$. As noted in Section 5.2, this can be obtained from the keystream of Alpha1 by exclusive-oring the regularly clocked $R1$ output sequence $\{R1(t)\}_{t=0}^{n}$ to the observed keystream $\{z(t)\}_{t=0}^{n}$. Let f' define the combining function for the reduced Alpha1. The output keystream at time t can be written as

$$\begin{aligned}\mathbf{z}(t) &= f'(R2(t), R3(t), R4(t)) \\ &= R4(t) \oplus (R2(t) \text{ OR } R3(t))\end{aligned}$$

The clocking mechanism for registers $R2$, $R3$ and $R4$ forming the reduced version of Alpha1 is identical to that of the original Alpha1, described in Section 5.1.

$R2(t)$	$R3(t)$	$R4(t)$	$\mathbf{z}(t)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Table 5.2: Truth Table for Update Function f' in Reduced Version of Alpha1

From Table 5.2, note that $\mathbf{z}(t)$, the output of the reduced version of Alpha1, is strongly correlated to the input value $R4(t)$. $R4(t)$ disagrees with $\mathbf{z}(t)$ in six cases out of eight, so $P(\mathbf{z}(t) \neq R4(t)) = 0.75$. This bias enables a correlation attack targeting $R4$, provided an appropriate measure of correlation can be found. The correlation measure used in our attack is the joint probability, and is described in the next section.

5.3.3 Recovery of $R4$

The classical correlation attack described in [213], using correlation measures based upon Hamming distance, is useful when the underlying sequences are of identical length. Irregular clocking of LFSRs, as occurs for Alpha1, produces a keystream sequence that has a different length from the underlying LFSR sequence. This provides resistance to the classical correlation attack. However, a correlation attack can still be performed, but with a different correlation measure.

Model for the Probabilistic Attack

For the correlation attack on $R4$, view the keystream of the reduced Alpha1, $\mathbf{z}(t)$, as a version of the underlying $R4$ sequence, to which bit insertion and additive noise have been applied, as shown in Figure 5.3. Bit insertion allows for the increased length of the keystream segment compared to the regularly clocked $R4$ sequence (due to the stop/go clocking) and the additive noise allows for the contribution from $R2$ and $R3$. The task of the cryptanalyst is to determine the correct initial state of $R4$ given a segment of keystream $\{z(t)'\}_{t=1}^{n}$. A possible correlation measure is the joint probability proposed in [84]. This is based upon Levenshtein distances, and has been successfully used in attacking another irregularly clocked keystream generator, the shrinking generator [50]. The Levenshtein distance is the minimum number of edit operations needed to transform one sequence to another. Valid operations include bit insertion, bit deletion and bit complementation. To attack $R4$ of the reduced Alpha1 where stop/go clocking has been used, bit insertion and bit complementation are required.

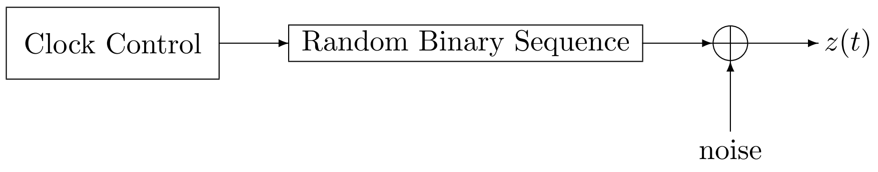

Figure 5.3: Model for Keystream Generation

Joint Probability

The joint probability $P(A^m, B^n)$ for arbitrary binary input and output strings $A^m = \{a(t)\}_{t=1}^{m}$ and $B^n = \{b(t)\}_{t=1}^{n}$, respectively, is computed using a recursive algorithm [84] based on string prefixes. Let $A^s = \{a(t)\}_{t=1}^{s}$ denote the prefix of A^m of length s and $B^{e+s} = \{b(t)\}_{t=1}^{e+s}$ denote the prefix of B^n of length $e + s$. Let $P(s, e)$ denote the partial joint probability for A^s and B^{e+s}, for $1 \leq s \leq m$ and $0 \leq e \leq n - m$. Let $\delta(a, b)$ denote the complementation probability, with $\delta = (1 -$ noise probability$)$ if $a = b$ and $\delta =$ noise probability otherwise. Let p denote the bit insertion probability (1 − clocking probability). The partial probability satisfies the recursion

$$P(s,e) = P(s,e-1)p + P(s-1,e)(1-p)\delta(a(s), b(e+s))$$

%	Random	Correlated	Random	Correlated	Random	Correlated
	n = 100		n = 200		n = 300	
0	0	1.24E-26	0	4.81E-54	0	2.07E-82
10	0	3.45E-23	0	2.96E-48	0	1.23E-74
20	0	1.66E-22	0	2.15E-47	0	1.21E-73
30	0	4.81E-22	0	8.59E-47	0	6.91E-73
40	2.94E-29	1.19E-21	0	2.90E-46	0	3.23E-72
50	8.46E-28	2.72E-21	0	9.53E-46	0	1.24E-71
60	7.94E-27	6.46E-21	0	3.06E-45	0	4.70E-71
70	5.21E-26	1.63E-20	0	9.84E-45	0	2.00E-70
80	3.13E-25	4.56E-20	0	4.38E-44	0	1.12E-69
90	2.50E-24	1.99E-19	4.29E-64	3.15E-43	0	1.15E-68
95	9.51E-24	6.63E-19	1.21E-57	1.74E-42	0	8.86E-68
100	6.73E-21	1.32E-15	1.00E-50	3.55E-38	1.32E-84	5.14E-63

Table 5.3: Joint Probabilities for Random and Correlated Strings with p=$\frac{7}{16}$

for $1 \le s \le m$ and $0 \le e \le n-m$, with initial values $P(1,e) = p^e, 0 \le e \le n-m$, and $P(s,-1) = 0, 2 \le s \le m$.

Experiments were conducted to ensure the suitability of joint probability as an effective correlation measure for reduced Alpha1. These compared the joint probability values for random strings of length n and m, and the values for correlated strings of length n and m, where m was set to $n \times p - 2 \times \sqrt{n}$. The experiments were conducted for the three string lengths $n = 100, 200,$ and 300 bits respectively, with a bit insertion probability p of $\frac{7}{16}$ and noise probability of 0.75.

For each experiment, 10,000 trials were conducted. The results are given in Table 5.3. The left column shows the percentage of random or correlated strings that achieve a given joint probability, which is shown in a column to the right, according to the length of n. For strings of lengths $n = 100$, the minimum joint probability between correlated strings was 1.24E-26. More than 60% of tested random strings with length 100 failed to achieve this level of correlation, as can be seen by their lower joint probability of 7.94E-27. For $n = 200$, the joint probability of the correlated strings was higher than that for 95% of random strings. For n = 300, the joint probability clearly distinguished all correlated strings from all random strings. The results indicated that joint probability is an effective correlation measure.

Attacking $R4$

Let $\{R4(t)\}_{t=0}^{n}$ be the binary sequence under some unknown clock control. Assume the clocking probability of $R4$ is $\frac{15}{28}$, based on the results presented in [225]. That is, $R4(t)$ has $\frac{13}{28}$ probability of being repeated. Assume the distributions of $R2(t)$ and $R3(t)$ are uniformly random, then

$$R2(t) \text{ OR } R3(t) = 1$$

with probability 0.75. In a correlation attack on $R4$, the input from $R2$ and $R3$ can be considered noise.

The correlation attack on $R4$ requires exhaustive search through all possible initial states of $R4$. For each initial state, a segment of the LFSR sequence of length m is produced under regular clocking. Given the known keystream $\mathbf{z}(t)$, a joint probability value can then be calculated for every state. The joint probability for the actual $R4$ initial state should be greater than the joint probability for the vast majority of the incorrect states. Also, from Table 5.3, we expect the difference between the joint probability value for the correct state and the joint probabilities for the incorrect states to increase if the length of known keystream increases. The attack requires exhaustive search over the 2^{35} initial states of $R4$; for each of the 2^{35} candidate $\hat{R}4$ initial states:

Input: reduced Alpha1 keystream $\{\mathbf{z}(t)\}_{t=0}^{n}$

Output: the joint probability value of $R4$

1. Generate $\{\hat{R}4(t)\}_{t=0}^{n}$, the regularly clocked $\hat{R}4$ output sequence
2. Calculate the joint probability of $\{\hat{R}4(t)\}_{t=0}^{n}$ and $\{\mathbf{z}(t)\}_{t=0}^{n}$, using the iterative procedure discussed previously in this section.

The joint probability values are stored and indexed by the candidate initial states. This array is then sorted on the joint probability value. One of the candidate $\hat{R}4$ initial states with the highest joint probability is likely to be the correct initial state.

The joint probability algorithm compares the two input strings, the random binary string and the keystream. This attack has complexity $2^{35} \cdot n^2$. There is a trade-off between the length of the known keystream n and the accuracy of the attack.

In one computer simulation, we calculated the joint probability value for the correct $R4$ initial state and 100,000 other random $R4$ initial states using 7,000 bits of keystream. We repeated the experiment 100 times to observe that the joint probability values of the correct $R4$ initial states were always in the top 1% of the values. Using this attack to search through all 2^{35} initial states of $R4$ gives $2^{28.4}$ candidate states. Using a further 7,000 bits of keystream to search through the pool of candidates reduced it to $2^{21.8}$ members. A total of five iterations is required to reduce the pool to a handful of candidates. For each iteration, the pool of candidates is reduced by a factor of 99. The text requirement of this attack is 35,000 bits, and the complexity

$$2^{35} \cdot 7,000^2 + 2^{28.4} \cdot 7,000^2 + \ldots + 2^{8.6} \cdot 7,000^2 \approx 2^{61}$$

The memory requirement is roughly equal to n^2. For $n = 7,000$, $2^{29.8}$ bits of memory are required by this stage of the attack.

5.3.4 Recovery of $R2$ and $R3$

Following the recovery of the correct initial states of $R1$ and $R4$, the next target in the divide and conquer attack is $R2$, due to its shorter length comparative to $R3$. The attack proceeds by exhaustively searching through all possible initial states of $R2$ and reconstructing $R3$ for each of the $R2$ states. If the wrong $R2$ state is used in the reconstruction of $R3$, the reconstruction eventually fails. For the correct $R2$ state, successful reconstruction of $R3$ is possible. The following algorithm is used:

Inputs: reduced Alpha1 keystream $\{\mathbf{z}(t)\}_{t=0}^{n}$, initial register state $R4$

Outputs: initial register states $R2$ and $R3$

1. Guess $\hat{R2}$, a candidate initial state for $R2$. Set $t = 0$
2. Calculate $\hat{R3}(t)$:
   - if $\mathbf{z}(t) \oplus R4(t) = 0$ and $\hat{R2}(t) = 0$ then $\hat{R3}(t) = 0$
   - if $\mathbf{z}(t) \oplus R4(t) = 0$ and $\hat{R2}(t) = 1$ then $\hat{R2}$ is wrong; go to 1
   - if $\mathbf{z}(t) \oplus R4(t) = 1$ and $\hat{R2}(t) = 0$ then $\hat{R3}(t) = 1$
   - if $\mathbf{z}t \oplus R4(t) = 1$ and $\hat{R2}t = 1$ then guess $\hat{R3}(t)$.
3. Guess the value of two clocking taps, $\hat{R3}_{10}(t)$ and $\hat{R3}_{22}(t)$

4. Clock $\hat{R}2$, $\hat{R}3$ and $R4$ according to majority function
   - if $\hat{R}3$ did not clock but $\hat{R}3(t) \neq \hat{R}3(t-1)$ then $\hat{R}3$ is wrong; decrement t and backtrack to 2
5. If $\hat{R}3$ has clocked fewer than eleven times, increment t and go to 2
6. Reset $R4$ to initial state. Set $\hat{R}2$ and $\hat{R}3$ to guessed states
   - Clock cipher to produce $\hat{\mathbf{z}}(t)$. If $\hat{\mathbf{z}}(t) \neq \mathbf{z}(t)$, $\hat{R}2$ is wrong; go to 1
   - Iterate clocking of the previous step until $t = n$
   - Output $\hat{R}2$ and $\hat{R}3$

In the algorithm above, all possible clocking taps are tested before changing the guess for the $\hat{R}3(t)$ value. If all the guesses are exhausted, that is, if the reconstruction of $R3$ fails, then the next $\hat{R}2$ state is tested. This algorithm will give a candidate $\hat{R}3$ initial state for the guessed $\hat{R}2$ state after $R3$ clocks eleven times (since there are eleven stages between the top of the R3 register and the first clocking tap). This result can be checked by running the Alpha1 generator until $R3$ clocks another twenty-two times (to cycle through $R3$ completely). If the output sequence agrees with the observed $\mathbf{z}(t)$, then the guessed $R2$ state and the reconstructed $R3$ state are the correct initial states. The clocking probability of $R3$ is $\frac{15}{28}$, so thirty-three clocks of $R3$ produce sixty-two bits of output keystream on average, which is therefore the text requirement of the attack.

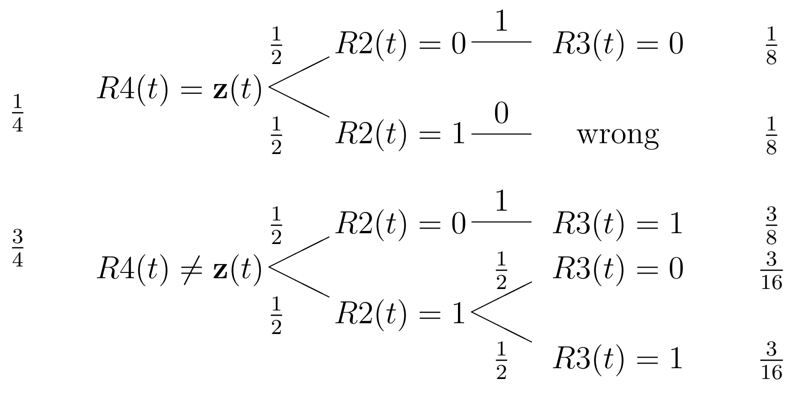

Figure 5.4: Probability Tree of Guessing Register 3 in Alpha1

Figure 5.4 shows the probability tree of Step 2 of the algorithm. Making a wrong guess for the value of $R3(t)$ or the initial state $R2$ can be detected with

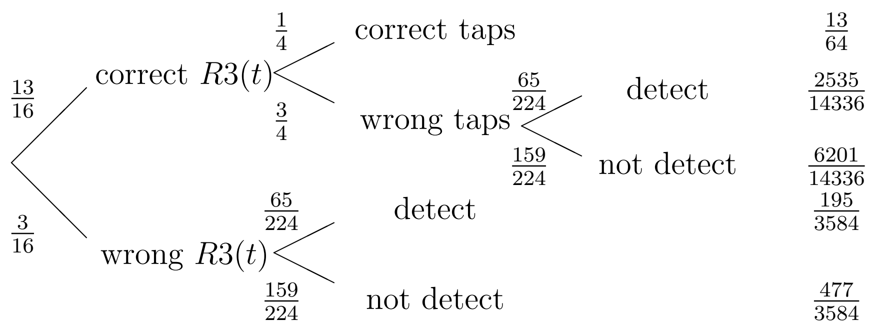

Figure 5.5: Probability Tree of Guessing Register 3 Clocking Taps in Alpha1

probability $\frac{1}{8}$. Assuming the guess for $R2$ is correct, the value of $R3(t)$ can be calculated with probability $\frac{4}{8}$ or guessed with probability $\frac{3}{8}$. The probability that a register does not clock in a given cycle is $\frac{13}{28}$. Wrong guesses of the clocking taps, $R3(t)$ and $R2$ could be detected with probability $\frac{5}{8} \cdot \frac{13}{28} = \frac{65}{224}$ (including the $\frac{1}{8}$ mentioned above).

Figure 5.5 shows the probability for guessing the correct $R3$, clocking taps and also the probability of detecting wrong guesses. The probability of wrong guesses not being detected is $\frac{8109}{14336} = 0.5656$ at each branch, meaning that when eight levels of the search tree are traversed, the probability of not detecting wrong branches diminishes to 0.01. The probability of following a correct branch is therefore 0.99. There are seven wrong branches at each step, giving a search space of $7^8 \approx 2^{22.5}$ in the worst case scenario. Therefore, to break $R2$ and $R3$ using the method described requires $2^{31} \cdot 2^{22.5} = 2^{53.5}$ operations. This is less effort than exhaustive search of the $R2$ and $R3$ initial states (2^{64}), and also less than the complexity of recovering $R4$ (2^{59}). Negligible memory is required in this stage of the attack.

5.4 Summary

Alpha1 is designed to have an effective key space of 128 bits. However, in this chapter we have presented a divide and conquer attack that reduces the security level to approximately sixty bits. This attack recovers the initial states of Alpha1's four registers, unlike previous partial attacks which dealt only with the shortest and regularly clocked register.

The attack described in this chapter begins with an improved recovery of $R1$, followed by a new probabilistic correlation attack on $R4$, and finally a guess and check attack to recover the initial states of $R2$ and $R3$. The attack on $R1$ has complexity 2^{41}, uses 2^{13} bits of memory and requires 3,000 bits of keystream. The correlation attack on $R4$ takes 2^{61} operations, and requires $2^{29.8}$ bits of memory and 35,000 bits of keystream. This is the dominant stage of the attack in terms of the complexities. The initial states of $R2$ and $R3$ can be recovered with about $2^{53.5}$ operations and sixty-two bits of keystream, using negligible memory.

The complexity of the complete attack on Alpha1 is 2^{61} operations. It requires 35,000 bits of keystream and 2^{30} bits of memory. A time-memory-tradeoff attack with comparable operational complexity uses 2^{68} bits of memory and 2^{30} bits of keystream. Our attack is the most effective against Alpha1 to date. The failure of Alpha1 to meet its designed security level of 128 bits means it is not an ideal candidate for securing mobile and wireless applications.

Chapter 6

Rekeying Issues in the MUGI Stream Cipher

MUGI [224] is a Pseudo Random Number Generator (PRNG) designed for use as a stream cipher. It uses a 128-bit master key and a 128-bit initialization vector. Its design strength, of 128 bits, is commensurate with the length of the key.

MUGI's structure is based on the PANAMA PRNG [58], which can be used either as a stream cipher or hash function. A schematic generalization of PANAMA and MUGI is shown in Figure 6.1. The update function Υ is composed of a linear sub-function λ and a non-linear sub-function ρ. The function λ updates the buffer, using input from both the buffer and the state. The function ρ updates the state, again input from the buffer and the state. An output filter f operating on the state produces the keystream.

The cipher is targeted towards 64-bit architectures, which means it is currently non-competitive with most of the 32-bit word-based ciphers discussed in Section 4.1. This is a situation that will almost certainly change when 64-bit architectures finally become commonplace. MUGI's mediocre performance in software is not due entirely to the mismatch between the algorithmic requirements and implementation characteristics. It has a large state space, which as discussed in Section 4.1, can lead to poor key agility, through its complex and lengthy key initialization process. In this chapter, we show how to improve MUGI's key agility for both 32- and 64-bit architectures. In Section 6.1, we describe the MUGI keystream generation and key initialization algorithms. In Section 6.2, we review

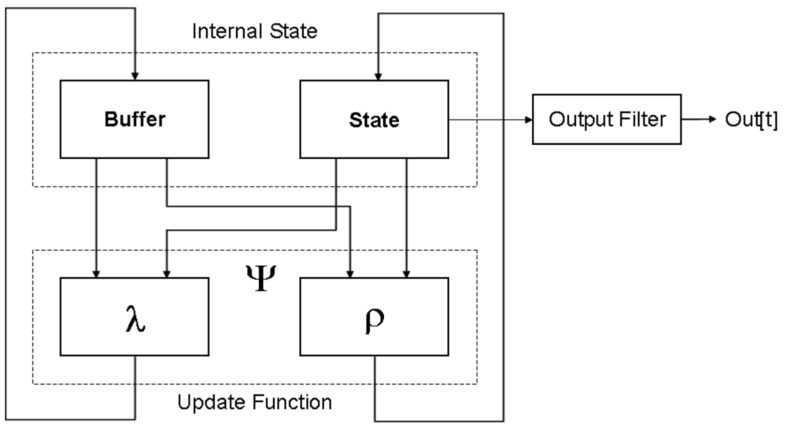

Figure 6.1: Generalization of the PANAMA and MUGI structures

previous cryptanalysis of MUGI, which leads to an interesting insight on the role of the buffer in the cipher. In Section 6.3, we discuss a peculiarity with the key initialization algorithm. In Section 6.4, we analyze further the performance of MUGI relative to other word-based stream ciphers, and suggest strategies that could be used to improve it, culminating in an algorithm for a 'modified MUGI' in Section 6.5. In Section 6.6 we perform a security and implementation analysis for the new algorithm. In Section 6.7, we summarize the contribution of this chapter.

6.1 The MUGI Algorithm

The MUGI algorithm uses 64-bit words. MUGI's internal state contains a 3-stage Non-linear Feedback Shift Register (NLFSR) denoted a, and a 16-stage Linear Feedback Shift Register (LFSR), denoted b. The output filter produces 64 bits of the output from state a at each iteration.

The non-linear function ρ is a target-heavy Feistel network structure:

$$\begin{aligned}
a_0[t+1] &= a_1[t] \\
a_1[t+1] &= a_2[t] \oplus F(a_1[t], b_4[t]) \oplus C_1 \\
a_2[t+1] &= a_0[t] \oplus F(a_1[t], b_{10}[t] \lll 17) \oplus C_2
\end{aligned}$$

where C_1 and C_2 are known constants, $(M \lll k)$ indicates leftwise k-bit rotation of M, and F is a function that uses the components of the round function of the AES [62]. Note that the state receives at most 128 bits of new material each time ρ is called. Each of the state words is used in a different way: a_0 is used to provide new material to the buffer; a_1 is used for mixing in the F function; and a_2 is used for output and feedback.

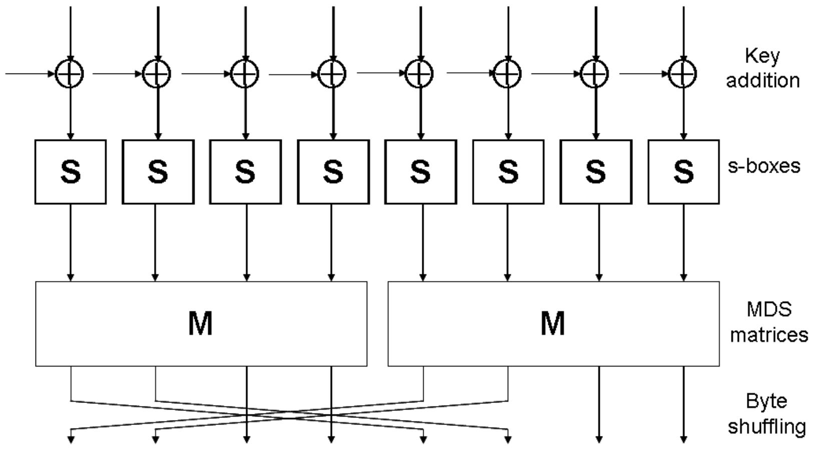

Figure 6.2: MUGI F Function

The details of the F function are shown in Figure 6.2. The function has four layers. In the first layer, which resembles key addition in an SPN, eight bytes from a buffer word are added to each of eight state bytes. In the second layer, the state is modified by eight parallel applications of the AES s-box. The third layer contains a repeated Maximum Distance Separable (MDS) matrix. The final layer consists of byte shuffling. The polynomials used in the MDS are identical to those used in AES.

Denoting stage i $(0 \leq i \leq 15)$ of the buffer as b_i and stage j $(0 \leq j \leq 2)$ of the state as a_j, the details of function λ are as follows:

$$\begin{aligned}
&b_i[t+1] = b_{i-1}[t](i \neq 0,4,10)\\
&b_0[t+1] = b_{15}[t] \oplus a_0[t]\\
&b_4[t+1] = b_3[t] \oplus b_7[t]\\
&b_{10}[t+1] = b_9[t] \oplus (b_{13}[t] \lll 32)
\end{aligned}$$

where $b_i[t+1]$ and $a_i[t+1]$ are the content of stage i of buffer b and respectively state a after the completion of t iterations.

Because of the interaction of the state and the buffer it is possible to view MUGI as a 19-stage NLFSR as in Figure 6.3 below. This view gives a clearer picture of the interactions between buffer and state and makes it easier to trace the path of the contents of each stage during operation.

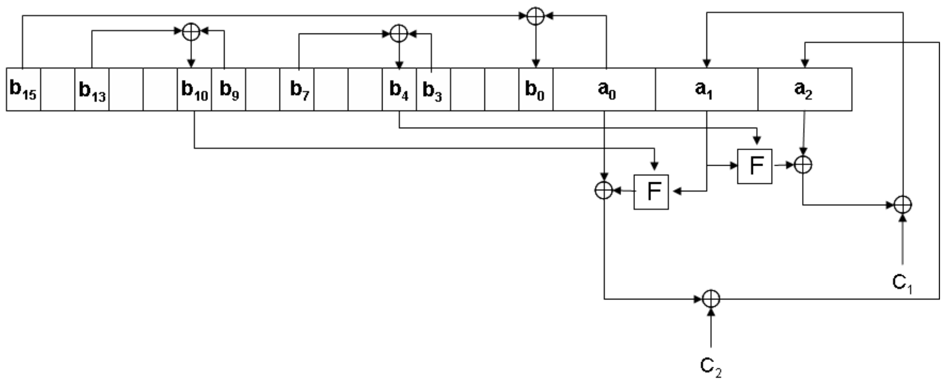

Figure 6.3: Alternative View of MUGI

Output Filter

Each application of the Υ function produces new values within the state a. The output filter selects the 64-bit block a_2 to output as the keystream.

Initialization and Rekeying

The initialization process of MUGI consists of five phases. All must be executed in full during rekeying of a master key. Only phases three, four and five are executed during rekeying of an initialization vector.

Phase 1: Master Key Injection The 192-bit MUGI state a is initialized using the 128-bit master key K. The key is divided into two segments $K_0 \parallel K_1$ and state a set, using known constant C_0, as follows:

$$a_0 = K_0$$
$$a_1 = K_1$$
$$a_2 = (K_0 \lll 7) \oplus (K_1 \ggg 7) \oplus C_0$$

Phase 2: State Mixing and Buffer Initialization The non-linear state function ρ is used to mix the state a a total of sixteen times, using a null buffer.

After each iteration, a stage in the NLFSR buffer is filled with key-dependent material from the state word a_0. The last stage in the buffer is filled first; therefore, the first stage is filled using key material which has undergone the most mixing:

$$b(K)_{15-i} = (\rho^{i+1}(a[-48], 0))_0 \quad 0 \leq i \leq 15$$

Phase 3: Initialization Vector Injection The 128-bit initialization vector $I = I_0 \parallel I_1$ is added to the mixed state a in a similar way to the key injection.

$$\begin{aligned} a[-32]_0 &= a[-33]_0 \oplus I_0 \\ a[-32]_1 &= a[-33]_1 \oplus I_1 \\ a[-32]_2 &= a[-33]_2 \oplus (I_0 \lll 7) \oplus (I_1 \ggg 7) \oplus C_1 \end{aligned}$$

Phase 4: Further State Mixing The state is again mixed, using a null buffer, sixteen times. At the end of this phase, the state is represented by:

$$a[-16] = \rho^{16}(a[-32], 0)$$

Phase 5: State and Buffer Mixing The rekeying procedure is finished by iterating the state and buffer sixteen times using the Υ function, and discarding the resulting keystream.

$$a[0] = \Upsilon^{16}(a[-16]), b(K))$$

6.2 Related Work

In [223] the designers of MUGI analyze their cipher. They claim that MUGI is immune to linear cryptanalysis because the minimum number of active s-boxes within an approximation is 22, and the maximum linear probability of an s-box is 2^{-6}. Consequently the maximum probability of an approximation is 2^{-132}; this is insufficient to attack the cipher given its design strength of 128 bits. They leverage the studied properties of the AES round function to claim immunity against a resynchronization attack that uses differential, linear or integral cryptanalysis.

In [63], it is shown that MUGI is not vulnerable to a linear masking attack due to the difficulty in finding a biased linear combination of the inputs and outputs of the non-linear function ρ. Also the large size of the state (1,216 bits) precludes a time-memory-data attack. The dependence of the state and buffer upon each other makes discovery of divide and conquer and correlation attacks non-trivial, and to date, none have been discovered. They note that MUGI passes all statistical tests from the CRYPT-X package [87].

In [166], Mihaeljevic studies a variant of MUGI in which MDS matrices are excluded from the F component of the ρ update function. Because MUGI uses the AES s-box, which is well known to produce over-defined and sparse equations, the simplified MUGI can be subjected to an XL attack. However, Mihaeljevic does not produce any definite conclusions about the complexity of the attack, except that increasing the length of the key could increase the design strength above the attack complexity (which would make the attack successful). Also Mihaeljevic need not exclude linear operations like the MDS from the attack; these enable the production of additional equations which should reduce the complexity of the attack, although increase the difficulty in rendering the over-defined equations.

In [83], the linear function λ is analysed using a system of recurrences in b_4 and b_{10} , and solved using generating functions. From this, the author discovers the period of the subsequences related to the recurrences is equal to or less than 48, and the linear complexity is 32. These properties are considered too small for use in a cryptographic application, although no attack has been forthcoming on this basis. He studies a simplified MUGI in which the buffer is made autonomous by decoupling the feedback from the state. Linear cryptanalysis is applied to both the simplified and full versions of MUGI: in both cases, the attack succeeds when compared to the large state size, but requires greater complexity than brute forcing the key. The attack is much easier on the simplified version, proving the success of the non-linear feedback between the buffer and the state. Golic finds that the algorithm is immune to the XL attack due to the large state and complex rekeying algorithm.

6.3 An Observation on Key Initialization

As shown in Section 6.1, the rekeying strategy of MUGI consists of five phases. In phase two, the fifteenth word of the buffer (b_{15}) is assigned the output $(\rho^1(a,0))_0$, which is the value of the state variable a_0 after a single invocation of the ρ function. In the ρ function, the a_0 word is modified simply by replacing its value with that of a_1 (that is, one third of the state is not changed by the ρ function). Since each buffer word is only updated once in the second phase, at the end of phase two, b_{15} contains the unmodified key word K_1, which entered the state as a_1.

Stages three and four of the initialization do not touch the buffer at all, meaning that at the start of the final stage, after thirty-two rounds of the ρ function, half of the key material is present in the buffer in its unmixed state. An attacker has to work backwards through only sixteen rounds of ρ to obtain K_1. While there is no known way of doing this faster than brute force, this is still significantly less effort than is suggested by the lengthy and complex initialization process.

The contents of the buffer at the end of stage four are shown in the following equations

$$\begin{aligned} b_{15} &= K_1 \\ b_{14} &= (K_0 \lll 7) \oplus (K_1 \ggg 7) \oplus F(b_{15}, 0) \oplus C_1 \\ b_{13} &= K_0 \oplus (K_0 \lll 7) \oplus (K_1 \ggg 7) \oplus F(b_{14}, 0) \oplus C_2 \end{aligned}$$

For $b_{12} \ldots b_0$:

$$\begin{aligned} b_{even} &= K_0 \oplus (K_0 \lll 7) \oplus (K_1 \ggg 7) \oplus F(b_{even+1}) \oplus b_{15} \oplus b_{even+2} \oplus b_{even+1} \oplus C_1 \\ b_{odd} &= K_0 \oplus (K_0 \lll 7) \oplus (K_1 \ggg 7) \oplus F(b_{odd+1}) \oplus b_{15} \oplus b_{odd+2} \oplus b_{odd+1} \oplus C_2 \end{aligned}$$

The state at the end of phase two is shown in the equations below.

$$\begin{aligned} a_0 &= b_0 \\ a_1 &= b_2 \oplus F(b_1, 0) \oplus F(b_0, 0) \oplus C_1 \oplus C_2 \\ a_2 &= b_1 \oplus F(b_0, 0) \oplus C_2 \end{aligned}$$

In phase five, all the key material, including K_1, quickly becomes mixed with other buffer and state.

Following initialization, the first word of output contains the key material K_1, but by now it is thoroughly mixed in the form of $a_0[15] \oplus F(a_1[15], (b_{11}[0] \oplus a_0[4] \oplus b_{15}[0] \oplus a_0[0] \oplus b_3[0] \oplus b_7[0]) \lll 17) \oplus C_2$.

6.4 Improving Key Agility of MUGI

Compared to all but three of the other ciphers reviewed in Section 4.1, MUGI has a large ratio of key size to state size. This can be seen in Table 6.1, which is ordered by increasing ratio of key to state size. One implication of a large state size is reduced key agility, since the key initialization algorithm needs to touch each element of the state. A rule of thumb observed in Snow, Dragon, HC-256 and MUGI, all of which mix the internal state using the update function, is that the function should be called twice for each element in the state. Scream chains each element in its masking table by iterating the update function four times on the previous element. Consequently, MUGI, Scream and HC-256, all of which have large states, also have lengthy key initialization functions and are poor performers in terms of key agility. While Dragon and MUGI have comparable state sizes, Dragon's key is twice the length, providing better security per byte of state. Its update function is much faster, so the key initialization algorithm, at a throughput of 11 cycles/byte, is completed in approximately twenty percent of the time required by MUGI.

Cipher	Key Size (bits)	State Size (bits)	Ratio
Helix	256	160	**1:0.6**
Turing	256	544	**1:2.1**
Snow	256	576	**1:2.2**
Rabbit	128	513	**1:4.0**
Dragon	256	1,088	**1:4.2**
Hiji-Bij-Bij	128	640	**1:5.0**
MUGI	128	1,216	**1:9.5**
RC4	128	2,048	**1:16.0**
Scream	128	2,432	**1:19.0**
HC-256	256	65,536	**1:256.0**

Table 6.1: Key to State size of Modern Word Based Stream Ciphers

There are two obvious strategies that can be considered to improve the performance of MUGI. The first is to migrate the cipher from a 64- to 32-bit design, by halving the size of each of the components, including the stages in the NLFSR and the words within the non-linear state. This has the added advantage that the design of MUGI now matches the architecture on which it is most likely to be implemented. It has the fatal weakness that the non-linear state naturally houses a 96-bit rather than 128-bit key. This key size is too small. Also the reduction in size of components necessitates rethinking the design of the core function F, which contains eight 8×8 s-boxes and two 32×32-bit MDS matrices. Using eight 4×4 s-boxes increases the maximum characteristic probability across four rounds from 2^{-132} to 2^{-50}, and using four 8×8 s-boxes increases the maximum probability across four rounds to 2^{-100}. In both cases, this is a significant loss of security. In this case the trade-off of security to benefit efficiency is inappropriate.

An alternative strategy is to leave the non-linear state and its ρ update function as they are, and act upon the deficiencies of the buffer. By reducing the buffer to 8×64-bit stages, for a total state size of $512 + 192 = 704$ bits, the speed of the rekeying strategy is increased significantly, the speed of the update function is slightly increased, and the security is marginally decreased. The state size is still more than five times the size of a 128-bit master key. This is the strategy that will be adopted in the modification of MUGI.

Shrinking the buffer means altering the taps used for feedback, and also the indices to stages used by the non-linear filter function. From halving the size of the buffer, the natural progression is to halve the indices of the taps and stages, leaving the order unaltered. This means that the distance between the taps decreases, to the point that the stages receive feedback from their neighbours.

Another improvement is to remove phase four of the keying scheme. In standard MUGI, this phase mixes the non-linear state sixteen times. Consequently, by the end of the initialization, each element of the non-linear state and the buffer has been modified forty-eight and thirty-two times respectively. By removing this stage, each element of the non-linear state and buffer has been altered sixteen times. This brings the cipher into line with the design principles of other ciphers, and the rule of thumb that each element of the state should be touched by a non-linear function (at least) twice.

To remove the property discussed in Section 6.3, we change the state word that is fed into the buffer in phase two. If a_1 is used as feedback to the buffer, then the state word a_0 reflects the contents of the buffer word last modified. This is a benign property, since it is destroyed immediately upon commencement of phase three. But using a_2 as feedback in phase two avoids this relationship, with the obvious proviso that as it is used post-initialization to generate output, its role in providing feedback to the buffer is localized to the key initialization algorithm.

6.5 The MUGI-M algorithm

In the modified algorithm, denoted MUGI-M, the only changes that effect the update sub-function ρ are the changes in the buffer words used as inputs:

$$\begin{aligned}
a_0[t+1] &= a_1[t] \\
a_1[t+1] &= a_2[t] \oplus F(a_1[t], b_2[t]) \oplus C_1 \\
a_2[t+1] &= a_0[t] \oplus F(a_1[t], b_5[t] \lll 17) \oplus C_2
\end{aligned}$$

The update sub-function λ operates on the buffer as follows:

$$\begin{aligned}
b_i[t+1] &= b_{i-1}[t] (i \neq 0, 2, 5) \\
b_0[t+1] &= b_7[t] \oplus a_0[t] \\
b_2[t+1] &= b_1[t] \oplus b_3[t] \\
b_5[t+1] &= b_4[t] \oplus (b_6[t] \lll 32)
\end{aligned}$$

The initialization process of MUGI-M consists of four phases. All must be executed in full during rekeying of a master key. Only phases three and four are executed during rekeying of an initialization vector.

Phase 1: Master Key Injection The 128-bit MUGI-M state a is initialized as per Phase 1 of the MUGI algorithm.

Phase 2: State Mixing and Buffer Initialization The non-linear state function ρ is used to mix the state a a total of eight times, using a null buffer. After each iteration, a stage in the buffer is filled with key-dependent material from the state word a_2. The last stage in the buffer is filled first; therefore, the first stage is filled using key material which has undergone the most mixing:

$$b(K)_{7-i} = (\rho^{i+1}(a[-16], 2))_0 \quad 0 \leq i \leq 7$$

Phase 3: Initialization Vector Injection The 128-bit IV is added to the mixed state a as per Phase 3 of the MUGI algorithm.

Phase 4: State and Buffer Mixing The rekeying procedure finishes by iterating the state and buffer eight times using the combined Υ function, and discarding the resulting keystream.

$$a[0] = \Upsilon^8(a[-8]), b(K))$$

Code (using the C language) for this algorithm is presented in Appendix E.

6.6 Analysis of MUGI-M

Cipher	Keystream Generation	Key Initialization (IV)	Key Initialization (Full)	Ratio
Units per iteration				
MUGI	181	4987	7540	**1:27.6:41.7**
MUGI-M	140	1652	2784	**1:11.8:20.0**
Cycles per byte				
MUGI	25.2	36.8	55.7	**1:1.5:2.2**
MUGI-M	19.4	12.2	20.6	**1:0.6:1.1**
Ratio	**1.3:1**	**3.0:1**	**2.7:1**	

Table 6.2: Comparison of the Speed of MUGI and MUGI-M

Table 6.2 shows the contrast in efficiency between MUGI and MUGI-M on the Intel Pentium 4 processor. In particular, there is an improvement of 200% in the speed of rekeying an IV, and 170% in full rekeying. The modified rekeying is not as efficient as the rekeying of the Dragon stream cipher (see Chapter 7), which is due to the different architectures for which the designs were intended. There is a modest 30% increase in the speed of the keystream generation, which is likely due to reduced register pressure and smaller buffer loops.

The attacks discussed in Section 6.2 are ineffective against MUGI for the following reasons: the effectiveness of the highly non-linear state function ρ, which leverages the properties of the AES block cipher; the large size of the buffer; the feedback between the internal state and the buffer; and the complex rekeying strategy. None of the attacks rely on properties of the buffer other than its size. Golic [83] argues that the properties of the buffer, when considered autonomously, are cryptographically poor. This argument is deflected by the fact that the buffer is coupled to the non-linear state, and that he is ignoring one of MUGI's great strengths. However, from this it can be claimed that by changing the location of the taps in the buffer, we are not altering any special properties of the buffer, which was constructed in an ad-hoc manner. We are aiming to repair the performance of MUGI rather than engender it with additional security properties. In the remainder of this section, the resistance of MUGI-M against individual attacks is considered.

Block-cipher style attacks

Block-cipher style attacks rely on the properties of the non-linear function: for example, the maximum differential and linear probabilities across the function. Given that only the size of the buffer, and the location of its taps have been changed, the resistance of MUGI-M against block-cipher style attacks remains unchanged from that of MUGI.

This resistance is due to the properties of the F function, which is a modified AES round function containing key additions, the AES 8×8 s-box, and two interwined MDS matrices. It is well-known that this function is resistant against differential and linear attacks. This is due to the combined effect of the s-boxes and the MDS matrices. The s-boxes in the F function have a maximum probability of 2^{-6}, although almost half of the s-box characteristics have a probability of 2^{-7}.

A well-known property of the AES MDS matrix is that it has a branch number of five, which is the lower bound on the sum of non-zero input bytes and output bytes. When combined with the other diffusion elements in the AES, the MDS causes the number of active S-boxes in four rounds of AES to be lower-bounded by twenty-five [62]. Because in MUGI and MUGI-M, the outputs of MDS matrices are permuted, the branch number does not strictly hold, and over four rounds as few as twenty s-boxes may be activated.

To launch a successful attack against the F function requires a differential that incorporates fewer than ten active s-boxes, as $2^{-7\times10} < 2^{-64}$. If a differential style attack can be launched against MUGI or MUGI-M, it will need to use fewer than four words of keystream. The F function exhibits a vulnerability to integral cryptanalysis across no fewer than four, and no more than nine rounds. The synchronous nature of the cipher means that the attacker does not have sufficient control over the inputs to launch it on either MUGI or MUGI-M. The resilience of MUGI-M against block-cipher style attacks appears to be the same as that of MUGI. If an attack of this style affects one, it will presumably affect the other.

Linear cryptanalysis

The self-evaluation report of MUGI [223] includes an analysis of linear cryptanalysis incorporating both the non-linear state and the buffer. This form of linear cryptanalysis consists of two phases: the first determines a linear approximation of ρ. In the second, a path is searched to acquire an approximation that consists only of output bits (as the internal state is not available to the attacker). For MUGI-M, the first phase remains unaltered from that of MUGI: if an approximation can be found that includes fewer than twenty-two active s-boxes, linear cryptanalysis may be possible. The second phase does not depend upon the length of the buffer; since the nature of the buffer has not been fundamentally altered, the analysis of MUGI applies equally to MUGI-M.

Time-Memory-Data trade-off attacks

MUGI-M is immune to time-memory-data trade-off attacks because it has a small key size relative to the size of the buffer. For a brute-force equivalent attack with $T = 2^{128}$, $M^2 \times D^2 = 2^{896}$. Assuming that a limit is placed on generating 2^{128} bits of keystream under one key, then to launch an attack requires 2^{287} gigabytes of memory. This is clearly infeasible.

Divide and conquer attacks

A successful divide and conquer attack on MUGI that determines the contents of the components sequentially, has a complexity of $2^{192} + 2^{1024}$, assuming that the components are autonomous. This is contrasted to a brute-force complexity of $2^{192} \times 2^{1024}$. The shorter buffer length of MUGI-M reduces this complexity to $2^{192} + 2^{512}$. This analysis ignores the fact that the components are not autonomous, and that the complexity may be much higher. The complexity of the attack needs to be less than 2^{127} to be considered successful, given the 128-bit design strength of MUGI. Therefore, divide and conquer attacks are very unlikely to succeed against MUGI-M.

Correlation attacks

A correlation attack on MUGI or MUGI-M requires a measure of correlation between the NLS and the NLFSR. No measure has been found in either cipher, due to the absence of a perceivable bias in the non-linear filter, and to the feedback between the NLS and the NLFSR. A correlation attack against MUGI-M seems unlikely.

Guess and determine attacks

Guess and determine attacks, as shown in Section 4.2.4, have been successful against a number of word-based ciphers. In a guess and determine attack against a PANAMA-style cipher, a cryptanalyst can adopt one of three approaches: fix elements within the non-linear state and use them to guess the contents of the NLFSR; fix elements within the NLFSR and use them to guess the contents of the NLS; or a hybrid approach in which elements from both components are guessed.

MUGI has shown resistance to guess and determine attacks because of the high non-linearity in the ρ function, and the large sizes of both the state and the buffer. Adopting either of the first two approaches outlined is fruitless, because the material guessed exceeds the number of bits in the master key, so a hybrid approach needs to be adopted. While this may be possible, no guess and determine attack has been possible, because no simple relationship between the non-linear state and the buffer has been discovered. As the buffers in MUGI and MUGI-M are similar in structure and size (relative to the master key size), and the ρ function is essentially unchanged, a guess and determine attack on one of the ciphers is likely to apply (with modifications) to the other.

Linear masking attacks

Linear masking attacks depend on two factors: finding a linear approximation to the non-linear filter, and finding a linear combination of the buffer that causes the bias in the non-linear filter to vanish. To date, no effective bias has been discovered in the non-linear filter ρ of MUGI, which is unaltered in MUGI-M. We do not expect that MUGI-M is vulnerable to linear masking attacks.

Algebraic attacks

Algebraic attacks depend upon developing systems of equations on the non-linear components of ciphers. In MUGI-M, the sole non-linear component is the AES s-box, which is well-known to be over-defined. The linear components of the non-linear filter and buffer allow extra equations to be added to the system. In principle, MUGI-M is vulnerable to an XL attack, with a complexity similar to

Round	State Differences	Buffer Differences
1	$\Delta a_0 = \Delta x$ $\Delta a_1 = F(\Delta x)$ $\Delta a_2 = 0$	$\Delta b_7 = 0$
2	$\Delta a_0 = F(\Delta x)$ $\Delta a_1 = F^2(\Delta x)$ $\Delta a_2 = F^2(\Delta x) \oplus \Delta x$	$\Delta b_6 = \Delta x \oplus F^2(\Delta x)$
3	$\Delta a_0 = F^2(\Delta x)$ $\Delta a_1 = F^3(\Delta x) \oplus \Delta x \oplus F^2(\Delta x)$ $\Delta a_2 = F^3(\Delta x) \oplus F(\Delta x)$	$\Delta b_5 = F(\Delta x) \oplus F^3(\Delta x)$
4	$\Delta a_0 = \Delta x \oplus F^2(\Delta x) \oplus F^3(\Delta x)$ $\Delta a_1 = F(\Delta x \oplus F^2(\Delta x) \oplus F^3(\Delta x)) \oplus$ $F(\Delta x) \oplus F^3(\Delta x)$ $\Delta a_2 = F(\Delta x \oplus F^2(\Delta x) \oplus F^3(\Delta x)) \oplus F^2(\Delta x)$	$\Delta b_4 = F(\Delta x \oplus F^2(\Delta x) \oplus$ $F^3(\Delta x)) \oplus F^2(\Delta x)$

Table 6.3: Propagation of Differences through the MUGI-M State and Buffer

that on MUGI, which shares the same non-linear filter. However, in both cases, the complexity of the XL attack exceeds the design strength of the 128-bit master key [166], and is therefore not practical.

Rekeying Attacks

MUGI-M appears to be secure from rekeying attacks, despite the fact that the key initialization algorithm mixes the non-linear state sixteen instead of forty-eight times, and the buffer sixteen instead of thirty-two times. The level of mixing per buffer stage remains the same.

Also the attacker has no control over any stage in the buffer, except indirectly through the non-linear state. No raw key material enters the buffer at any time.

Consider a resynchronization attack using multiple master keys, in which there are differences between the keys. For extra freedom, the attacker is allowed to control the difference in the initial a_2 state word. For simplicity, constants and rotations are ignored. In the best known attack, the attacker uses two keys with the word differences $(F(\Delta x), \Delta x, 0)$. The state and buffer differences for the first four iterations of the ρ function are shown in Table 6.3. Because the F function is optimized against differential cryptanalysis, and because each of the stages in the buffer is chained to previous stages, the attacker very quickly loses the ability to track differences within the keystream. As shown in previously in this section (see Block-cipher style attacks), no differentials through the F function are possible after it has been iterated four times. Table 6.3 shows that after b_6 and b_7 have been populated, subsequent words are affected by at least four iterations of the F function and therefore activate too many s-boxes for an effective related-key attack to be launched. In phase four, b_6 and b_7 are filled with material dependent upon all buffer words, so the low non-linearity present in these words in phase two is not a weakness.

6.7 Summary

In this chapter we have reviewed past cryptanalysis of the MUGI stream cipher, and pointed out a peculiarity in the key initialization algorithm, whereby one key word was visible in the buffer after thirty-two out of forty-eight iterations of the update function.

We determined that MUGI had poor key agility, compared to other word-based stream ciphers because its design targets 64-bit architectures, which are not yet commonly available, and because its large state size requires a lengthy key initialization process. The state size is large relative to the key size, so does not serve well the security-efficiency trade-offs in MUGI's design.

We suggested a variant of the MUGI algorithm, MUGI-M, in which the size of the buffer was halved, and the key initialization reduced from forty-eight to sixteen steps. This resulted in an improvement of 200% in the speed of rekeying an IV, and 170% in full rekeying. We analysed the new variant with respect to security and determined that it remains secure against attacks. This is because: we made no significant alterations to the non-linear filter; each stage in the buffer is sufficiently modified by the key initialization algorithm; and the buffer is still large relative to the key size. This alteration will serve the security-performance trade-off of MUGI well, both now and in the future, when 64-bit architectures, for which MUGI was designed, become commonplace.

Chapter 7

Dragon: A Fast Word-Based Stream Cipher

This chapter presents Dragon, a new 32-bit word-based stream cipher designed to fulfill stringent security and efficiency targets.

Dragon uses a word-based non-linear feedback shift register (NLFSR), in conjunction with a non-linear filter to produce key stream in blocks of 64 bits. The update function of Dragon has two optimal s-boxes and, like many of the stream ciphers reviewed in Chapter 4, is comparable to a reduced round block cipher. We have analysed the security of Dragon using modern cryptanalytic techniques, and believe it is suitable for use as a secure cryptographic primitive for at least the next decade. Dragon has a throughput of gigabits per second in both modern software and hardware, and requires around four kilobytes of memory, so is suitable for use in constrained environments. Both the keystream generation and key scheduling algorithms of Dragon are efficient, making it especially suitable for applications that require frequent rekeying, as are found in the mobile and wireless communications paradigms.

Dragon can be considered an evolution of the output feedback mode (OFB) of block ciphers that overcomes a shortcoming of that mode: that the output keystream is also the feedback to the internal state. For many ciphers in OFB mode, birthday paradox attacks exploit this knowledge of the feedback, but are prevented from doing so with Dragon, which produces separate output and feedback words from the update function. Dragon follows the rule-of-thumb that the internal state size of a stream cipher must be at least twice the designed security level, in order to prevent time-memory-data trade-off attacks [30]. To increase the difficulty of the guess and determine attacks [95], Dragon selects taps from the NLFSR according to a Full Positive Difference Set (FPDS).

Section 7.1 presents the specification of the cipher. Section 7.2 describes the design decisions behind the Dragon algorithm. Section 7.3 outlines some of the properties of the cipher, including its expected period, absence of weak keys and a statistical analysis. Section 7.4 includes a security analysis of Dragon using modern cryptanalytic techniques. Section 7.5 discusses the performance of Dragon in software and hardware, and associated implementation issues. Section 7.6 concludes this chapter with a summary.

7.1 Specification of Dragon

Dragon is a stream cipher constructed using a single word based 1,024-bit NLFSR and a 64-bit memory register M. From the 1,092-bit state, the combined update function/output filter F, which is called once per round, produces a 64-bit word of output. The state is initialized by processing a 256-bit secret master key in conjunction with a 256-bit initialization vector (IV) through Dragon's key scheduling algorithm.

7.1.1 Dragon's State Update Function (F Function)

The F function is a 192-to-192 bit three-stage reversible mapping that is used in both the key initialization and keystream generation algorithms. It takes six 32-bit words as input (denoted $a, b, c, d, e,$ and f) and produces six 32-bit words as output (denoted a', b', c', d', e' and f'). The F function has two component functions denoted G and H, as described below. It uses the G and H functions to provide algebraic completeness [117] and high non-linearity. Diffusion is provided through a network of modular and binary additions. The F function is depicted in Figure 7.1. It is also notationally described in Table 7.1, where $\oplus$ denotes exclusive-or and $\boxplus$ denotes addition mod 2^{32}.

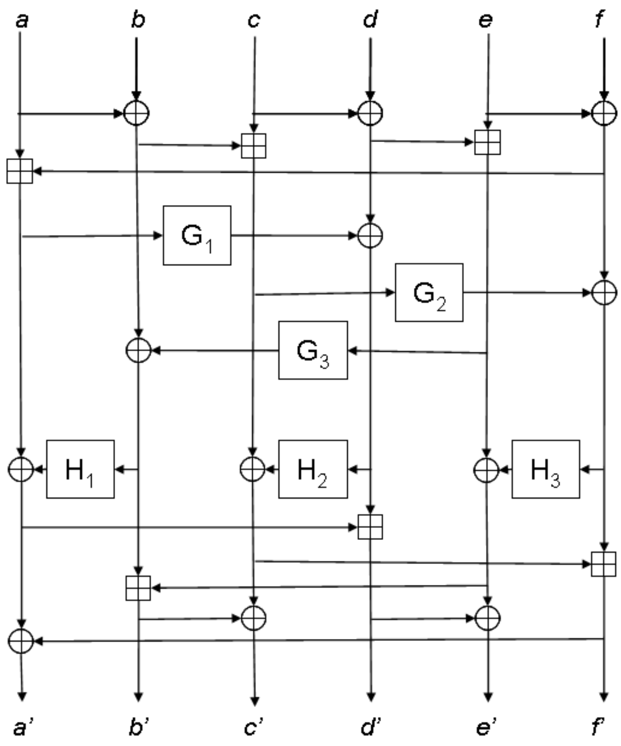

Figure 7.1: Schematic of Dragon's F Function

Input = { a, b, c, d, e, f }			
Pre-mixing Layer:			
1.	$b = b \oplus a$;	$d = d \oplus c$;	$f = f \oplus e$;
2.	$c = c \boxplus b$;	$e = e \boxplus d$;	$a = a \boxplus f$;
S-box Layer:			
3.	$d = d \oplus G_1(a)$;	$f = f \oplus G_2(c)$;	$b = b \oplus G_3(e)$;
4.	$a = a \oplus H_1(b)$;	$c = c \oplus H_2(d)$;	$e = e \oplus H_3(f)$;
Post-mixing Layer:			
5.	$d' = d \boxplus a$;	$f' = f \boxplus c$;	$b' = b \boxplus e$;
6.	$c' = c \oplus b$;	$e' = e \oplus d$;	$a' = a \oplus f$;
Output = { a', b', c', d', e', f' }			

Table 7.1: The F Function: Core of the Dragon Stream Cipher

```
Input = { $x$ }

    1.  $x = x_0 \| x_1 \| x_2 \| x_3$
    2.  $y = S(x_0) \oplus S(x_1) \oplus S(x_2) \oplus S(x_3)$

Output = { $y$ }
```

Table 7.2: Virtual S-box Construction in the Dragon Cipher

G and H Functions

The G and H functions are constructed from two 8×32-bit s-boxes, S_1 and S_2 to form virtual 32×32 s-boxes. These s-boxes are the only components of Dragon that are explicitly non-linear, and are included in Appendix D. In the virtual mapping process, the 32-bit input x is broken into four bytes x_i $(0 \leq i < 4)$. Each byte is passed through an 8×32 s-box and the four 32-bit outputs y_i combined using binary addition. This process is shown in Table 7.2.

The individual G and H functions are defined as:

$$\begin{aligned}
G_1(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\
G_2(x) &= S_1(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\
G_3(x) &= S_1(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_1(x_3) \\
H_1(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_2(x_2) \oplus S_1(x_3) \\
H_1(x) &= S_2(x_0) \oplus S_2(x_1) \oplus S_1(x_2) \oplus S_2(x_3) \\
H_1(x) &= S_2(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_2(x_3)
\end{aligned}$$

7.1.2 Key Scheduling Algorithm

Dragon has a simple keying (and rekeying) strategy that uses a 256-bit key K and a publicly known 256-bit initialization vector IV. Dragon's 1,024-bit internal state W is divided into eight 128-bit words, labelled W_0 to W_7 and is initially filled by concatenating K and IV with their bitwise sum and its complement such that $W = K \parallel K \oplus IV \parallel \overline{K \oplus IV} \parallel IV$. A 64-bit register M is initially filled with a constant and updated in each round using two of the output words of the F function. The state initialization process makes extensive use of the F function, which simplifies analysis and increases implementation efficiency. The initialization involves sixteen iterations of the F function, as shown in Figure 7.2 and Table 7.3. To protect against unknown future attacks, and against attacks that require large amounts of keystream, the cipher should be rekeyed at least once for every 2^{64} bits of keystream generated.

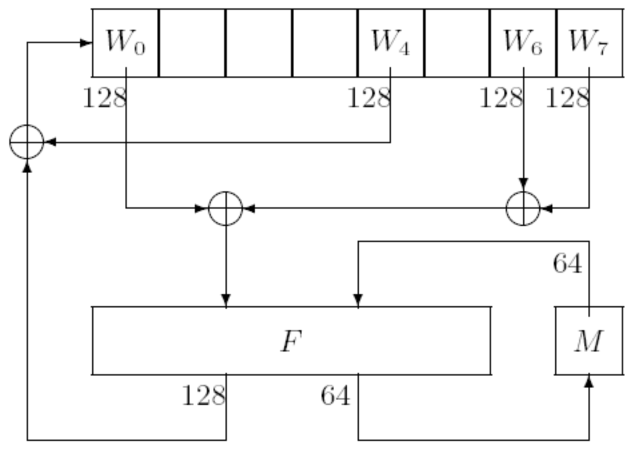

Figure 7.2: Dragon's Key Initialization Algorithm

```
Input = { $K, IV$ }
    1.    $W_0 \| ... \| W_7 = K \| K \oplus IV \| \overline{K \oplus IV} \| IV$
    2.    $M$ = 0x0000447261676F6E
Perform steps 3-8 16 times
    3.    $a \| b \| c \| d = (W_0 \oplus W_6 \oplus W_7)$
    4.    $e \| f = M$

    5.    $\{a', b', c', d', e', f'\} = F(a, b, c, d, e, f)$

    6.    $t = (a' \| b' \| c' \| d') \oplus W_4$
    7.    $W_i = W_{i-1}, 1 \leq i \leq 7$
    8.    $W_0 = t$
    9.    $M = e' \| f'$
Output = $\{W_0 \| ... \| W_7\}$
```

Table 7.3: Dragon's Key Initialization Algorithm

7.1.3 Keystream Generation Algorithm

Dragon has a large NLFSR of one kilobyte divided into thirty two 32-bit words $B_i, 0 \leq i \leq 31$. During each round, six words from the internal state are used as inputs to the F function. The indices to these words form a Full Positive

Difference Set (FPDS): these are 0, 9, 16, 19, 30 and 31. The 64-bit register M acts as a counter in keystream generation, with the initial value for keystream generation being the final value of M defined by the key initialization algorithm (that is, the value of the counter is maintained between algorithms). Each round of the keystream generation results in the output of a 64-bit word k, an updated state B and memory M. Figure 7.3 and Table 7.4 show one round of keystream generation.

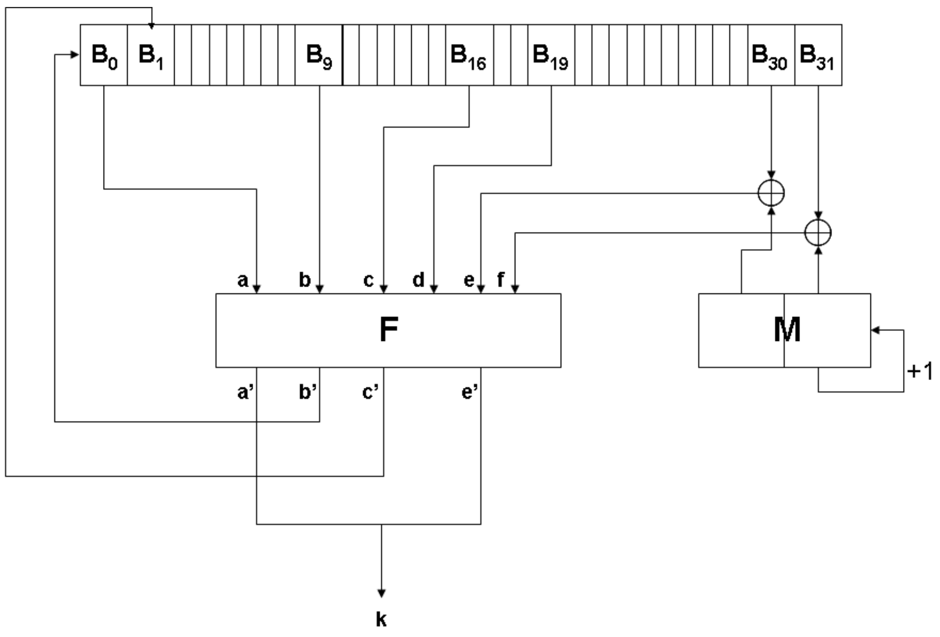

Figure 7.3: Dragon's Keystream Generation Function

Input = { $B_0 \parallel ... \parallel B_{31}, M$ }	
1.	$(M_L \parallel M_R) = M$
2.	$a = B_0, b = B_9, c = B_{16}, d = B_{19}, e = B_{30} \oplus M_L, f = B_{31} \oplus M_R$
3.	$(a', b', c', d', e', f') = F(a, b, c, d, e, f)$
4.	$t_0 = b', t_1 = c'$
5.	$B_i = B_{i-2}, 2 \leq i \leq 31$
6.	$B_0 = t_0, B_1 = t_1$
7.	$M = M + 1$
8.	$k = a' \parallel e'$
Output = { $k, B_0 \parallel ... \parallel B_{31}, M$ }	

Table 7.4: Dragon's Keystream Generation Function

7.2 Design Principles of Dragon

The following section outlines the theory behind the design of different components within Dragon, including the F function used in both the key initialization and keystream generation algorithms, the s-boxes incorporated into the F function, and the key scheduling algorithm.

7.2.1 Design of F Function

The F function is a reversible mapping of 192 bits to 192 bits. It can be divided to three parts: pre-mixing, substitution, and post-mixing. Each step is designed to allow for parallelization, giving Dragon its speed.

All the keystream words and feedback words are dependent on all inputs, both at the bit level and word level. A single bit change in any of the six input words results in completely different keystream and feedback words.

7.2.2 Design of S-boxes

Dragon uses two 8×32 s-boxes that have been designed heuristically to satisfy a range of important security related properties. As shown in Table 7.2, these s-boxes are used in the construction of the 32×32-bit non-linear functions G and H. Both s-boxes were designed to have balanced component boolean functions with:

- best known non-linearity of 116;
- optimum algebraic degree 6 or 7 according to Siegenthaler's trade-off [212];
- low autocorrelation;
- distinct equivalence classes;
- all XOR pairs satisfying:
  - better than random non-linearity with 102 minimum;
  - almost balanced (the imbalance is not more than 16);
  - distinct equivalence classes;
  - same optimal degree as the components.

We adopt a standard notation (n, t, d, x, y) to describe Boolean function properties where n is the number of variables, t is the order of resiliency (where $t = 0$ indicates a balanced function), d is the algebraic degree, x is the non-linearity and y is the largest magnitude in the autocorrelation function. All the components of S_1 are (8,1,6,116,y) where $32 \leq y \leq 48$ which is considered sufficiently low. S_1 functions achieve the highest non-linearity possible for resilient functions. All the components of S_2 are (8, 0, 7, 116, 24), where we note that the achieved autocorrelation of 24 is the lowest known for balanced functions of this size.

These s-boxes were created one output bit at a time using heuristic techniques. Existing methods [169] were adopted to generate the individual functions, then they were compared to the existing s-box functions to check the above-listed requirements for the XOR pairs. When the candidate function was acceptable, it was appended to the s-box, otherwise another function was tested. We found it was possible to generate thirty-two functions for each s-box, while satisfying the stringent requirements outlined above.

7.2.3 Design of Key Initialization Algorithm

The key initialization and keystream generation algorithm of Dragon both use the F function for ease of analysis, implementation and efficiency. However, the key setup of Dragon is deliberately designed to be different to keystream generation, so that the mapping of internal state to the feedback is different.

There are three differences between the key setup and the keystream generation: the way in which the 64-bit register M is used, the size of the feedback and the FPDS selection used.

The F function is a reversible mapping, and the design of the key setup network uses this property of F to produce a bijective process. For any unique pair of K and IV, the key setup procedure initializes the internal state and M to unique values.

A small number of rounds (sixteen) in key setup translate directly into high rekeying performance. This makes Dragon very competitive in practical applications that require frequent rekeying, such as mobile and wireless transmissions that usually use the frame number as the IV. The feedback of Dragon consists of four words of the F function outputs, totalling 128 bits (in the keystream generation, it is only sixty-four bits). This means that F can mix K and IV effectively with minimum number of rounds.

A different FPDS has to be chosen for the key setup because of the change in size of the feedback. The indices from the internal state – 0, 4, 6 and 7 — form a FPDS both in the forward and reverse direction. This is designed to frustrate the cryptanalysis of key setup by guess and determine techniques.

7.3 Analysis of Dragon

This section discusses properties of the Dragon cipher, in terms of its statistical properties, its period length, and the absence of weak keys.

7.3.1 Statistical Tests

The frequency, binary derivative, change point, subblock and runs tests were executed with thirty streams of Dragon output, each eight megabits in length. The sequence and linear complexity tests were executed for the thirty streams with two hundred kilobits each. Dragon passed all pertinent statistical tests provided by the CRYPT-X [87] package.

7.3.2 Period Length

Given that Dragon has a 1,024-bit internal state, the expected period of the internal state is 2^{512}, assuming the mapping is pseudo-random [47]. Each round of Dragon is under the influence of a 64-bit counter, M. Since the counter M has a period of 2^{64}, the period of Dragon's internal state is lower bounded by 2^{64} 64-bit words, or 2^{70} bits. Taken together, the internal state and the counter M give Dragon an expected period of 2^{576} 64-bit words, or 2^{582} bits.

The amount of keystream produced by a unique pair of K and IV is limited to 2^{64} bits (in most applications the actual keystream would be much smaller). This is a small fraction of the lower bound of the period (and an even smaller fraction of the expected period), and therefore avoids the possibility of keystream collision attacks.

7.3.3 Weak Keys

Weak keys are those keys that cause some operations to have no effect on the calculation of the feedback or the output keystream.

Dragon is designed to avoid weak keys. The internal state is a non-linear feedback shift register that avoids fixed points through its use of a counter. Therefore the all-zero state, which is problematic in many LFSR-based ciphers, does not produce weak keys in Dragon.

While it is easy to bypass the pre-mixing phase of a single iteration of the F function by having repetitive inputs, such as all zeroes, or all ones, it is only possible for the first of the sixteen iterations of F in the key scheduling algorithm. Also, selected values are limited to the first four inputs of the F functions, as the last two inputs take the value of M, which is beyond the control of the attacker. The network of G and H functions ensure that the initial states which bypass the pre-mixing phase cannot bypass the s-box or post-mixing layers in F. We believe that the above design features provide a strong guarantee that there are no weak keys for Dragon.

7.4 Cryptanalysis of Dragon

The following section demonstrates how Dragon prevents different methods of cryptanalysis.

7.4.1 Related Key and IV Attacks

The Dragon rekeying strategy is simple, and the use of initialization vectors provides a way to reuse a master key without generating identical keystreams. The rekeying strategy prevents related key and IV attacks before even the first word of output is produced using the keystream generation algorithm. During each iteration of the highly non-linear F function, the 128 leftmost bits of the internal state are populated with the four outputs of the F function (the state is shifted by 128 bits before the following iteration). After eight rounds, all of the initial keying material in the state has been replaced by unknown output from the F function. After sixteen rounds, each bit of the key is mixed into all words of the initial state.

Of the six inputs to the F function, four words are taken directly from the keyed internal state, while two are taken from the 64-bit register M. The contents of this register are initially known, since it is determined by a published constant. Also, the register cannot be manipulated by the attacker in the same way as the internal state, since it is not keyed. Two outputs from the function feedback to

1	0	ΔA	0	0	0	0	0	0
2	0	0	ΔA	0	0	0	0	0
3	0	0	0	ΔA	0	0	0	0
4	0	0	0	0	ΔA	0	0	0
5	ΔA	0	0	0	0	ΔA	0	0
6	ΔB	ΔA	0	0	0	0	ΔA	0
7	ΔC	ΔB	ΔA	0	0	0	0	ΔA
8	ΔD	ΔC	ΔB	ΔA	0	0	0	0
9	ΔE	ΔD	ΔC	ΔB	ΔA	0	0	0
10	ΔF	ΔE	ΔD	ΔC	ΔB	ΔA	0	0
11	ΔG	ΔF	ΔE	ΔD	ΔC	ΔB	ΔA	0
12	ΔH	ΔG	ΔF	ΔE	ΔD	ΔC	ΔB	ΔA

Table 7.5: Propagation of Non-zero Differences In Internal State of the Dragon Stream Cipher

the memory, making its value hard to determine after the first round. All output words of F are affected by the memory, increasing the difficulty that the attacker faces in controlling inputs to subsequent rounds.

Diffusion One strategy in an attack is to minimize the number of words with a non-zero difference in the internal state. The aim of this strategy is controllability. The larger the number of non-zero words used as input to the non-linear function, the more complex the resulting output. The key schedule of Dragon is designed so that after twelve rounds, even a initial single word difference is propagated to all words in the internal state. This is demonstrated in Table 7.5, which is formed by an examination of the outer structure of Dragon, treating the F function as a highly non-linear black box with the characteristic $\Delta A \rightarrow \Delta B$.

Having non-zero differences in all words after twelve out of sixteen rounds leaves an ample margin to ensure an attacker is unable to determine the state contents after rekeying. The speed of this diffusion is aided by the fact that the first word of the state is used as input to F function, and the the output of the F function replaces the first word.

Even a single round of Dragon F function prevents high probability differentials due to its use of the G and H functions, and its high diffusion. A single input difference is propagated to differences in each of the six outputs. The F function consists of three layers: pre-mixing, confusion through s-box application, and post-mixing. Referring to the notation of section 7.1, only inputs a, b, c and

d can be initially and indirectly controlled by an attacker, since e and f come from internal and inaccessible memory.

The attacker may wish to make use of the fact that b and d are mixed with only one other word in the pre-mixing phase, while a and c are mixed with two others. For the input

$$-(e \oplus f), b, -(b \oplus e \oplus f), -(b \oplus e \oplus f), e, e \oplus f)$$

the pre-mixing stage produces the output

$$(0, b \oplus -(e \oplus f), 0, 0, e, e \oplus f)$$

For difference input Δb, this produces the difference $(0, \Delta b, 0, 0, 0, 0)$ since e and f are at this stage constants. This bypasses the G row of s-boxes and activates a single s-box in the second row to produce the post-mixing input

$$(\Delta H_1(\Delta b), \Delta b, 0, 0, 0, 0)$$

The post-mixing output is

$$(\Delta H_1(\Delta b), \Delta b, \Delta b, \Delta H_1(\Delta b), 0, 0)$$

At this stage, all of the feedback words to the internal state are non-zero. However, the difference of the feedback to the internal state is still zero. During the next round, the attacker can choose to keep the memory M to zero, by choosing an appropriate value of W_5. During the third round, the attacker needs to choose the correct value of W_4, which has already been used to influence previous inputs to the F function. Combined with the fact that the inputs to the F function in the third round come from all but two of the NLFSR words, the attacker cannot meet the input differences necessary to keep the value of the memory to zero. Consequently Dragon is not vulnerable to related key attacks that are more efficient than a brute force search of the 256-bit key.

7.4.2 Time-Memory-Data Trade-off Attacks

As discussed in Section 4.2.1, Time-Memory trade-off attacks [30] rely on pre-computation to reduce the effort required for a key recovery attack on a keystream.

The pre-computation increases the memory requirements of the attack, but since it can be performed off-line, reduces the time complexity of the attack. The off-line computation involves determining the relationships between internal states and keystream prefixes. By observation of online keystream prefixes, the associated internal states can be determined.

One way to thwart time-memory-data attacks is to increase the internal state of a cipher relative to its stated design strength. Dragon has an internal state space of 1,088 bits (including the 64-bit memory). Since the design strength of Dragon is 256 bits, the time-memory trade-off attack is infeasible. For the brute-force equivalent attack with $T = 2^{256}$, data requirements are limited to 2^{64} bits, which imposes a lower bound on memory for the attack of 2^{896} bits. As of 2004, a state-of-the-art computer, between its physical memory and hard-disk, possess around 2^{37} bits of memory. No network of computers in the world comes close to 2^{896} bits of memory, so the attack is infeasible.

7.4.3 Guess and Determine Attacks

The indices {0, 9, 16, 19, 30, 31} of the state elements used in Dragon's update function form a full positive difference set. This is a design decision to prevent guess and determine attacks [95].

In keystream generation, guessing six inputs (192 bits) to F in a round allows an attacker to calculate the feedback words b' and c' and the keystream words a' and e', which can be used to discard most incorrect guesses. At this point the attacker has knowledge of the state words at indices {0, 1, 2, 11, 18, 21} and some information about the value of M. The only state word that is both known by the attacker and used in the next round, has the index 0, so the attacker needs to guess the inputs for {9, 16, 19, 30, 31}, constituting a further 160 bits of guessed material. This exceeds the operational requirements for mounting a brute-force attack. The attacker can attempt to jump ahead to a future keystream word pair; for example, to the third round, in which case he needs to guess the inputs for {2, 3, 4, 13, 20, 23}, for a total of 384 bits of guessed material. If he skips too far ahead, his knowledge of the state words diminishes as the values become shifted off the NLFSR. In fact, the best attempt the attacker can make is to jump to the sixth round and guess the indices for {8, 10, 26, 28} which requires guessing a further 128 bits. This still exceeds the 256 bits required for a brute force attack. In addition, the interplay of B_{30}, B_{31} and M means there will be more than one set of values for these three elements for an unique pair of e and f, further complicating the cryptanalytic attempt by guess and determine attack.

The attacker is unable to reduce the complexity of a guess and determine attack by guessing individual state bytes, rather than whole words. The use of large s-boxes (G and H functions are effectively 32×32 s-boxes) means that guessing three of the four input bytes is insufficient to deduce any byte of the s-box output.

To calculate keystream words from two rounds of Dragon, the attacker is required to guess more than 256 bits of the internal state. This is worse than exhaustive key search, and makes guess and determine attacks on Dragon infeasible.

7.4.4 Distinguishing Attacks

If the output sequence of a stream cipher can be statistically distinguished from a random sequence, then the cipher is not strong enough for cryptographic applications. Dragon is designed with a large state and complex initialization and update function. It has no linear masking, and is therefore immune to this type of distinguishing attack [49]. Dragon is expected to have a very large period of 2^{582} and it passes standard statistical tests for randomness. The amount of keystream output for an unique pair key and initialization vector is limited to 2^{64} bits. We conjecture that it is impractical to collect an amount of output sufficient to distinguish Dragon keystream output from a random binary sequence.

7.4.5 Linear Approximations

Lemma 15 from [208] conjunctures that the non-linearity of a composite function can be calculated as follows. Let g be a function on V_{s+t} defined by

$$g(x_1, \ldots, x_s, y_1, \ldots, y_t) = f_1(x_1, \ldots, x_s) \oplus f_2(y_1, \ldots, y_t).$$

Then the non-linearity of g satisfies $N_g \geq 2^{s+t-1} - \frac{1}{2}P_1 \cdot P_2$, where P_1 and P_2 are the maximum Walsh-Hadamard transform values of f_1 and f_2 respectively.

The G and H functions of Dragon are composed from two 8×32 s-boxes, S_1 and S_2. Both s-boxes have all outputs with nonlinearity 116, therefore $P_{S_1} = P_{S_2} = 2^8 - 2 \cdot 116 = 24$. The non-linearity of the output bits of the G and H functions can then be calculated as $N_G = N_H \geq 2^{8+8+8+8-1} - \frac{1}{2} \cdot 24 \cdot 24 \cdot 24 \cdot 24 =$

$2^{31} - 165888$. The best affine approximation to the G or H function output bits has bias no greater than $\frac{2^{31}-2^{31}+165888}{2^{31}} = 2^{-14.66}$. At any given round, the keystream words of Dragon are the results of five G or H functions each, hence the best affine approximations to the Dragon F function output bits has bias no greater than $(2^{-14.66})^5 = 2^{-73.3}$.

Linear cryptanalysis requires equations relating the key bits to the internal state bits, and in turn the keystream bits, where the internal state variables can be cancelled. The complete mixing of Dragon's key setup avoids the divide and conquer approach, therefore all the internal state variables are needed in the linear equations. The output keystream will be dependent on all 1,024 bits of the initial internal state after eight iterations of F. The bias of the best affine approximation over eight iterations of F is no greater than $(2^{-73.3})^8 = 2^{-586.4}$. As the key size of Dragon is 256 bits, an attack on Dragon using linear approximation has complexity greater than exhaustive key search.

7.4.6 Algebraic Attacks

Algebraic attacks on keystream generators [53] to date have been concentrated on LFSR based generators. The general attack model consists of the internal state S, the linear update function L and the output function f. Let S_0 denote the internal state at time $t = 0$, and $L^t(S_0)$ denote the internal state at time t. The attacker constructs a system of equations relating the internal state bits with the observed keystream bits, where $z_t = f(L^t(S_0))$ at time t. The attacker can set up a large number of equations just by merely collecting keystream bits, since the internal state at time t can easily be derived from the linear nature of LFSRs.

This model cannot be applied to Dragon since the update function is non-linear. Let the non-linear update function be N, then the equation becomes $z_t = f(N^t(S_0))$. Note that N has a poor linear approximation of $2^{-73.3}$ as shown in Section 7.4.5. The lack of the linear update function means the attacker can not simply calculate the internal state at time t to construct the system of equations.

Suppose that the attacker finds a method for constructing a system of equations for Dragon; however, he or she will find that the degrees of equations grow exponentially, since any output of G or H is a degree seven function of the inputs because S_2 has algebraic order seven. If we linearize the system by approximating $\boxplus$ operation with $\oplus$, we can then write equations of degree $7^2 = 49$ that maps the 192 input bits to the first 64 output keystream bits. However, the feedback is used immediately in the production of the next 64 bits of keystream, and results in degree $7^4 = 2,401$ equations. Note that at this point, the inputs consist of only 352 bits, and therefore the equations would in fact be of degree 352. The degree of the equations would grow to the full size of the internal state, 1,024, after eight iterations of the F function, or equivalently, after 512 bits of keystream were produced.

Using the technique published in [55] to describe the 8×32 s-boxes of Dragon using quadratic equations results in 565 quadratic equations in 256 monomials for each s-box (identical to the analysis of CAST [2]). Again, approximate $\boxplus$ with $\oplus$; after eight iterations of F, the system of equations also has degree 1,024 as well. This is to say, even if there exist some annihilators [165] that reduce Dragon's Boolean functions to quadratic, the degree of the overall equations would still grow to unmanageable sizes.

It is clear that the system of equations for Dragon will be very difficult to solve, if it is solvable at all. Furthermore, it will require far more effort than exhaustive key search since solving techniques all have complexities exponential in the degree of the equations. It is interesting to note that with the modular addition in place, it will be even more difficult for algebraic attacks to work on Dragon.

7.5 Implementation and Performance

Dragon is designed to be efficient in both software and hardware, in terms of throughput and a small implementation footprint. Its 32-bit word size is chosen to match that of the ubiquitous Intel Pentium family, since this leads to the best software efficiency on that platform. See Chapter 8 for more information relating to the implementation-related design principles for the cipher. Test vectors for the cipher are given in Appendix C.

7.5.1 Software

Dragon is very efficient in software. Most operations are expected to perform with latencies of $\frac{1}{2}$ or 1 cycles on modern processors, such as the Intel Pentium family.

On an Intel Pentium 4, a naïve C implementation of Dragon produces one byte of keystream every 6.74 clock cycles. This is competitive with many of its peers, including SNOW 2 (5.5 cycles/byte), Turing (6.1 cycles/byte) and RC4 (7.1 cycles/bytes) [97]. On a 3.2 GHz Pentium 4 (Northwood), the throughput of Dragon is 3.8 gigabits per second. Complete rekeying of Dragon takes 1,395 cycles.

Storage requirements include 2,048 bytes to store Dragon's two 8×32 s-boxes, 1,024 bits (128 bytes) for the internal state, and a further eight bytes for the 64-bit counter. Including temporary variables and an object code size of 2,810 bytes, Dragon has memory requirements totalling 5,890 bytes. This is suitable for even very constrained environments.

7.5.2 Hardware

The design of Dragon allows high degree of parallelization in hardware. The operations on the six inputs of the F function can be divided into three groups, each operating on two inputs. The pre-mixing and the post-mixing are implemented using 32-bit modular adders. The G and H functions are implemented using look-up tables and exclusive-or operations. The hardware complexity is about 6,524 gates and 196,672 bits of memory. On Samsung 0.13um ASIC running at 2.6 GHz, the minimum delay is 2.774 ns with a throughput of 23 Gbps.

The speed in hardware can be improved by using m-parallel-structure proposed in [148]. This hardware implementation strategy applies to all shift registers, and achieves an m times increase in efficiency with m times increase in hardware complexity. On Altera FPGA/CPLD running at 16.67 MHz, an implementation of Dragon achieves a throughput of 1.06 Gbps with sixteen times hardware complexity.

7.6 Summary

This chapter presents Dragon, a new stream cipher constructed upon a word based non-linear feedback shift register. The key and initialization vector are both 256 bits in length. Dragon is designed with both security and efficiency in mind. It has been shown that the keystream sequence produced by Dragon is secure against all known cryptanalytic attacks. The most effective attack is brute-forcing the key, which assuming a known initialization vector, has an expected complexity of 2^{255} bits.

Chapter 8

Implementation of Symmetric Ciphers on the Intel Pentium 4

The days of slow block and bit-based stream ciphers that operate in the range of megabits per second are over. In today's busy networked world, where e-commerce demands rapid transactions and data transfer, efficiency of ciphers is nearly as important as their security.

Certainly the number of seemingly secure block and stream ciphers demands that new entrants into the arena must distinguish themselves in ways other than estimated bit-security. The metric by which the speed of symmetric ciphers is measured is either in gigabits of throughput per second, or clock cycles consumed to produce one byte of ciphertext or keystream. The Advanced Encryption Standard [62] set a benchmark for block ciphers of about 25 cycles/byte on the Intel Pentium family. Ciphers which fail to approach this benchmark are not competitive. Recent times have seen the emergence of word-based stream ciphers with clock cycle metrics of single digits, one of the more notable being RC4 [5] at around seven cycles/byte on the Pentium 4. In a cipher, size and speed are inextricably linked, and an improvement in one of these measurements is not necessarily to the detriment of the other.

This chapter investigates issues pertaining to the implementation of symmetric ciphers on the Intel Pentium 4, the most ubiquitous personal processor. The Pentium 4 boasts a range of features that Intel claimed would revolutionize the performance of many personal applications, of which cryptography was named as one [102]. Section 8.1 describes the features of the Pentium 4's high-level architecture that are pertinent to symmetric cipher design. Section 8.2 gives a brief overview of platform-independent optimization rules. These rules apply particularly to programmers of high-level languages, since assembly-level programmers require more awareness of the underlying architecture. Section 8.3 investigates the performance of some cryptographic primitives on the Pentium 4. Finally, Section 8.4 describes the implementation and optimization of the Dragon stream cipher on this chip.

8.1 Architecture

Programmers of high-level applications generally do not need to be overly aware of the architectures on which they are programming. They value portability, for the reduction of programming effort that it brings, but which is antithetical to architecture-based optimizations. For them, optimization advice comes in the form of "code in high-level languages; leave assembly language to optimizing compilers".

Implementation of symmetric ciphers is an exception. These ciphers have extremely small code sizes, limited to hundreds of instructions, and speed is paramount. Optimization at the assembly level can be extremely rewarding, particularly when writing for recent architectures, for which compilers may not be capable of squeezing the last cycle of computing power.

Even when using high-level languages to implement ciphers, knowledge of the underlying architecture can be vital. For example, a cipher that requires more registers than are available, can never be optimal, irrespective of the compiler used. Knowledge of the register set can prevent unfortunate designs that lead to bad performance.

The Intel Pentium 4 uses a new "Netburst" architecture. The architecture is complex and more fully described in [101], [102], [103]. Computer architectures change rapidly and optimization rules that apply to one chip may not apply to their successors. For example, the U-V pipeline pairing rules for the Pentium processor, as described by [189] did not apply to the Pentium Pro processor or its successors; likewise the 4-1-1 decoding rules for the Pentium III processor [104] no longer apply to the Pentium 4.

The rules for optimizing the Pentium 4 are fewer than those for its recent predecessors. In some ways, it is a more difficult target since its increased complexity makes optimization effects unpredictable.

The following sections describe some of the architectural features that designers and implementers of symmetric ciphers should bear in mind. Section 8.1.1 describes the register set of the Pentium 4, which should influence the number of variables used within a cipher design. Section 8.1.2 discusses memory, and highlights a problem with using large tables or code blocks in fast ciphers. Section 8.1.3 describes the mechanism for executing programs. This affects the constructs that cryptographers choose for their cipher designs or implementations.

8.1.1 The Register Set

Registers are very fast memory locations that reside in the core of the processor. Because they operate at clock speed, rather than the speeds of slower general memory, they are highly prized to the programmer. Consequently either the programmer, or the compiler, attempts to map variables within high-level code directly to these registers, to achieve optimal performance.

Unlike other architectures, such as the Alpha and Intel's failed iTanium architecture, the Intel Pentium family (and its predecessors in the x86 family) are register poor. Initially for reasons of cost, and later for backwards compatibility, all members of the x86 family have sported only a small set of general purpose registers to which the majority of the processor's instruction set applies. For the Intel Pentium 4, this set (shown in Figure 8.1) consists of eight 32-bit registers labelled **EAX**, **EBX**, **ECX**, **EDX**, **EBP**, **ESP**, **ESI** and **EDI**. Not all registers are treated equally: for example, the **IMUL** and **MUL** multiplication instructions work exclusively upon the **EAX** and **EDX** registers, which must be available by the time the instruction is ready for processing. Likewise, **ESI** and **EDI** are used for string operations and **EBP** and **ESP** are reserved for stack operations.

The registers are general purpose in the sense that they can be used by the programmer for any task, but when the number of variables used in a code segment exceeds the number of registers available within the processor, the register contents need to be preserved to the stack (using the **PUSH** instruction), and later retrieved from the stack (using the **POP** instruction). This represents an unexpected and possibly expensive cost (a cumulative latency of at least 3 cycles) to the programmer unfamiliar with the x86 architecture. The situation that occurs when there are fewer available registers than cipher variables, is termed *register pressure.*

Register pressure on the Pentium 4 processor is frequently a major concern to cipher developers. The designers of Helix [73] specifically chose to use a 160-bit state because this maps to five 32-bit registers, allowing the remaining two general purpose registers to be used without the need for expensive stack swapping.

Under pressure from graphics consortiums seeking fast multimedia capabilities, the later version of the Intel Pentium chip introduced Multimedia eXtensions (MMX), which duplexed a series of 64-bit registers on the back of the Floating Processor Unit (FPU), and introduced a limited set of Single Instruction Multiple Data (SIMD) instructions that applied exclusively to those registers. One idiosyncracy of the chip was that floating point and MMX modes could not be used concurrently. The operation that switches between modes has a long latency. To cipher designers, this is not particularly important, since floating point operations are rarely called into use in cipher design. It used to be that some symmetric ciphers performed better when implemented on the floating point registers, because of general purpose register pressure, but as the integer-based MMX uses the same registers as the floating-point unit, this is no longer the case.

The Intel Pentium III introduced Streaming SIMD Extensions (SSE) which applied an extended set of floating-point instructions to the 64-bit MMX registers. The Intel Pentium 4 considerably improved the SIMD capabilities of the family by bringing a wide range of integer operations to an additional set of eight 128-bit SSE2 registers that are independent of the FPU registers. A significant cost is still incurred in transferring data between the general purpose and SSE2 registers, and the number of SSE processing units within the chip core have been halved.

8.1.2 Memory

One of the major issues to affect performance of ciphers in software is the way in which they access computer memory.

The registers in the core of the Pentium 4 are accessible at clock speed (between 1.5 GHz and 3.8 GHz depending upon the chip). The memory from which the registers acquire their data is very much slower; typical memory operates at a frequency of only 133 MHz. Even the system bus that connects the main memory to the CPU may act as a limiting factor; at between 400 MHz and 800 MHz, it has a throughput of between 3.2 and 6.4 Gigabytes/second. Clearly this has the

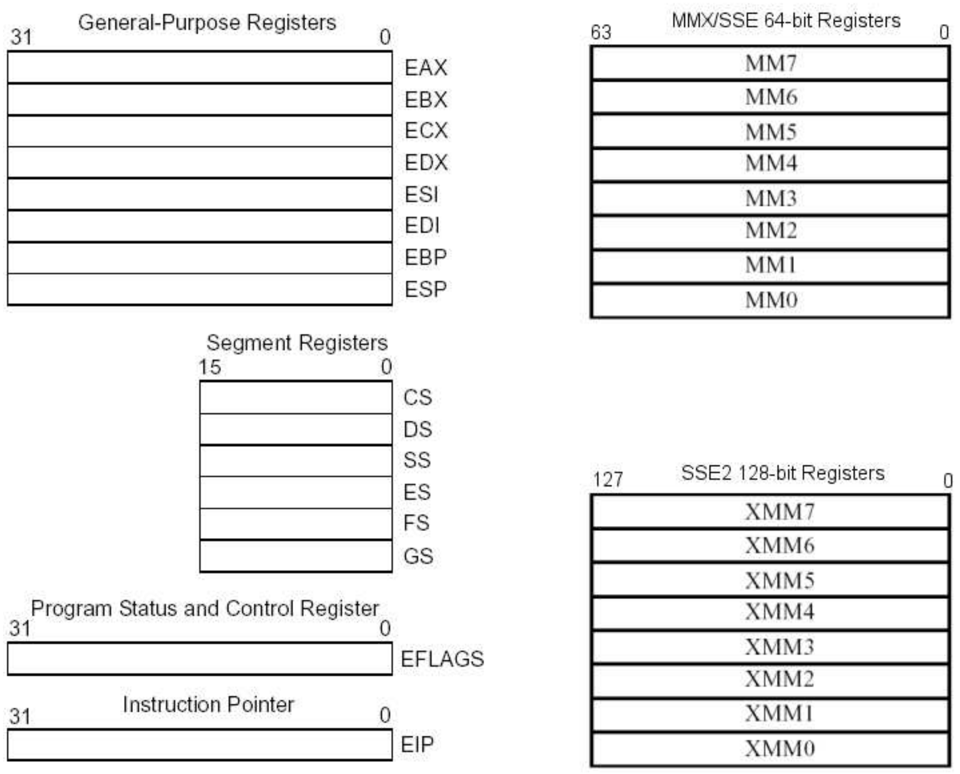

Figure 8.1: Intel Pentium 4 Register Set
[101]

potential to impose an upper limit on the speed of a symmetric cipher, as the data being encrypted comes from main memory.

A series of high-speed caches are positioned between the CPU and the main memory. Data is summoned, on demand, to the caches to reduce the effect of the disparity in speeds of the CPU and memory. When the data is not available, a latency penalty is incurred to bring it to the cache. This may cause delays within the associated application.

The Pentium 4 has two or more caches: the first, the L1 cache, operates at clock speed but has a capacity of eight kilobytes for data. This is half the capacity of the Pentium III L1 cache, but enables smaller latency penalties. For integer operations, the penalty for the L1 cache is only two clock cycles, but for floating-point operations, it is six. The later Prescott version of the chip enlarges the data portion of the L1 cache to sixteen kilobytes; the penalty for this is an integer latency of three cycles.

The L2 cache has a much larger capacity of between 256 kilobytes and two megabytes, but runs at about one-third clock speed, so has the capacity to impose significant performance penalties. The Intel Xeon subfamily offers a slower but larger L3 cache. The cache structure for the Intel Pentium 4 is shown in Figure 8.2.

The caches need to be shared between system and user processes, and in a multitasking environment, there are no guarantees on how much of the cache is available for a particular process. However, the sizes of the caches act as upper bounds. In particular, generating amounts of data in excess of eight kilobytes, such as large s-boxes (for example, those of HC-256 [227]) result in frequent L1 cache misses, and the resultant penalties as the data is brought in from slower memory sources.

Data is loaded into the cache in sixty-four byte chunks (called a *line*). It is loaded at the explicit request of the programmer (in an action called a *pre-fetch*, which uses a member of the **PREFETCHxxx** instruction set), or automatically when the data is required but not available in the cache. Pre-fetching, when managed correctly, can alleviate the delays associated with cache loading. Twelve clock cycles are required for the pre-fetch to reach the system bus and return data to the processor through all of the caches.

The Pentium 4 has a 4-way L1 cache (8-way for the Prescott variant). This means that the cache is grouped into rows, each containing four lines. To allow

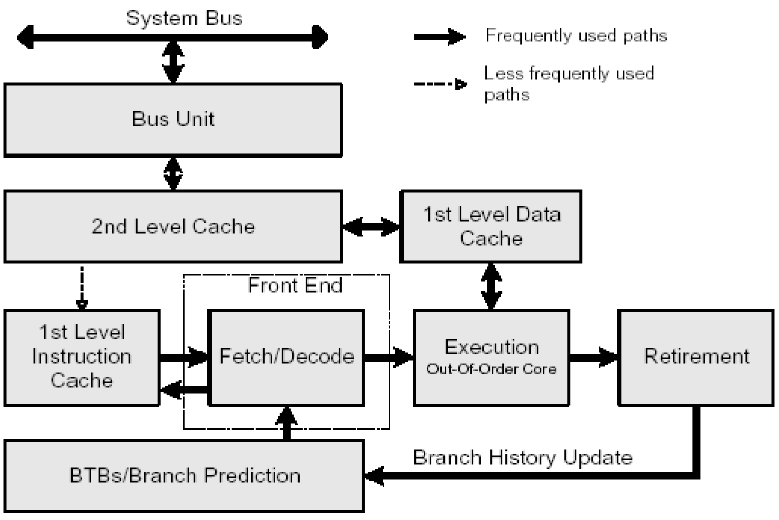

Figure 8.2: Intel Pentium 4 Architecture
[102]

fast retrieval of data from the cache, each row within the cache is associated with a set of addresses in the main memory. If the row is full, and data is loaded from an associated address, a conflict occurs, and a line is displaced to make room for the new data, even if other rows in the cache are empty. Attempting to access the earlier data will result in a cache miss.

A further cache issue concerns misaligned data that cross line boundaries. In these events, two cache loads are required. Most higher level compilers ensure data alignment, but the assembly programmer needs to be particularly aware of this issue.

One of the issues associated with bench-marking ciphers is the kinds and speeds of memory, buses and caches available on a bench-marking machine. For example, an implementer who tests on a Pentium 4 Xeon with an L3 cache and a bus speed of 800 MHz is may have superior metrics to the owner of an older machine even though the processors are the same model. Also, due to the longer pipeline and integer latencies, some small programs may run slower on the newer Prescotts than on earlier models with slightly lower clock speeds.

8.1.3 Execution Pathway

The Intel Pentium 4 has a three-stage instruction execution pathway, beginning with the instruction front-end, followed by the out-of-order execution core, and terminating in the instruction retirement section. This is seen in Figure 8.2, alongside the cache architecture discussed in Section 8.1.2.

Instruction Front-End

The Instruction Front-End fetches assembly program instructions that are to be executed, decodes them into micro-operations and supplies them in order through the instruction portion of the L1 cache to the Instruction Execution core. This holds up to 12,000 micro-operations. The assembly language instructions may already have been decoded from a high-level language by a compiler.

Instruction Execution

The Pentium 4, like all members of the Pentium family, is *super-scalar*, meaning that it has the ability to execute multiple operations in a single clock cycle. This is the job of the Instruction Execution core, which executes the micro-operations supplied by front-end in the fastest possible order. The core has four ports that provide access to its execution units. Each port can be accessed concurrently by the front-end, to complete six micro-operations in one clock cycle. The ports include:

- a port with a fast integer unit and a floating point (FP) move unit. The associated units can dispatch two fast integer operations per cycle; or one fast integer operation and one floating point move or store operation.

- a port with a fast integer unit, a normal integer unit and a floating point execute unit. The associated units can dispatch two fast integer operations per cycle. An integer multiply, shift or rotate, or FP/MMX/SSE operation can be substituted for the second fast integer operation.

- a port that deals with memory loading or pre-fetching. The associated unit can dispatch one operation per second.

- a port that deals with memory storage. The associated unit can dispatch one operation per second.

This means that the cipher designer needs to consider the types and order of operations used within a cipher. Reducing dependencies within a cipher is not necessarily going to create opportunities for parallelization. For example, if two competing operations are multiplications, no gain will be made, as there is only one port available to them. But it is possible to execute an addition on the fast integer port, and a multiplication on the normal integer port concurrently (although, all things being equal, the addition will finish first).

The architecture uses dynamic data flow analysis to execute instructions out-of-order. This reduces delays caused by cache misses, and allows the execution of dependency-free instructions before others that are waiting upon some resource. In some ways, this feature conflicts with one of the central ambitions in symmetric cipher design: to create as many dependencies between variables as possible in order to maximize diffusion.

The CPU uses deep branch prediction, in which it attempts to analyze which path a branch takes, execute the path, and have the results ready by the time the branch expression is evaluated. When the branch is mis-predicted, the results are discarded and as many as thirty-one cycles of the CPU time are wasted, depending upon the length of the execution pipe-line, and where in the pipe-line the branch is situated. Unfortunately, it is difficult to utilize branch prediction in symmetric cipher design, since it performs poorly with random data; when the result of the branch expression is random, the prediction is on average correct only fifty percent of the time. Cipher designers should strive to minimize the number of branches in their code.

One effect of the strategies employed by the Instruction Execution Core is that the time taken to complete an instruction is not always predictable. However, a lower bound can be calculated.

Instruction Retirement

The Instruction Retirement unit takes the completed micro-operations from the Instruction Execution Core, and retires them in the correct order to maintain program correctness. It also sends updated data to the Branch Target Buffer (seen in Figure 8.2), which generates the branch prediction information.

Although the instruction core can execute six micro-operations in parallel, the instruction retirement unit retire threes operations in one clock cycle; this is a limiting factor on the CPU throughput, and should be considered by cipher designers when they are estimating the throughputs of their ciphers on paper.

Dependencies

One of the important rules to note when designing a cipher, or theoretically calculating its throughput is to avoid, where possible, write-read or write-write dependencies. The value of a variable cannot be read or written if a calculation that affects its value is still being undertaken (that is, if the corresponding instruction has not been retired). This has obvious implications on parallelization opportunities. Where two consecutive operations act upon the same variable, in which the first performs a modification, a bottleneck in the code is created.

8.1.4 Streaming SIMD Extensions 2

The streaming SIMD Extensions 2 (SSE2) are a new feature introduced with the Pentium 4. They extend the Single Instruction Multiple Data (SIMD) technology first introduced with the Pentium processor in the form of Multimedia Media Extensions (MMX) and evolved in later processors.

SSE2 is associated with eight 128-bit XMM registers that can be accessed from the general purpose registers and vice versa indirectly via memory. Each register can be treated as a single 128-bit register, two 64-bit registers, four 32-bit registers, eight 16-bit registers or sixteen 8-bit registers. A set of 144 instructions can be applied to each of the registers. This set largely resembles the instruction set for the general purpose registers, and includes many instructions that are useful to the implementation of symmetric ciphers, but which may be faster when applied in the SSE2 mode. The Intel Pentium 4 Instruction Set Manual [102] cites RC5 as an application benefited by the use of SSE2.

Most of the SSE2 integer operations have a throughput and latency of two cycles, which put them on an even footing with the fast integer operations on general purpose registers, which operate on 32-bits and execute in half a cycle. However, the XMM registers are not capable of indirectly addressing memory (for example, to implement s-boxes). The **MOVDQU** instruction is used to transfer memory to the XMM registers and costs six cycles, although a new instruction can be scheduled on each cycle. This makes operating in SSE2 mode less flexible than operating on the general purpose registers.

SSE2 also lacks branching instructions. The data in the XMM registers can be masked to generate a branch indicator, which needs to be transferred to general purpose registers, on which the conventional branching instructions can be applied.

An additional eight 64-bit MMX registers are available for use in conjunction with the XMM registers and are directly accessible using the costly **MOVDQ2Q** instruction, which requires a latency of eight cycles and a throughput of two cycles. Considering the cost of the **MOVDQ2Q** operation, it is unlikely that the cipher designer will make use of the MMX registers. Between the general purpose registers and the XMM registers, the available fifteen registers should be sufficient for any canny designer to produce a secure and fast cipher. An additional one-off cost is required to tidy the MMX registers when the application is complete, due to the **EMMS** instruction, which costs twelve cycles.

A treatment of the relevant SSE2 instructions to symmetric cipher implementation can be found in Section 8.3. SSE3 instructions were introduced with the Prescott variant of the processor, but appear to have no bearing on symmetric cipher development.

8.2 General Optimization Rules

Cryptography and compiler technology are the last bastions of assembly language. In almost every field, compilers are smarter than the average programmer in squeezing extra clock cycles out of increasingly complex code. However, symmetric ciphers are small and simple applications, and a deep knowledge of the underlying architecture can put the programmer at an advantage over the compiler. At the very least, every programmer concerned with optimization should be able to read assembly code generated by the compiler.

There are general optimizing techniques that benefit the programmer of the higher level language. Almost all of the simple high-level rules proposed for cipher design by [206] in 1997 still hold, even though the architecture of the Intel chips has changed substantially.

The most important rule in optimizing cipher performance is consideration of algorithmic optimization prior to implementation optimization. A good compiler matches the program to the underlying architecture and inserts useful implementation shortcuts, but there is little it can do with a poorly designed algorithm. A block cipher that iterates sixty hefty rounds will always be slow; likewise, a stream cipher in which complex and widespread dependencies cripple opportunities for parallelism, will never reach target metrics irrespective of the compiler that is used.

8.2.1 Know Your Compiler

In [77], Fuller et al. describe important techniques such as loop unrolling and strength reduction, and proceed to empirically demonstrate how such techniques improve the speed of simple cryptographic algorithms. But the authors do not describe the limitations on these techniques, or indicate performance penalties caused by over-use of these techniques. More importantly, they neglect to mention that most compilers implicitly support these techniques. In fact it is easy to improve upon the metrics provided by [77] as they omit examination of specific processor techniques.

One of the most common C compilers, **gcc** supports multiple levels of optimization which can be invoked on the command line. Actually most compilers support similar levels, which can be viewed with a cursory inspection of the relevant software.

```
gcc dragon.c
```

supports the default level of optimization, which is none, so as not to interfere with debugging tasks, particularly at the assembly level. Optimization does strange things to assembly code, such as destroying direct mappings between variables and registers, which are invaluable during the debugging process.

However,

```
gcc -Ox dragon.c
```

where **-Ox** represents **-O0**, **-O1**, **-O2**, **-O3**, or **-Os**. The first four of these options implement increasing levels of optimization in terms of speed, while **-Os** optimizes according to size.

At the default level of optimization, the **EBP** register is reserved exclusively for debugging, which means that only six general purpose registers are available (assuming that **ESP** is also used only for stack operations). One of the most interesting and important optimizations made at the **-O1** and all subsequent levels, is the

```
omit-frame-pointer
```

directive, which on the Intel Pentium 4, frees the **EBP** register for use in the compilation.

The **-O2** level supports all optimizations made by **-O1** in addition to all optimizations that do not involve a time-space trade-off, including strength reduction (as reviewed in [77]) and automatic alignment (see Section 8.1.2).

The **-O3** level supports almost all optimizations, including function in-lining (see Section 8.2.3) and register renaming (which utilizes an advanced feature of the Intel Pentium 4 architecture related to pipelining).

The **-Os** level optimizes for size rather than speed. This can benefit symmetric ciphers, given that this trade-off may make the difference between the cipher fitting into the L1 cache or otherwise. This level disables alignment-related optimizations, reordering of program blocks to minimize branching and automatic pre-fetching instructions.

For optimal performance, the programmer may wish to mix-and-match optimization flags rather than use a generic **-Ox** switch.

There are additional optimization flags that are not covered by the **-O3** switch. One example is the switch **unroll-loops** that unrolls all loops where the number of iterations is known at compile time, or **unroll-all-loops**, for loops where the number of iterations is not known. However, the latter option can frequently make programs run more slowly.

Switches specified later on the command line override those specified earlier. So, for example, to get the most speed, the programmer may wish to use **-Os** including the disabling of the alignment, but with branching minimized, and pre-fetching enabled, and basic loop unrolling. This is actualized as

```
gcc -Os -freorder-blocks -fprefetch-loop-arrays -funroll-loops
   dragon.c
```

It is important to let the compiler know the architecture for which it is compiling. This allows it to take maximum advantages of the underlying instruction set, optimal alignment, etc . For **gcc**, this is done by supplying **-march** and **-cpu** flags; in the case of the Pentium 4 processor, it is simply a case of compiling with

```
gcc -march=pentium4 -mcpu=pentium4 -O2 dragon.c
```

8.2.2 Loop Unrolling

Loop unrolling is one of the most common optimizations and is generally handled well by compilers. However, there are some cases where compilers fail to unroll loops appropriately; also assembly-level programmers need to be aware of the benefits and limitations of loop unrolling.

When a loop is fully unrolled, the overhead caused by the loop construct disappears, and frees the register that would otherwise hold the indexing variable. This can be extremely useful for ciphers that are otherwise using only five or six variables in the locality of the loop. One example involves unrolling the loop that iterates rounds within a block cipher. For example:

```
for (int i = 0; i < 4; i++) {
   block_round(plaintext[i], ciphertext[i], k[i]);
}
```

becomes

```
   block_round(plaintext[0], ciphertext[0], key[0]);
   block_round(plaintext[1], ciphertext[1], key[1]);
   block_round(plaintext[2], ciphertext[2], key[2]);
   block_round(plaintext[3], ciphertext[3], key[3]);
```

which both frees the register containing the variable i and hard-codes the addresses for the pointers to **plaintext**, **ciphertext** and **key**. When an additional variable becomes available, it may free ciphers from register pressure if they conservatively use registers (such as Helix [73]).

The final iteration of the loop always creates a branch mis-prediction and the associated cycle penalty. Loops also hide data dependencies and even create additional dependencies that reduce opportunities to exploit parallelism [79].

However, unrolling loops means that the size of the code increases. This has two effects: excessive unrolling may cause the code not to fit within the L1 cache; also the number of instructions that the processor has to decode increases, and this may cause a decrease in performance.

The Intel Pentium 4 handles loop unrolling better than its predecessors. The optimization guide for the Pentium II and III processors advised against completely unrolling loops of more than four iterations [104]. For the Pentium 4, this is relaxed to loops of more than sixteen iterations for loops that do not contain branches. Unrolling loops that contain branches may cause additional branch prediction penalties if more than $\frac{16}{\#cb}$ loops are unrolled, where $\#cb$ is the number of conditional branches [105].

In stream and block ciphers, where the bodies of loops are generally short, branch mis-prediction penalties and loop overheads may damage performance. In [79], Gerber advises that where the compiler does not adequately unroll the loops (observed in the generated assembly code), manual unrolling is useful in loops with low loop counts or short bodies. Loops with high loop counts and long bodies should only be manually unrolled where data dependencies can be visibly reduced.

Generally the only loop that should be incorporated into a stream cipher is that one that iterates the update function to produce keystream. In a block cipher with fewer than sixteen rounds, a single loop should provide plaintext; in a block cipher with sixteen or greater rounds, an additional loop should be utilized to handle partially unrolled round functions. All other loops should in general be designed out of the cipher.

8.2.3 Inlining

Inlining is a common technique in which the contents of a short procedure are substituted into each procedure call. For example, the following procedure:

```
void function mutex(bool locked) {
    this.lock = locked;
}

mutex(true);
do_something();
mutex(false);
```

when inlined becomes:

```
this.lock = true;
do_something();
this.lock = false;
```

Inlining is commonly implemented using macros in C. C++ discourages the use of macros, but possess an **inline** keyword that implements a similar technique.

Because inlining can have a significant and positive effect on the speed of ciphers, it is not uncommon to see an entire cipher implemented using macros, with only the public interface being implemented as a procedure call.

Each procedure call sets up a stack frame, which involves manipulating the **ESP** and **EBP** registers, shifting out any prior contents, and restoring them to the registers once the stack frame is destroyed. Parameters that are passed to the procedure may be copied into the stack along with the return address. This activity represents a moderate amount of computation, and may actually overshadow the resources required by the body of a short procedure, in addition to increasing register pressure caused by reserving **EBP**.

Inlining avoids this. The disadvantage of inlining is that it increases code size; as with loop unrolling, this increases the number of instructions to decode, and may cause the resulting code not to fit in the L1 cache. Both the compiler and the processor can automatically perform certain amounts of inlining.

8.2.4 Removing Branches

Unpredictable branches in symmetric ciphers fall foul of the Pentium 4's branch prediction capability and cause hefty penalties. The cipher designer should consider omitting unpredictable branches from the algorithm.

Where branches can not be omitted, [105] has four suggestions to improve their performance:

- partially or fully unroll loops, as discussed in Section 8.2.2.
- rearrange code to make basic blocks contiguous when applicable. Symmetric ciphers are concise segments of code, so the use of branches as an organizational device seems unlikely.
- use **CMOV** or **SETcc** instruction to remove unpredictable conditional expressions and replace the contents of their branch statements. **CMOV** performs a conditional move based on a status flag that may represent the result of the branch expression, whereas **SETcc** sets the value of a register based on a status flag that may indicate the result of an operand comparison. Modern compilers may automatically perform this optimization, but the assembly code should be checked manually to ensure its correct implementation.

- for branches with forward targets (if statements and for loops), make the fall-through code (that which is accessed when the branch fails) the most likely target. For branches with backward targets (loops), make the fall-through code the least likely target. This is consistent with the branch-prediction algorithm on the Pentium 4.

8.3 Cryptographic Primitives on the Intel Pentium 4

This section indicates the efficiency of different cryptographic primitives on the Intel Pentium 4. The efficiency of an operation is described in terms of its latency (the number of clock cycles it takes to complete) and its throughput (the minimum number of clock cycles that elapse between the scheduling of two successive instances of an operation).

8.3.1 Additions and Subtractions

Modular ($+$) and binary additions ($\oplus$) and subtractions ($-$) *without carry* are cheap and easy to implement on the Intel Pentium 4. All of the ciphers reviewed in Section 4.1 use at least one of these operations; many use two. These operations are implemented using the **ADD**, **XOR** and **SUB** instructions respectively. Provided that there are no dependencies between arguments, the two integer ports can be used to schedule and complete two of these types of operations in each clock cycle.

Sixty-four-bit additions and subtractions require carry between words, and are implemented using **ADC** and **SBC** respectively. One of these operations can be scheduled on the normal-speed integer port every three cycles. The operation takes eight cycles to complete, but only influences the higher thirty-two bits of the calculation. An additional **ADD** or **SUB** operation is required to complete the operation on the lower bits, but can be scheduled for free on the last cycle.

The SSE2 **PADDD**, **PXOR** and **PSUBD** instructions are executed on the MMX ALU and all have throughputs and latencies of two cycles. Speedwise, the use of SSE2 instructions offers no advantages.

Assembly programmers should note that the Intel Pentium 4 penalizes the use of the increment and decrement instructions **INC** and **DEC** respectively, which on older processors are useful for implementation of counters (for example, the M register of Dragon's keystream generation, or the i index of the RC4 cipher). Although throughput of **INC** and **DEC** is only half a cycle, the latency is double that of an **ADD** or **SUB** operation, which should be used instead.

8.3.2 Logical Operations

Logical operations like AND and OR are treated as fast integer operations and like addition and subtraction, have a latency and throughput of a half cycle.

The SSE2 **PAND**, **POR** and **PNAND** instructions are executed on the MMX ALU and have throughputs and latencies of two cycles. They offer no intrinsic advantages over the conventional general purpose instructions.

8.3.3 Shifts and Rotations

On the Pentium 4, shifts and rotations in the general purpose registers are strongly penalized. Generally they take four cycles to complete, although they may be quicker if the shift/rotate operand has a value of 1. Variable and fixed shifts or rotations have the same performance characteristics. In the Prescott version of the chip, the shifts and rotations can be executed on the fast ALU units, so have latencies and throughputs of 1 cycle.

The SSE2 **PSLLD** instruction simultaneously shifts four 32-bit operands left with a latency and throughput of two cycles. The corresponding right shift instruction is **PSRLD** and bears the same performance characteristics. However, the SSE2 instruction set does not contain a rotate: this needs to be synthesised using two shift instructions, two masking instructions and a single OR instruction, totalling ten cycles on a 128-bit operand. This places it at a slight advantage over the general purpose rotate instruction.

8.3.4 Multiplications

On the Pentium III architecture, the multiplication operation, which has a throughput of one per cycle and a latency of four cycles [104], provides excellent diffusion for such a simple operation. This prompted the designers of block ciphers like RC6 [197] and MARS [42], as well as designers of stream ciphers like Rabbit [34], to adopt the use of native multiplication as a cipher primitive.

However, the performance of the multiplication instruction **IMUL** unexpectedly took a nose dive, when its implementation upon the Pentium 4 chip was shifted to the FPU (the Pentium 4 is a massive chip that fits 55 million transistors and has an area of 145 mm^2). When the Pentium 4 executes a multiplication operation, it shifts the operands from the integer unit to the FPU, and the results back to the integer unit. Consequently the operation has a throughput of five cycles and a latency of fifteen cycles. The Prescott edition of the Intel Pentium 4 chip has a dedicated multiplier. However, the latency remains at ten cycles, meaning that the operation is generally to be avoided in symmetric ciphers.

Ciphers like IDEA [143] use multiplication in the field of integers modulo $2^{16} + 1$. Hence, each multiplication operation had to be coded in software as a sequence of operations, including a native multiplication modulo 2^{16} . For such ciphers, performance due to the multiplication operation is even worse.

The SSE2 **PMULUDQ** instruction simultaneously multiplies two 32-bit multiplicands by their multipliers and stores the resulting 64-bit results. This has a throughput of two cycles and a latency of eight cycles, making it much more efficient than the corresponding general purpose **MUL** instruction.

8.3.5 Permutations

Permutations, such as those used by the Camellia [6] and DES [177] block ciphers, can be implemented using the **MOV** instruction and logical operators where necessary. The **MOV** instruction between registers is a fast integer operation that has a latency and throughput of a half cycle. The speed of movement between registers and memory depends on whether relevant data is held in the L1 cache.

SSE2 includes **PSHUFLW**, **PSHUFHW**, and **PSHUFD** instructions to shuffle sixteen and thirty-two bit quantities within the XMM registers. In terms of latency and throughput, these are on an equal footing per byte with moving between general purpose registers. Both **PSHUFLW** and **PSHUFHW** which operate on 64-bit quantities have throughputs and latencies of two cycles. **PSHUFD** has a throughput of four cycles and a latency of two cycles.

8.3.6 S-box Lookups

S-box lookups are slow.

A single 8×8 s-box lookup $\mathbf{y} = \mathbf{S}(\mathbf{x})$ on a 32-bit machine may take up to four operations, as shown in Figure 8.3. The first copies the source word **x** to an index register (in this case, **EAX**). The second masks out the higher bits in the index. The third copies the address of the s-box to an index register (in this case, **ECX**), and the fourth stores the result of the lookup in a register or an address in memory (in this case, the location of the **y** parameter in the stack).

```
y = sbox(x);

mov eax, DWORD PTR _x$[ebp]
and eax, 255
mov ecx, DWORD PTR _sbox[eax*4]
mov DWORD PTR _y$[ebp], ecx
```

Figure 8.3: Assembly Code for 8×8 S-box Lookup

Of course, use of $m \times n$ s-boxes with larger input sizes is impractical, due to the size of the resulting table ($2^m \times n$ bits). More efficient virtual s-boxes can be constructed, as shown in Chapter 7. Performing a lookup on Dragon's virtual 32×32 s-box is much slower than an 8×8 s-box lookup, since the former consists of four 8×32 s-box lookups, in addition to accessing individual bytes in the source word, and combining the results. If the operations are performed in parallel, this requires the use of four index registers, which places pressure on the usage of the Pentium 4's limited register set. The pressure is alleviated in the example shown in Figure 8.4, generated by Visual C++ 6.0, since the addresses of the s-boxes are not loaded into registers, but accessed directly from slower memory. Also **ECX** is reused, which creates a dependency between the processing of BYTE2 and BYTE0, reducing opportunities for parallelism.

S-boxes can be expensive in terms of the precious L1 cache. In [206], the authors suggest limiting total table sizes to four kilobytes for a combined instruction and data cache size of sixteen kilobytes. This rule also holds for the Intel Pentium 4, with its L1 data cache of eight kilobytes.

The block cipher LOKI97 [39] uses two s-boxes, one 13×8 and the other 11×8. Together this amounts to ten kilobytes of memory, so many s-box lookups cause cache misses. If, for a stream cipher such as Dragon, four separate 8×8 s-boxes are used to compose a virtual 8×32 s-box, this means a table consisting of four kilobytes of entries, which begins to place pressure on the L1 cache. This is one reason why Dragon only uses two 8×32 s-boxes totalling only two kilobytes.

```
y = sbox1(BYTE0(x)) ^ sbox2(BYTE1(x)) ^
    sbox1(BYTE2(x)) ^ sbox1(BYTE3(x));

mov eax, DWORD PTR _x$[ebp]
and eax, 255                       ; eax contains BYTE3(x)
mov ecx, DWORD PTR _x$[ebp]
shr ecx, 8
and ecx, 255                       ; ecx contains BYTE2(x)
mov eax, DWORD PTR _sbox1[eax*4]
xor eax, DWORD PTR _sbox2[ecx*4]
mov edx, DWORD PTR _x$[ebp]
shr edx, 16
and edx, 255                       ; edx contains BYTE1(x)
xor eax, DWORD PTR _sbox1[edx*4]
mov ecx, DWORD PTR _x$[ebp]
shr ecx, 24
and ecx, 255                       ; ecx contains BYTE0(x)
xor eax, DWORD PTR _sbox1[ecx*4]
mov DWORD PTR _y$[ebp], eax
```

Figure 8.4: Assembly Code for 32×32 S-box Lookup in Dragon

Note that other operations such as permutations and multiplications can be simulated by s-box or lookup tables. The Advanced Encryption Standard (AES) is optimized in this way on many processors. As noted by [231], the AES performs badly on the Pentium 4 because the large tables do not fit in the L1 cache.

8.4 Implementing the Dragon Stream Cipher on the Intel Pentium 4

Dragon was not specifically designed for the Intel Pentium 4 processor. In the early stages of its conception, the design team were mostly using Intel Pentium III machines, but the Pentium 4 was gaining in popularity and it was clear that it would become a principal architecture for the cipher. As a consequence, generic optimization rules for the Pentium family in general were applied to the design first, but decisions were avoided that would have harmed performance on the Pentium 4.

Design Decisions In one early design for Dragon, its update function used four input variables, and the output filter produced a 32-bit keystream. To increase its performance, the cipher was altered to a produce 64-bit output, and the number of variables increased to six (resulting in a net performance increase of about 50%). Using six variables in the update function is less than optimal in terms of register pressure, and requires some of the variables to be swapped in and out of the stack, since some registers are required for indexing and addressing. However, the use of fewer variables was deemed to have a negative effect on security, and early designs incorporating more variables were discarded due to the expected impact on performance.

An implementation of the Dragon update function is shown in Figure 8.5. It is used in both the keystream generator and the initialization algorithm since the commonality reduces the size of the code (and consequently the number of L1 trace-ops cache misses), and also the number of instruction decodes.

It is easy to observe a number of parallelization opportunities. Any two of the instructions in (1) can be scheduled concurrently on the fast integer ports (likewise in (2), (5) and (6)). The third instruction in (1) can be scheduled with any from (2), although at this stage a dependency is formed between the write on a variable in (1) and the read on the same variable in (2); since all right-hand side expressions in (2) are written to in (1), there is no way to avoid this through a different choice of variables. Later in (2), these dependencies can be partially mitigated by interleaving with processing of (3). Similar dependencies exist between (4) and (5), and (5) and (6). The s-box lookups in (3) and (4) are complex operations which are more amenable to parallelization except that register pressure means that the best performance will come from dealing with one or two lookups at a time. The limitations in parallelism come from dealing equally with all variables in a concise cipher that offers little choice in the way of implementation. In a variant of Dragon with more variables, more opportunities for parallelization would exist, but the gains would be more than offset by the penalties caused by register pressure.

With the exception of the s-box lookups, which are slow, all of the other operations in the Dragon update function are fast and can be scheduled on the fast-integer ports to complete two operations in each cycle. Because these operations are so effective using the general registers, Dragon is not amenable to optimization using SSE2. Because SSE2 has no means of indirectly addressing

```
(1)     b ^= a; d ^= c; f ^= e;
(2)     c += b; e += d; a += f;

(3)     d ^= g1(a); f ^= g2(c); b ^= g3(e);
(4)     a ^= h1(b); c ^= h2(d); e ^= h3(f);

(5)     b += e; d += a; f += c;
(6)     a ^= f; c ^= b; e ^= d;
```

Figure 8.5: Parallelization Opportunities in the Dragon Update Function

memory, the bulk of the s-box lookups needs to be performed in the general purpose registers. It would be beneficial to have an SIMD instruction that could perform a lookup on four 32-bit words in an XMM register. In that case, an SSE2 version of Dragon would outperform the general purpose register version, but the fact that Dragon uses two s-boxes rather than four, within the update function, means that the instruction would probably not be applicable.

The combined data and state of the Dragon cipher was chosen to fit in about four kilobytes. This meant a trade-off between the size of the internal state and the number of 8×32 s-boxes used to compose the virtual 32×32 s-box. The final decision was to choose a state consisting of thirty-two 32-bit words (totalling one kilobyte) and two 8×32 s-boxes (totalling two kilobytes). An alternative, which was deemed less secure, was an internal state of sixteen 32-bit words and four 8×32 s-boxes. Because of the smaller state size, this could be more vulnerable to time-memory-data trade-off or guess and determine attacks.

Timing the Code No two programmers produce the same code from the same cipher specification (with the possible exception of the beautifully simple RC4 [5]). Likewise, it seems that no two profilers time the same cipher code in the same way. Should the cache be warmed by filling it with data before the timing begins, thereby removing latency penalties? Is this cheating, because in the real world, the cache will never be warmed? Or by not warming the cache, are we profiling the cache specifications as well as the algorithm? Should real data be read from the hard-disk (which gives an accurate measurement of the deployment of the cipher, so long as all machines use hard-disks with the same latency and throughput)? Or should random data generated in memory be used instead? How should profilers measure only the cipher application and not other threads or processes running in the same system?

A wide range of methods are used, of which two extremes are mentioned here. The timing method employed by [189] uses the **RDTSC** (read time-stamp counter) assembly instruction to average the speed of encrypting only three data blocks using random data and keys. **RDTSC** is a serializing instruction that acts as a fence around out-of-order execution, and ensures that the right operations are being timed. However, it needs to be executed three times prior to executing the timed code, once to serialize, and twice to determine an overhead offset, which is deducted from the benchmark. It is, to say the least, a messy procedure, but ensures that the only instructions being measured are relevant ones. The approach taken by the authors of [201] is to bulk encrypt one megabyte of random data and to derive the throughput using the C system call **clock()**. While [189] cites this as inaccurate, the call measures only the CPU time used by the algorithm, and the vast quantities of data are more than enough to smooth out any irregularities caused by the operating system. Additionally the precision of **clock()** is in the order of hundreds of microseconds, which is more than sufficient during bulk encryption. The very popular OpenSSL [187] library uses a similar approach to [201] but encrypts with real data, reflecting the commercial focus of that application.

The metrics obtained for Dragon use a similar approach to that taken by [201] and are obtained by encrypting gigabytes of random data. This procedure takes less than one minute because Dragon is fast, but averages out small-scale disturbances that are endemic to multi-threaded processors. Each metric is obtained from five runs of the cipher, discarding outliers, and averaging the results. Random rather than stored data is encrypted.

Results Four sets of Dragon code were generated and the code listings are included in Appendix B. This includes the un-optimized Dragon C Code, hand-optimized C code, optimized assembly code, and optimized assembly code that uses SSE2 instructions and registers. Benchmarks for each implementation are included in Table 8.1. Throughputs for the two sets of C code are given for both implementations generated using **gcc** using no compiler optimization (**-O0**) and with full optimization for speed (**-O3**, **-funroll-loops**).

In the un-optimized C code, the update function and the s-boxes are located in their own procedures, as dictated by the software principle of modularity. Whenever the update function is invoked, or the s-boxes called, the implementa-

Code	Throughput (cycles/byte)	Throughput (Compiler-Optimized) (cycles/byte)
Un-optimized C	34.7	13.2
Optimized C	13.3	7.4
Optimized ASM	6.7	N/A
Optimized SSE2	8.8	N/A

Table 8.1: Metrics for Dragon Code

tion endures the construction and destruction of a stack frame on the procedure stack. This is a process that involves many hidden instructions, including reserving the contents of registers, pushing procedure parameters and the return address onto the stack and unconditionally branching between code areas. When the un-optimized code is compiled using the **-O3** switch, the compiler is able to determine that some registers and parameters don't need to be reserved. This amounts to a significant saving in stack-based overheads, resulting in a reduction of 21.5 cycles per keystream byte.

However, even using the **-O3** switch, the compiler seems unable to determine that the s-box procedures can be inlined in the un-optimized code. This is one advantage of the hand optimized code, in which the update function and s-box are inlined within the cipher algorithm. The other advantage is that manual selection enables the update function inputs to be placed directly in registers. The compiler does a poor job of this, reserving these variables in the stack, which resides in the L1 cache. Use of the **-O3** switch is still able to improve upon the hand-optimized code. It saves 6.1 cycles per byte, by ordering instructions for optimal architecture performance, determining which variables to map to registers, and other small optimizations.

The comparatively poor performance of the SSE2-encoded algorithm is due to two factors: firstly, the dependencies within the cipher make it unable to take advantage of the parallelism offered to 32-bit operands within the larger XMM registers. Secondly, the dominating factor within the update function are the s-boxes, which reference memory locations within the L1 cache. The SSE2 instructions are incapable of referencing memory, so this routine must be performed within the general purpose registers. The cost of transferring operands between the general purpose and the XMM registers negates the advantages of having additional registers at the programmer's disposal.

8.5 Summary

Symmetric cipher design is an area in which there are a lot of successful candidates, fulfilling both security and efficiency requirements.

Consequently, a cipher designer who does not understand at a high-level, the architecture of the machine on which the cipher will be deployed, should expect sub-optimal performance from the cipher. This results in a non-competitive cipher. All that is required to provide better results is some knowledge about the characteristics of the architecture, including the register set, the number of general purpose registers; the memory layout including the size and speed of the caches; the types, throughputs and latencies of available instructions in the processor instruction set; special features of the architecture, such as the Intel Pentium 4's Streaming Single Instruction Multiple Data Extensions; and general optimization tricks such as seizing opportunities for parallelization.

Because of modern compiler technology, implementation optimization is less important than algorithmic optimization. However, the implementer, who may have a different identity to the designer, should at least have a working knowledge of assembly language, to capture any opportunities caused by deficiencies in the compiler optimizer. Additionally, he or she should have an intimate knowledge of the compiler to ensure its correct operation.

This chapter demonstrated some of the design decisions made to ensure the new stream cipher Dragon met its efficiency goals through the awareness of its designers and implementers about the target architecture. It also showed how to achieve a five-fold improvement in the throughput of the cipher, from 34.7 cycles/byte to 6.7 cycles/byte, due to judicious use of the compiler and of assembly language.

Chapter 9

Conclusion and Future Research

This thesis investigated the efficient implementation of secure symmetric ciphers, with particular focus on generating agile and robust key schedules, and designing stream ciphers using efficient block cipher components. In Section 9.1, the contributions of this thesis are reviewed. In Section 9.2, further avenues for research are explored.

9.1 Review of Contributions

Chapter 2 modifies the key schedule classification proposed by Carter et al. [45] to remedy problems that occur when key schedules exhibit polymorphic behaviour, depending upon the number or identities of round keys held by an attacker. As with the original classification, the modification indicates whether a key schedule is a Type 1 (weak) or Type 2 (robust) key schedule. The most important contribution of this classification is the Type 2A category. This category is missing in Carter et al.'s classification, but yields critical information about the vulnerability of the cipher when the attacker holding more than one round key, can with less effort than exhaustive search, identify extra round key or master key bits. The new classification also remedies other problems with Carter et al.'s classification, and can be used as a simple bench-mark of the security of a cipher's key schedule.

This modification was applied, as part of a wider classification, to the block ciphers entered into four international competitions, each with the aim of standardizing a cipher. Of the twenty-nine block ciphers surveyed, fifteen have Type

1 key schedules, emphasizing the need for cipher designers to understand the importance of robust key schedules. One surprising result is that the winner of the Advanced Encryption Standard has a Type 1B key schedule, as do many of its descendants in the NESSIE competition, such as Anubis, Grand-Cru and Noekeon. The categorization has the purpose of clarifying the nature of components used in block ciphers, which is necessary in studying the influence of block ciphers upon stream ciphers.

Chapter 3 verifies the Type 1B classification of the AES key schedule, by demonstrating statistically that it suffers from bit leakage and poor diffusion. It reviews attacks on reduced rounds of the AES that received footholds through its key schedule. The chapter contains a new and efficient key schedule for the AES cipher that reuses the AES round function, exhibiting good diffusion and minimal bit leakage. The performance of the key schedule is contrasted to the AES competition finalists, using a benchmark of the number of blocks each cipher can encrypt in the time it takes to initialize the cipher's key schedule. With the new key schedule, the AES still outperforms all but one of the finalists, which it surpasses in terms of raw throughput. The security of the key schedule was analyzed, and found to be satisfactory. The key schedule was defended against a claimed attack by Wu [226], which uses an unrealistic attack model.

Chapter 4 reviews some recent word-based stream ciphers, indicating how their designs have been influenced by block ciphers, and examining their key agility. Almost all of these ciphers use components from block ciphers, including s-boxes (frequently the AES s-box), PHTs, and even Feistel structures. The update function of some ciphers, such as Scream and Helix, bear strong resemblance to the round functions of block ciphers. Most of these ciphers reuse their update functions in their rekeying strategies. The conclusion is drawn that this strategy in conjunction with a large internal state causes poor key agility. The chapter conducts a literature review of attack methodologies on word-based block ciphers. It finds that Leviathan, LILI-128, RC4, SNOW, SOBER-t16, SOBER-t32 and Scream are vulnerable to distinguishing attacks; that Snow, Hiji-Bij-Bij, Helix are broken by guess-and-determine attacks; that RC4 can be attacked using related keys; and that RC4 and BMGL are vulnerable to time-memory-data attacks. However, it identifies that only one cipher - the self-synchronous cipher Helix - can be attacked using a block cipher methodology. The chapter concludes with a rebuttal of a claim by Miranov [170] that states the first byte of each RC4 keystream is biased. It reiterates the need for a strong key schedule.

Chapter 5 presents an attack against the proposed cellular stream cipher Alpha1. This is a conventional irregularly clocked bit-based stream cipher with a 128-bit state spread over four LFSRs. The attack is a divide and conquer attack, in which two of the registers are broken using a guess-and-determine attack, and another by a correlation attack, using joint probability based upon Levenshtein distances as the correlation measure. Despite the cipher's design strength of 128 bits, this chapter shows how to break it with 2^{61} operations, $2^{29.8}$ memory, and 2^{15} bits of keystream, demonstrating that the algorithm is not suitable for use.

Chapter 6 describes MUGI, which is immune to correlation and divide and conquer attacks because of its large 1,024 bit NLFSR state, and a block-cipher-inspired non-linear filter. However MUGI suffers from key agility problems identified in Chapter 4 because it uses the update function to modify a single stage of its large state at a time. The state is unnecessarily large for its 128-bit key. A variant, MUGI-M, is proposed, for which the rekeying strategy is 200% faster. The accompanying cryptanalysis does not find any problems with the security of the new proposal.

Chapter 7 presents the specification for the Dragon cipher, which has been developed at an optimal trade-off between security and efficiency on 32-bit architectures. It uses a 256-bit master key and 256-bit initialization vector. The cipher uses only operations that are known to be efficient on the Intel Pentium 4, which is one of the most common processors at the time of the writing. The keystream throughput is 6.7 cycles/byte and it rekeys the 128-byte state in 1,395 cycles. Dragon is a secure cipher that is competitive with other modern word-based stream ciphers, and is much faster than block ciphers with equivalent security estimations.

Chapter 8 provides general guidelines for fast cipher designs and implementation, and specific advice for development of ciphers for the Intel Pentium 4. It indicates that it is necessary to understand the architecture of the machine for which the cipher is being designed; that ignorance of these details will present themselves in a cipher that has a mediocre performance. It also reiterates that while algorithmic optimization is more important than implementation optimization, knowing the tools of implementation well pays dividends when the cipher is being profiled.

9.2 Future Directions

In the last two years, following the conclusion of the AES and CRYPTREC competitions, interest in symmetric ciphers has shifted from block ciphers to stream ciphers. This is not surprising; block ciphers are slow compared to the new breed of word-based stream ciphers. Additionally, security in block ciphers has become well understood, whereas word-based stream ciphers are only distant cousins to the bit-based stream ciphers, for which a wealth of theory has been developed. This provides opportunities to better understand the design and cryptanalysis of word-based stream ciphers.

One of the few new block cipher proposals to emerge in the last year is the FOX family [115], which is targeted towards the media distribution industry. This family consists of 64-bit and 128-bit block ciphers, and avoids algebraic s-boxes and weak key schedules. The high level structure is neither SPN nor Feistel, but instead the Lai-Massey scheme, which permits provable immunity against linear and differential cryptanalysis. The cipher is designed with efficiency on a wide range of platforms in mind. The ratio of key setup to encryption is 6:1 and the key schedule appears to be Type 2C. It is pleasing to see a new block cipher designed with an optimal security-efficiency tradeoff in mind, and this family deserves attention in the form of cryptanalysis.

Klimov and Shamir [128] recently proposed T-functions as a replacement to stream ciphers. They claim that T-functions are $n-$to$-n$ mappings where each bit $i \leq n$ of output depends only on bits $0 \dots i$ of the input, and that have guaranteed large periods (for example, $n = 256$ gives a period of 2^{256}). The advantage of T-functions over stream ciphers is ostensibly speed. They claim T-functions may be an order of magnitude faster, although the example that they provide - RC4 - is not particularly fast compared to stream ciphers like Rabbit, Dragon or HC-256. Additionally, it is penalized by lack of parallelization opportunities on the Pentium 4, on which the T-function benchmarks were obtained. Clearly, both the security and performance properties of T-functions comparative to stream ciphers need to be examined further.

The hot topic in stream cipher cryptanalysis is the XL series of algebraic attacks. This requires further investigation because of the lack of empirical evidence – only implementations of attacks on toy ciphers have been produced to date, as the systems of equations grow extremely rapidly with the size of the cipher state. Also, the attack has so far been impotent on word-based stream ciphers, yet as each block of keystream provides many additional equations to add to the over-defined systems, the ciphers should in theory be at least as vulnerable to the attack as bit-based ciphers.

In the recent paper [188], Paul and Preneel reinforce the claim by Mironov [170] that there is a bias in the first byte of RC4 keystream. In the case when the second word of the RC4 table, after key initialization, has a value of two, the first two output bytes will always be different. This contradicts the statistical results of Section 4.3, which failed to detect a statistical bias in the first byte of keystream output. Both an empirical statistical approach using a larger collection of keystream, and a theoretical analysis of the biases within RC4 should be conducted.

Improvements can be made to the MUGI-M variant of MUGI. The non-linear state is 192 bits wide, and is inelegantly keyed using the 128-bit master key. Currently, the third word of the state is filled by combining the two 64-bit key words, each rotated by a small offset. Using the third key-word of a 192-bit master key, to directly populate the state word may increase the security by bringing more entropy into the buffer. Research by Mihaeljevic in [166] indicates that increasing the design strength of MUGI may make it vulnerable to XLS attacks (by the nature of increasing the complexity of the brute-force attack which acts as a measure of success for other attacks). However, it is clear that the estimation of the XLS (or more appropriately the XL attack) needs to be refined for an accurate judgement to be made. Neither MUGI nor MUGI-M have been successfully attacked, and further cryptanalysis needs to be conducted. As stream ciphers targeted towards 64-bit architectures, their value should not under-estimated, because these architectures will become commonplace in the next decade. When this occurs, MUGI and its variants are likely to outperform many of the currently proposed 32-bit block and stream ciphers.

Dragon is a valuable cipher due to its high throughput on 32-bit architectures, but the only cryptanalysis published to date has been conducted by its designers. It is one of the most competitive stream ciphers in terms of keystream generation, key agility and security, and this means that it warrants further analysis.

Implementation issues differ between processors. Although the Intel Pentium 4 is still ubiquitous, other processors will become increasing popular, including 64-bit processors such as the Athlon processor. There is a need to investigate how the performance of block and stream ciphers differ on these architectures, and whether the guidelines for the Intel Pentium still hold on these processors.

Appendix A

Block Cipher Cryptanalysis

It is good practice in cryptanalysis to adhere to Kerchoff's principle, which is the assumption that the cryptanalyst has full knowledge of the algorithm that is being studied. The goals and text requirements of the cryptanalyst vary. The cryptanalyst may be attempting a certificational attack, in which the strength of the cipher is reduced by only a single bit below its design strength, or try to recover all of the plaintext from an encrypted transmission. More formally, a cryptanalytic attack may have one of the following goals [107]:

- distinguisher. This attack distinguishes the ciphertext of a cipher from a random permutation. The footprint of the cipher may be found in patterns in the keystream; the cryptanalyst can use the patterns to identify the algorithm used to generate the ciphertext. Unless the keystream possesses sufficient bias, this kind of attack is frequently certificational in that it is not of any practical use. For example, many protocols already explicitly indicate "in the clear" the algorithm used for encryption in a handshake [106], [187]. For some cryptographers, a certificational attack is enough to diminish confidence in the algorithm, given that the output of a strong cipher is supposed to be indistinguishable from random. Additionally, development of a distinguisher may be the first step along the road to a complete cipher break.

- key recovery. The goal of this attack to find some or all of the key material used in an encryption. The attack may identify the master key, in which case, once collected, decryption of other texts enciphered under the same key becomes trivial. A key recovery attack may be generated from a distinguisher with a sufficiently strong bias. In this process, the distinguisher is applied to all but a small number of final cipher rounds; the attacker anticipates that by correctly guessing key bytes in the remainder of the cipher, and decrypting to the point where the distinguisher occurs, evidence of the bias will be identified. Incorrectly guessing key bytes will not lead to evidence of the bias. A successful key recovery attack will discredit a cipher completely.

- instance deduction. The goal of this type of attack is to find an algorithm that simulates the cipher algorithm, enabling the encryption of some plaintexts into ciphertexts (or the reciprocal decryption), without knowledge of secret key material. This kind of attack is rare.

- global deduction. The goal of this kind of attack to find an algorithm that simulates the cipher algorithm, enabling the encryption of all plaintexts into ciphertexts, without knowledge of secret key material. This is the most difficult goal to achieve in cryptanalysis.

Before a cryptanalytic attack is launched, texts must be collected and analysed for key material. Different attacks have different text requirements. Each attack may be categorised (in decreasing order of practicality) as:

- ciphertext-only attack. In this attack, the cryptanalyst possesses encrypted text. This is easily acquired, for example, through the use of a network sniffer. Most cryptanalysts assume more demanding requirements in an attack, since only very simple ciphers can be broken using only ciphertexts.

- known-plaintext attack. In addition to ciphertexts, the cryptanalyst knows at least some of the corresponding plaintext. The goals in this attack are to acquire more of the plaintext, or some of the secret master key. Direct observation of the plaintext is not necessary; if the type of encrypted material is known, then the attacker may be able to guess some of the plaintext (for example, a greeting or a date). The combination of the guessed plaintext and the corresponding ciphertext is known as a *crib*.

- chosen-plaintext attack. In this attack, the cryptanalyst is able to use the encryption device (with a fixed but unknown key) to produce ciphertext from the plaintext of choice. This attack style is feasible when the attacker is able to coerce the owner of the key to encrypt some material, or when the attacker is able to use a device with the unknown key embedded within it.

- chosen-ciphertext attack. In this attack, the cryptanalyst is able to choose ciphertext to be decrypted. The means by which this occurs may be similar to in a chosen-plaintext style.

- chosen-key attack. In this attack, the cryptanalyst knows of a relationship between keys used to encrypt material, and has observed ciphertext and possibly some plaintext. Some cryptographers believe this attack is impractical [61], or that it is only possible in conjunction with a protocol flaw that permits unacceptable manipulation of keys.

- adaptive chosen-plaintext or ciphertext. In this attack, the cryptanalyst is able to modify the plaintext for encryption or ciphertext for decryption based upon previous results. In most cases this is a highly theoretical attack.

The success of an attack is measured using the number of texts and operations (and sometimes memory) required to derive a certain number of key bits. Most of the attacks described in subsequent sections are more of theoretical than practical interest for two reasons: the limited distributed computing power around the world makes implementation of attacks with complexity greater than 2^{64} problematic; and many protocols demand the cycling of keys after a specific amount of ciphertext has been generated, which means that a cryptanalyst may find it difficult to amass the required text generated under a single key.

When the number of operations in an attack is less than 2^r where r is the design strength of the cipher, the cipher is considered broken. However, confidence may remain in the cipher, unless the break is significant. Irrespective of the design strength of the cipher, a break with complexity of 2^{112} operations is beyond the reach of implementation, and remains of interest theoretically. A break with complexity of 2^{64} operations may be implemented.

A.1 Basic Attacks

In the following section, generic attacks that apply to all block ciphers are described. These include exhaustive search and dictionary attacks.

Exhaustive Search

The exhaustive search attack is the most rudimentary style of practical cryptanalysis. Also known as the brute-force attack, it is a known-plaintext attack in which the attacker acquires a group of plaintexts and corresponding ciphertexts, and the algorithm known to have encrypted them. The attacker methodically guesses keys, which are applied to the algorithm, using a single plaintext, until the correct ciphertext results. The other plaintext-ciphertext pairs in the group can be used to verify the key guess. For a cipher with a key size of k, it is expected that the average complexity of the attack is 2^{k-1}.

If sufficient redundancy is present in the ciphertext, the brute force attack may be launched as a ciphertext-only attack. In this form, keys are successively guessed, until the decrypted ciphertext possess some meaning. For example, if the plaintext is written in English, guessing a key that was not used for decryption will not produce a meaningful passage of Hamlet. However, guessing the right key will produce an obviously successful attack when the rendered plaintext appears legible.

Whether the attack is launched as known-plaintext or ciphertext-only, brute-force attacks are rarely practical, but serve as a base-line against which other attacks are measured. In the case of 56-bit key ciphers like the Data Encryption Standard (DES), throwing sufficient computer power on a distributed network will succeed in a matter of hours or days [70]. However, against key lengths used in contemporary ciphers – 128 bits of more – brute force cryptanalysis is completely infeasible. Moore's law dictates that computing power doubles every three years, effectively removing one bit from the size of the key. This year, an organization hoping to launch a brute-force attack on a single encryption produced by the AES, will require 2^{70} of Intel's best processors. This is the equivalent of 250 billion Pentium 4 processors for each person on the Earth.

Against modern ciphers – those with 128-bit master keys– brute-force attacks are not practical. They are useful as benchmarks against which the success of other attacks can be measured.

Dictionary Attacks

All block ciphers are theoretically vulnerable to dictionary attacks. For this kind of attack, it is the size of the cipher block that is important, rather than the size of the cipher's master key. In its simplest form, the dictionary attack involves a cryptanalyst collecting blocks of ciphertexts and analyzing their frequencies for patterns [27]. The attack that follows this step is tailored to the resulting statistics.

In a more sophisticated form, the attacker makes collections of plaintexts and ciphertexts related under a single key, indexed on the value of the ciphertext block. This is the attacker's dictionary, which is valid only for a single master key. While the key continues to be used, any ciphertext observed by the attacker can be decrypted with complexity O(1) as long as it is available within the dictionary.

The birthday paradox indicates that, for a cipher with a block size of n, approximately $2^{\frac{n}{2}}$ ciphertexts need to be collected for the attack to commence, assuming that the plaintexts being encrypted are random.

There are three solutions that are guaranteed to make dictionary attacks impractical. The most obvious is to increase the block size of the encrypting cipher. This is the principle motivator for the stipulation that candidates in the Advanced Encryption Standard competition possess 128-bit blocks. For DES and other 64-bit ciphers, the attack is practical since modern hard disks can store around 2^{40} bits, more than the 2^{32} bits required by the birthday paradox. However, for block ciphers with a block size of 128 bits, it will be a long time until technology is ready to cope with the extremely heavy-duty storage requirements.

Chaining modes of operation such as CBC introduce dependencies between ciphertexts, so that no ciphertext depends only upon a single plaintext.

Another solution is to change the master key of the cipher after every $2^{\frac{n}{2}}$ bytes are encrypted. This is a common practice in protocols such as SSL [187] and IPsec [106].

A.2 Differential Cryptanalysis

Differential cryptanalysis exploits the fact that the difference between inputs to non-linear components does not always provide a uniform distribution of output differences. It is arguably the most powerful method of cryptanalysis, given that it is a generic method, and a large number of ciphers have exhibited vulnerability. It was the first key-recovery technique [26] to attack the DES with greater success than the brute force attack (in terms of the number of required operations).

The core tool of differential cryptanalysis is the *difference distribution table* (DDT). This maps how differences between inputs to non-linear components evolve to differences between the outputs. For a component with m inputs and n outputs, the DDT contains 2^m rows and 2^n columns. Each of the 2^{m+n} cells in the DDT contains a value between 0 and 2^n. The cell $(\Delta I, \Delta S(I))$ counts the number of times that pairs of inputs with difference ΔI produce pairs with the output difference $\Delta S(I)$. Without knowing the input pairs, so long as the difference between them is known, the output difference can be assigned probabilistically.

This is useful because: either the first operation in the round is a key addition; or the first round in a block cipher is preceded by key whitening. In each case, knowledge of the individual input values are lost following the key mixing. However, if the key mixing operator is invertible, it does not change the input difference. For two inputs X and X' with input difference ΔX, the key addition operation causes the output difference $(X \oplus K) \oplus (X' \oplus K) = (X \oplus X') \oplus (K \oplus K) = (X \oplus X') = \Delta X$.

By constructing a DDT for each non-linear component in the round function, a probability model of the round function, and ultimately the cipher, can be constructed. This is possible because linear components in the round function do not disrupt differences; that is, for linear component L, $L(I) \oplus L(I') = \Delta I$. Operators other than $\oplus$ can be used to provide the measure of difference; these include the $\boxplus$ operator, and under constrained circumstances, even modular multiplication [35].

The extension of the component mapping to a round is called a *characteristic*. The characteristic $\alpha \rightarrow \beta$ (with p) predicts that for two inputs to round i, P_i and P'_i, chosen such that $P_i \oplus P'_i = \alpha$, then with probability p, the difference between the two output texts of the round C_i and C'_i will be β.

In practice, attacks are launched using multiple round characteristics - it is necessary for a characteristic to approximate the number of rounds employed by the cipher it is attacking. One useful technique for creating large characteristics is to compose smaller characteristics. For example, $\alpha \rightarrow \gamma$ (with p) can be joined to $\gamma \rightarrow \beta$ (with q) to form $\alpha \rightarrow \gamma \rightarrow \beta$ with probability $p \times q$, assuming independence of the characteristics. Characteristics which self-iterate (ie. $\alpha \rightarrow \alpha$) are particularly useful [26]. As characteristics cover an increasing number of rounds, their probabilities are diminished by the non-linear components within the rounds. So long as the probability of the multi-round characteristic exceeds 2^{-m} where m is the number of bits in the block size, it is useful in applying an attack on (at least a reduced round) version of the cipher. Typically, the characteristic covers all but the last round of the cipher. Knowledge of the characteristic difference β, the difference between the ciphertexts $\Delta C = C \oplus C'$, and the values of the ciphertexts lead to a description of the round in which the round key is the only unknown.

An attack is implemented by collecting chosen-plaintext pairs, and *filtering* out those that do not meet the input differences of the characteristic (these are called *wrong pairs*). The attack proceeds from the ciphertext end - the attacker guesses key bits in those rounds not covered by the characteristic, and decrypts to the β end of the characteristic. For a characteristic of probability one, upon the correct guess of the key bits, the expected β differences can be observed. Unless the probability of the characteristic is one, some ciphertext pairs misrepresent the key bits. By collecting and analyzing sufficient pairs, the right key bit values become evident as the frequency of their representation towers over that of wrong key bit values (which are represented by a random distribution). Once the last round's round key is deduced, the round can be *peeled* from the cipher, and the attack can recommence on the reduced version.

The concept of the characteristic is subsumed by that of a *differential*, in which the plaintext and ciphertext differences are still considered, but in which the intermediate stages (the inputs and outputs of sub-characteristics) are abandoned. Conceptually, a differential contains multiple characteristics, and usually has a slightly higher probability [144].

The number of ciphertexts required to launch a chosen-plaintext attack is suggested by the signal-to-noise ratio: $S/N = \frac{|K| \times p}{\gamma \times \lambda}$, where K is the size of the key-space, p is the probability of the differential, γ is the number of keys suggested by each pair of plaintexts, and λ is the ratio of non-filtered pairs to all pairs. In [133], Knudsen states that the signal-to-noise ratio must be greater (hopefully, significantly) than one for the attack to succeed; that if the ratio is less than one, then the right key bits cannot be distinguished. Biham et. al [17] debunk this theory using impossible differentials. Generally for a differential with probability p, the text requirements for the attacks are p^{-1}. When p is smaller than 2^{-n}, the attack cannot succeed.

Some attacks involve the use of multiple, simultaneous differentials. This would appear to proportionally increase the number of ciphertexts required. However, Biham and Shamir [26] devised the method of using *structures* to minimize the plaintext count. Where two characteristics are to be met, a structure called a *quartet* is used (for three characteristics, an *octet* is used and so on.) A quartet that meets characteristics A ($\alpha \rightarrow \beta$) and B ($\gamma \rightarrow \delta$) consists of four plaintexts (p_1 = P, p_2 = P $\oplus$ α, p_3 = P $\oplus$ γ, p_4 = P $\oplus$ $\gamma \oplus$ α). Two plaintext pairs meet characteristic A (p_1 and p_2, and p_3 and p_4), and another two meet characteristic B (p_1 and p_3, and p_2 and p_4). This effectively gives the second characteristic at no extra cost (in terms of ciphertext pairs).

Because of the potency of differential cryptanalysis, there has been a lot of interest in methods to proof block ciphers against differential cryptanalysis.

One of the most obvious ways is reducing the probability of each one-round characteristic. This can be achieved by diminishing the counts of high-frequency output differences in the DDT [184] (the optimal case being a *differentially uniform* non-linear component, in which the probability of each of p output differences is $\frac{1}{p}$). Increasing the output size of non-linear components reduces their differential probabilities, but increases their susceptibility to linear cryptanalysis, and does not always succeed [17]. Some ciphers have been immunized against conventional differential cryptanalysis, using *bent* or *near-perfect non-linear* functions [185].

Immunity may be achieved by impregnating the round with operations that differential cryptanalysis does not model well. One commonly perceived example is the modular multiplication operation, which has the added bonus of performing high-quality diffusion. However, as shown in [35], multiplicative differentials can be developed which, in certain circumstances, approximate exclusive-or. For example, $-x \bmod 2^l = x \oplus 11\ldots10$ which holds for some l when x is odd.

One particularly elegant strategy, made popular by the Advanced Encryption Standard (AES), and adopted by ciphers frequently thereafter, is the Wide-Trail Strategy [62]. The aim of this strategy is to maximize the number of active s-boxes (that have an non-zero input difference) in a small number of rounds, using high diffusion components such as maximum distance separable matrices (MDS). In AES, the MixColumn MDS has a branch number of five, meaning that the number of its input and output bytes containing non-zero input differences is not less than five. The non-zero output bytes of the MDS activate s-boxes in the next round. Consequently, AES provides a guarantee that in four consecutive rounds, there are a minimum of twenty-five active s-boxes. Since each s-box has a single maximum characteristic probability of 2^{-6}, this means that the highest probability for a four round characteristic is 2^{-150}. So there are not enough texts in the AES codebook to allow the characteristic to succeed. The strategy has proven so popular that it has even been adopted by some stream ciphers [224].

The design criteria of all modern block ciphers should include optimal resistance against differential cryptanalysis. Mistakes can be made; although the cipher purported immunity to differential cryptanalysis, a fifteen-round characteristic was discovered in LOKI97, with a probability of 2^{-56} [136]. This enabled an attack on the full sixteen rounds of the cipher, reducing its effective keyspace from 2^{128} to 2^{56}.

Truncated Differential Cryptanalysis

A differential $\alpha \to \beta$ predicts all of the key bits used in studied components in the last round of a cipher. In some cases where the cipher is immune to conventional differential cryptanalysis, it is still possible to deduce some of the key bits, using a *truncated differential*. A truncated differential has the form $\alpha' \to \beta'$ where α' is a subsequence of the bits of α, and β' a subsequence of the bits of β [133].

Attacks using truncated differentials are similar to those for full differentials, and involve the collection and analysis of chosen-plaintexts.

In addition to increased applicability, truncated differentials offer two further advantages over full differentials. Firstly, probabilities are frequently higher, since they need to approximate fewer non-linear components. Secondly, it is easier to form structures (which reduce plaintext requirements) because of the extra freedom allowed by the reduced text specifications [138]. Despite the partial mapping between differences in ciphertexts and plaintexts, use of truncated differentials allows recovery of all the key bits in the last cipher round key, albeit at the expense of increased complexity [26], [133].

Recent ciphers have large block sizes, which exponentially increases the search time for bit-based truncated differentials. Additionally, as these ciphers tend to use byte- or word-based operations, it has become standard practice to search for byte- rather than bit-based differentials. For the round of a block cipher with $m \times n-$bit words, an input difference $(\Delta x_1, \Delta x_2, \ldots, \Delta x_m)$, and an output difference $(\Delta y_1, \Delta y_2, \ldots, \Delta y_m)$, a word-based truncated differential is defined as: $\delta x = (\delta x_1, \delta x_2, \ldots, \delta x_m) \to \delta y = (\delta y_1, \delta y_2, \ldots, \delta y_m)$ with probability p where $(\delta x, \delta y) \in (GF(2)^n)^m$ and $\delta x = \chi(\Delta x), \delta y = \chi(\Delta y)$. The $\chi(x)$ function evaluates to 0 if $\delta x = 0$, otherwise it evaluates to 1. In these differentials, all non-zero differences are considered to be equivalent.

The complexities of some successful differential attacks on modern ciphers are shown in Table A.1. In this table, the abbreviation *CP* denotes a chosen-plaintext attack.

Cipher	Key Size	Rounds	Complexity	
			Data	Time
Camellia [218]	All	10	2^{112} CP	2^{112}
E2 [161]	All	8	2^{107} CP	2^{128}
FROG [222]	All	All	2^{58} CP	2^{56}
LOKI97 [136]	All	All	2^{56} CP	2^{56}
Q [25]	All	All	2^{125} CP	2^{128}
SC2000 [229]	All	4.5	2^{104} CP	

Table A.1: Differential Attacks Against Recent Ciphers

Higher-order Differential Cryptanalysis

A conventional characteristic or differential, as discussed in the previous section can be expressed as having inputs R_i and R_i', with difference α that map through round function f to outputs R_o and R_o' with difference β; that is, it can be modelled as a first-order derivative $\triangle_\alpha f(R_i) = f(R_i+\alpha) - f(R_i)$ with probability $P(\triangle R_o = \beta | \triangle R_i = \alpha) = P(\triangle_\alpha f = \beta)$, given that R_i is uniformly random [142].

An obvious extension to the idea is to model the i^{th} order derivative $\triangle^{(i)}_{\alpha_1,\ldots,\alpha_i} f(R_i) = \triangle_{\alpha_i}(\triangle^{(i-1)}_{\alpha_1,\ldots,\alpha_{i-1}} f(R_i))$ where $\alpha_1, \ldots, \alpha_i$ are all linearly independent. In this case, the second order derivative is $\triangle_\beta \triangle_\alpha f(R_i) = f(R_i + \alpha + \beta) - f(R_i + \alpha) - f(R_i + \beta) + f(R_i)$.

The non-linearity of a non-constant i^{th} derivative is at least i degrees less than that of a polynomial approximating the round function. Practically, high-probability i^{th} order differentials can be used simultaneously with 2^i ciphertexts to recover key bits.

Jakobsen and Knudsen [107] claim that for a cipher in which the output bits of the second-to-last round can be approximated by a polynomial of degree d, and with a last round key of b bits, that a d^{th} order attack can be made with complexity 2^{b+d} and 2^{d+1} chosen plaintexts. The methodology of the chosen-plaintext attack is the same for first order differentials.

Theoretically, a cipher can be immunized against higher-order differential cryptanalysis as for conventional methods [142]. In practice, by using highly non-linear components, few ciphers succumb to higher-order differentials, although one such differential was developed for ten rounds out of eighteen of cipher Camellia in [118]. Also, boomerang attacks and rectangle attacks (described later in this section) are a specialized case of higher-order differentials that apply to heterogenous ciphers in which different round groups exhibit varying levels of diffusion.

Miss in the Middle Attacks and Impossible Differentials

The goal of all of the previous variants of differential cryptanalysis is to: identify a high-probability differential; apply it to all but the last round of the cipher; use the known outputs, and the high probability inputs to the last round to identify probable key bits. However, miss in the middle attacks are based on the reverse approach - they identify key material that can never occur. They sieve this material, to suggest a smaller pool of potentially correct key candidates [17].

A miss in the middle attack is constructed by identifying two differentials of probability one. The two differentials should never simultaneously hold, concatenating to form a *impossible differential* of probability 0. The attack uses chosen plaintexts that match the input to the impossible differential. The corresponding ciphertext pairs are obtained and decrypted, using guessed keys, to meet the output of the differential. If the differential holds using a guessed key, the key is incorrect and can be discarded. It is shown in [17] that an impossible differential is sufficient to uniquely identify the correct key.

The miss in the middle attack is applicable to ciphers which are provably secure to conventional differential cryptanalysis. In particular, it demonstrates that $m \times n$ non-linear components with larger output spaces do not necessarily offer better security. When m is significantly smaller than n, there are many values of n to which m does not map, potentially leading to one-round impossible differentials which can be concatenated. In [18], an impossible differential in CAST-256 is identified, and the large s-box of Twofish mentioned as a candidate in which they could be detected.

There is at least one 5-round impossible differential $(0, \alpha) \mapsto (\alpha, 0)$, $(\alpha \neq 0)$ in any Feistel structure with bijective round functions. This is used to launch an attack on 6-round DEAL [134]. It is also used to attack DFC, but the attack is executed using chosen ciphertexts rather than plaintexts. The reason for this is that, according to our classification, DFC has a Type 2B key schedule - the first round key contains only half the entropy of the master key, and is easier to attack than the last round key. These attacks are shown in Table A.2, in which CP denotes chosen plaintext, and CC chosen ciphertext.

Cipher	Key Size	Rounds	Complexity	
			Data	Time
Camellia [218]	All	9	Unknown	
DEAL [134]	All	6	2^{70} CP	2^{121}
DFC [135]	All	6	2^{70} CC	2^{126}
Zodiac [98]	All	All	$2^{103.6}$ CP	2^{119}

Table A.2: Impossible Differentials Against Recent Ciphers

Boomerang and Rectangle attacks

The boomerang attack [221] is an adaptive chosen-plaintext chosen-ciphertext attack that shares some similarities with higher-order differential cryptanalysis.

It attacks block ciphers that exhibit poor first-order characteristics, but that can be conceptually split into two segments, such that each contains a short, high-probability characteristic. The boomerang attack glues these characteristics together with a second-order characteristic. The second-order characteristic is satisfied using chosen-plaintext and chosen-ciphertext requests, via a quartet structure that simultaneously holds for the first-order characteristic from each segment. The result simulates a high-probability characteristic that extends across segments, and attacks ciphers that are provably immune to basic differential cryptanalysis. The attack is particularly effective for ciphers in which diffusion is stronger in the direction of output to input in the first segment, and from input to output in the second.

Consider a cipher $E = E_1 \circ E_0$, with characteristics of $\alpha \rightarrow \beta$ with probability p for E_0 and $\gamma \rightarrow \delta$ with probability q for E_1^{-1}. The text requirements and complexity of the attack are related to the probability $(p \times q)^2$. If this probability exceeds the probability of a conventional differential across the cipher, it may be more effective to launch a boomerang attack.

To construct a boomerang quartet, the attacker adaptively requests a right plaintext pair (P, P') with the difference α. After encryption across E_0, the corresponding pair (A, A') has the difference β. Since there is no characteristic for E_1 with input difference β, the difference between (C, C') is not predicted.

So the attacker constructs new ciphertext pairs $(C, D = C \oplus \delta)$ and $(C', D' = C' \oplus \delta)$ and requests the decryption of these pairs across E_1^{-1}. The difference between the pairs (A, B) and (A', B') according to the characteristic is γ. Since $A \oplus A' = \beta$, $A \oplus B = \gamma$ and $A' \oplus B' = \gamma$ then $B \oplus B' = \beta$. Thus the attacker

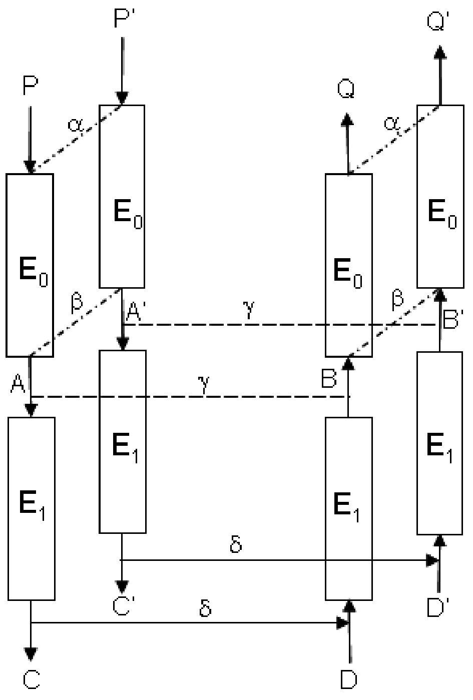

Figure A.1: Constructing a Boomerang

has a right pair for decryption across E_0 such that $Q \oplus Q' = \alpha$. See Figure A.1. In this figure, full arrows represent differences chosen by the attacker; partial arrows represent differences fulfilled by characteristics. By checking that this relationship holds for a sufficient number of plaintexts, the attacker can mount a distinguishing attack. This can be turned into a key recovery attack by guessing the key bits in the first and last rounds and checking that the quartet holds. If the quartet holds, the key guess is correct, the rounds can be removed, and the attack remounted on the reduced cipher.

In [221], Wagner considers using truncated differentials within a boomerang attack, but notes that while a differential used on plaintext P to cover $\alpha \rightarrow \beta$ can also be used to discover plaintext Q from $\beta \rightarrow \alpha$, the same does not hold for truncated differentials. Consequently while the probability for a successful boomerang attack with full differentials is $Pr(\alpha \rightarrow \beta)^2 \times Pr(\gamma \rightarrow \delta)^2$, in the case of truncated differentials it is $Pr(\alpha \rightarrow \beta) \times Pr(\delta \rightarrow \gamma)^2 \times Pr(\beta \rightarrow \alpha) \times Pr(w \oplus x \oplus y \in \beta | w \in \beta, x, y \in \gamma)$

The boomerang attack works from the outside of the cipher inwards. A related attack is the inside-out attack, also in [221] which works from the join of the segments outwards. It uses differentials ($\beta \rightarrow \alpha$) through E_0 and ($\beta \rightarrow \gamma$) through E_1. By collecting sufficient texts with the difference (β) at the join between E_0 and E_1, the cipher can be distinguished by recognizing differences between the plaintext and ciphertext pairs at a much greater rate than would be normally expected.

The amplified boomerang attack [120] is a chosen-plaintext variant of the boomerang attack that uses the inside-out attack to establish the quartet by chance, rather than by using adaptive queries. By choosing sufficiently many plaintexts pairs that match the first characteristic $\alpha \rightarrow \beta$ through E_0, some of the pairs can also be grouped to match $\beta \rightarrow \gamma$ through E_1 without the need to adaptively establish their identities. For m pairs of texts, the attacker can expect to acquire $m^2 \times 2^{-n} \times (pq)^2$ correct quartets, where n is the size of the block (and 2^{-n} the chance that two pairs that succeed through the first characteristic will match the starting requirements of the second).

Consequently the amplified boomerang attack uses much more text than the boomerang attack. However, this penalty is offset in three ways. If key guesses are used in conjunction with the attack, no extra text is required (which is not the case for an adaptive attack). Furthermore, the amplified-boomerang attack is able to work with tuples of texts, rather than just pairs. It is also amenable to extension with truncated differentials or differential-linear characteristics.

In [20], Biham et al. develop the rectangle attack by noting that the amplified boomerang attack counts the number of quartets that satisfy both the characteristics $\alpha \rightarrow \beta$ and $\gamma \rightarrow \delta$. However, these characteristics can be converted to differentials - it does not matter what the values of β and γ are so long as the characteristics are suitably low-probability. Also for two pairs that cover E_0, one with output difference a, the other with output difference b, the attack can be continued by recombining the pairs such that one covers E_1 using $\gamma \rightarrow \delta$ and the other using $(\gamma \oplus a \oplus b) \rightarrow \delta$. This improves the probability of acquiring a right quartet to $2^{-\frac{n}{2}}p'q'$ where p' is the probability of covering E_0 and q' the probability of covering E_1. However, finding the right quartets involves analyzing the ciphertext, which is a task quadratic in the number of pairs, so the rectangle trades a reduction in text pairs for an increase in time complexity. An improved algorithm in [23] improves the time complexity by performing preliminary tests on the ciphertexts before conducting a more thorough analysis on the texts that survive .

The results of the amplified-boomerang and rectangle attacks applied to contemporary ciphers are shown in Table A.3. In this table, CP refers to chosen plaintext and $APCP$ to adaptive chosen-plaintext chosen-ciphertext.

Cipher	Key Size	Type of Attack	Rounds	Complexity	
				Data	Time
MARS	256	Amplified Boomerang [120]	11	2^{65} CP	2^{229}
SAFER++	128	Boomerang [29]	5.5	2^{108} APCP	2^{108}
SHACAL-1	512	Amplified Boomerang [125]	47	$2^{158.5}$ CP	$2^{508.4}$
SHACAL-1	512	Rectangle Attack [24]	49	$2^{151.9}$ CP	$2^{508.5}$
Serpent	256	Amplified Boomerang [120]	8	2^{114} CP	2^{179}
SC2000		Rectangle [23]	3.5	2^{67} APCP	2^{67}

Table A.3: Boomerang and Rectangle Attacks Against Recent Ciphers

Integral Cryptanalysis

Integral cryptanalysis is a differential technique that was originally applied to reduced versions of SQUARE [60] and SQUARE-like ciphers such as the AES [62] and Hierocrypt-3 [186]. For other types of cryptanalysis, the pivotal points in the success of the attack are the strength of the s-boxes. But for integral cryptanalysis, so long as the s-boxes are bijective, they are irrelevant to the details of the attack. Instead the strength of the diffusion components have been the primary effecting factor in the complexities of the attack. This caught some designers off-guard, who were content to use AES-like diffusion elements such as the MDS. As a result, many integral attacks are uninteresting and virtually unmodified applications of the AES attack to similar ciphers such as [10]. The primary interest of recent attacks concerns how they need to adapt to modified diffusion elements [65], how they have become strengthened or weakened by flaws in the key schedule [65], [230], or by their application to non-SPN structures such as Feistel ciphers [153] or even bit-based ciphers such as IDEA [112] and DES [139].

Integral cryptanalysis is built around a key distinguisher that extends to some point in the cipher. The attacker then guesses key bits in the final rounds: if these guesses are correct, then the distinguished bytes can be identified; if they are incorrect, then the distinguished property will not appear. What is interesting about integral cryptanalysis is it takes some time for the distinguishing property to appear, and also that if it is not destroyed, the attacker has no measure to verify the correctness of the guessed key bytes. This limits the extent to which peeling off rounds can occur, so an Integral attack may be ineffective in identifying the master key of a cipher with a Type 2 key schedule. For many of the AES-like ciphers with Type 1 key schedules, only one or two round keys are necessary to retrieve the master key, without recourse to further iterations of the attack.

The integral attack is a chosen-plaintext attack, in which groups of plaintexts are chosen such that a single word of the block across the plaintexts is *saturated.* For byte-oriented ciphers like AES, this implies that the group contains 256 texts. For 32-bit word ciphers, such as Twofish, the group size is 2^{32} texts.

In all word positions but one, each of the texts in the group share the same value (denoted by in Figure A.2 by an empty cell). In the remaining word position, which situates the saturated word, each text has a unique value. This set of texts is called, in the terminology of [60], an Λ-set, and the saturated word called an active word (denoted by 'Λ'). The properties of each word position across the set of texts changes as the encryption progresses. When the texts at a particular position no longer have unique values, but their sum modulo 2 equals zero, the position is referred to as *balanced* (and denoted by '$\oplus$'). This property of balance is the key distinguishing property. At some stage in the attack, each word position will lose this property and become neither constant, active, nor balanced. This is denoted by '?', and its only use in the attack is during decryption, when the correct guess of the key words restores the '?' bytes to '$\oplus$' bytes.

From the distinguisher, a key recovery attack can be built. This involves guessing key bytes, the number of which is determined by the branch number of the diffusion element in those rounds involving guesses. Thus, the smaller the branch number of the cipher, the less complex the attack, as shown by the relative complexities of the attacks on Crypton (with a branch number of four) and the AES (with a branch number of five).

The crucial aspect of building a distinguisher is maintaining the balance for as many rounds as possible. Any non-linear bijective component, such as the AES s-box, will leave an active set unaltered, since it acts as a permutation on the complete set of inputs. It will destroy a balanced set in which some input values are missing. In AES, and similar ciphers, it is the interplay between the bijective substitution component ByteSub (γ) and the non-bijective high diffusion component MixColumn (θ) that both enables and limits the key distinguisher.

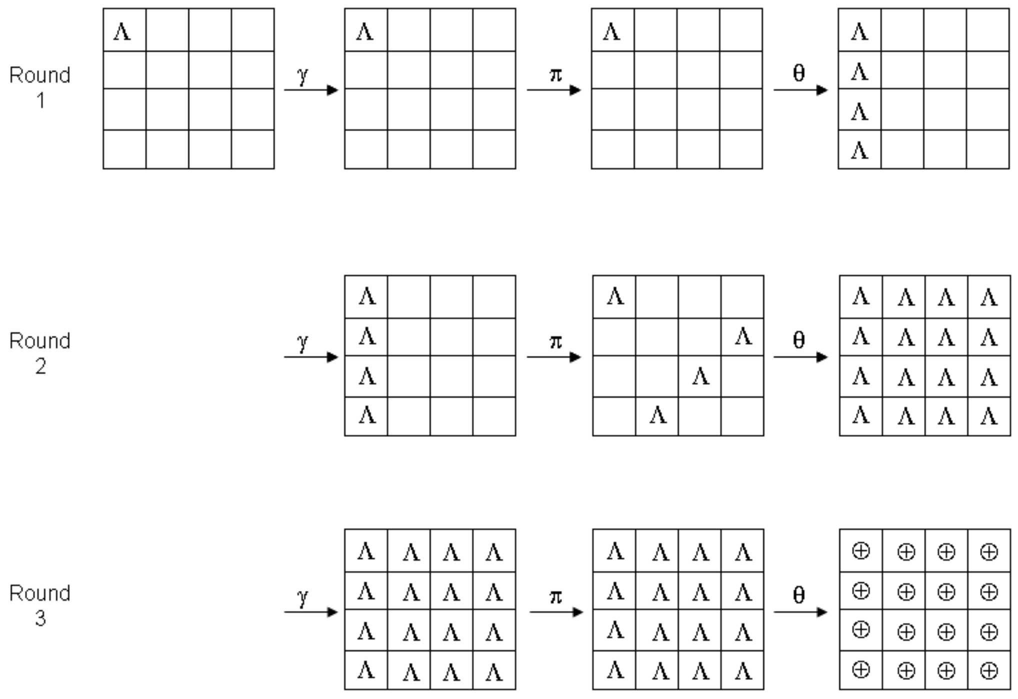

Figure A.2: Three-Round Integral Distinguisher for AES

When passed an array of bytes of which only one is active, the MixColumn propagates the active byte to all bytes in the output. When passed an array in which multiple bytes are active, and the others constant, it produces an output in which the bytes are balanced but not active. This enables an attacker to guess on any of the key bytes, since even though only one byte is initially active, the MixColumn ensures that all of the key bytes become active and then balanced. The subsequent ByteSub operation destroys this balance, and creates the point at which key guesses are verified.

As a concrete example, the three-round AES distinguisher, is shown in Figure A.2. The AddKey operation is not shown in this figure, since it has no effect on the development of the byte status throughout. In the distinguisher, the input differences are represented by a single active byte, which is transformed into a column of active bytes by the θ operation in the first round. The bijective, non-linear s-boxes in the γ operation do not change the active status of any of the differences. However, the π operation of the second round moves three of the active byte positions so that each column has a single active byte, each in a different row position. The θ operation in this round acts upon the single active byte in each column, transforming it to an entire column of active bytes. In the third round, the θ operation destroys the active status of each of the bytes as the operation is not bijective on its inputs. Now a complete set of values is not available in each byte position; the subsequent ByteSub operation (not shown) in the fourth round generates the distinguisher by destroying the balance of all of the bytes.

As shown in [153] and [139], the integral attacks can be wide ranging in their targets. For example, DES can be attacked, because for each of its 6×4 s-boxes, a set of all possible inputs produces a balanced set containing each output value four times. Since the other components in DES are linear, they behave in a similar way to AES's MixColumn, by propagating active sets and eventually transforming them to balanced sets.

In [139], Knudsen et al. note the similarities between Integral and Truncated Cryptanalysis, since both are concerned with the properties of bytes but not their actual values. They speculate that an Integral Attack may be combined with an Interpolation Attack, by treating each half of the cipher separately, and gluing the attacks in the middle. This approach is similar to Differential-Linear Cryptanalysis in Section A.4 and the Boomerang and Rectangle attacks in Section A.2.

Cipher	Key Size	Rounds	Complexity	
			Data	Time
AES [72]	256	8	2^{119} CP	2^{204}
Camellia [230]	256	9	$2^{60.5}$ CP	$2^{202.2}$
Crypton [65]	256	6	2^{32} CP	2^{56}
Hierocrypt-3 [10]	256	7	2^{36} CP	2^{176}
SAFER++ [29]	128	4.5	$2^{94.5}$ CP	$2^{94.5}$
Twofish [153]	256	8	2^{127} CP	2^{253}

Table A.4: Integral Attacks Against Recent Ciphers

A selection of attacks upon contemporary 128-bit block ciphers is shown in Table A.4. The complexities generated by the distinguishers are relatively low, but the attacks are extended across additional rounds by brute-force key word guessing. While a distinguisher enables verification of these guesses, it does not enable any short-cuts to be made in the number of bits guessed in each key word. This accounts for the high text complexity within each attack. An additional implication is that further research advances need to be made, before the attack can be extended to cover entire ciphers.

At present, the best defence against integral cryptanalysis has two elements. Firstly, use high diffusion components; these help minimize the number of rounds that the distinguisher covers, while increasing the number of key words that need to be guessed in later rounds. Secondly, use a sufficient number of rounds to prevent key guessing from the ciphertext to the point at which the distinguishing property emerges. The large block size of modern ciphers is a natural aid in making the number of key guesses prohibitive.

A.3 Linear Cryptanalysis

Linear cryptanalysis is a known-plaintext attack that replaces non-linear components within ciphers by probabilistic linear approximations. It is able to exploit the linear relationships between plaintext, ciphertext and key material, to derive a small number of round key bits in the first and/or last rounds. The remaining round key bits are derived via other means, typically a brute-force attack.

Linear cryptanalysis was developed in [162], but its power was demonstrated in [158] when it was used to break the DES with 2^{47} known-plaintexts, and again in [159] when the attack was refined with 2^{43} known-plaintexts and an 85% chance of success. In [114], Junod empirically found, that for some keys, the attack was successful with only 2^{39} texts. This is the best attack implemented on the DES. A summary of attacks by linear cryptanalysis on contemporary ciphers is shown in Table A.5. In this table, the notation KP refers to known plaintexts.

Linear cryptanalysis relies on replacing non-linear components with linear relationships, termed *approximations* of the form $P[\chi_P] \oplus C[\chi_C] = K[\chi_K]$, where χ_P represents a selection of plaintext bits $p_1, p_2, ..., p_a$, $P[\chi_P]$ its one-bit parity, and similarly for ciphertext C and key K.

The bias of an approximation is $\epsilon = |p - \frac{1}{2}|$, where p is the probability that the approximation is successful. The larger the value of ϵ, the fewer the number of plaintext-ciphertext pairs required to launch an attack using the approximation [158]. A negative bias, which means the approximation is wrong more often than it is right, is still useful, since the approximation can be complemented, and the bias inverted. But if $\epsilon = 0$, the approximation fails to model the component in as many cases as it succeeds, and is not useful in reconstructing key bits.

The plaintext, ciphertext and key bits in the approximation are selected specifically to optimize its probability. The probability p of an approximation is derived from the *linear approximation tables* (LAT) calculated for each non-linear component. The LAT is very similar to the DDT of differential cryptanalysis. It contains entries for each combination of the input and output patterns of the component. These are calculated by masking out bits of the input and output not involved in the approximation, and counting the number of times, for all desired inputs, that the linear parity of the selected inputs bits equals the linear parity of selected output bits. If the component is linear, then the parities will always match. If the component is affine, then the parities will never match. For non-linear components, the parities will sometimes match and sometimes not.

LATs are usually calculated for s-boxes. For $m \times n$ s-box S, with input mask α and output mask β, the LAT entry (α, β) is defined [158]:

$$\#\{x|0 \leq m, (\oplus_{s=0}^{log_2 m}(x[s] \bullet \alpha[s])) = (\oplus_{t=0}^{log_2 n}(S_a(x)[t] \bullet \beta[t]))\}$$

where $\bullet$ indicates masking. The probability that an approximation associated with (α, β) is valid is close to $\frac{LAT(\alpha,\beta)}{2^n}$. Good approximations for the non-linear component are acquired by choosing bits for those LAT cells with counts that deviate most from 2^{n-1}.

The approximations derived across non-linear components have to be expanded to cover single rounds, and ultimately, all but a few rounds in the cipher. They are concatenated so that the common terms between them cancel. According to the *piling-up lemma*, the probability of the resultant approximation is $\frac{1}{2} + 2^{n-1}\prod_{i=1}^{n}(p_i - \frac{1}{2})$, if the component approximations are independent [158]. In reality, the probability is usually close, regardless of independence [15].

An attacker constructs an approximation that covers all but the first and last rounds of the cipher, and uses it in one of two ways. In a *Type I attack*, an attacker collects N text pairs and derives a single key bit from an approximation $P[\chi_P] \oplus C[\chi_C] = K[\chi_K]$ which has bias ϵ. To derive the bit, the attacker calculates T, being the number of plaintext pairs for which $P[\chi_P] \oplus C[\chi_C]$ equals 0. If $T > \frac{N}{2}$ then the attacker guesses $K[\chi_K] = 0$ when $\epsilon > 0$ or $K[\chi_K] = 1$ when $\epsilon < 0$. If $T < \frac{N}{2}$ then the attacker inverts the guesses. By using the approximation multiple times in parallel, guessing other key material, and using counters to probabilistically observe successful guesses, many more key bits can be deduced. The number of known-plaintexts required to successfully execute a type I attack is approximately p^{-2} where p is the probability of the approximation used [158].

A *Type II attack* uses an approximation $(P[\chi_P] \oplus C[\chi_C] \oplus F_1(P_L, K_1)[\chi_{F_1}] \oplus$

$F_r(C_L, K_r)[\chi_{F_r}] = K[\chi_K]$ with bias ϵ, which is correlated to key bits used in the first (F_1) and last (F_r) rounds. The attacker considers only a small pool of key bits in these rounds (bits g in the F_1 and bits h in F_r) to optimize success of the attack. The approximation is expected to hold with a reasonable bias only if these key bits are guessed correctly [159]. For all possible values g and h, the attacker calculates statistics $T_{g,h}$ as the number of plaintexts in which $P[\chi_P] \oplus C[\chi_C] \oplus F_1(P_L, K_1)[\chi_{F_1}] \oplus F_r(C_L, K_r)[\chi_{F_r}]$ equal 0.

From the set of $T_{g,h}$ statistics, the attacker selects two statistics; T_{max}, which has the greatest value in the set, and T_{min}, which has the smallest value. If $|T_{max} - \frac{N}{2}| > |T_{min} - \frac{N}{2}|$, the attacker selects the key candidate corresponding to T_{max} and guesses $K[\chi_K] = 0$ when $\epsilon > 0$ or 1 when $\epsilon < 0$. If $|T_{max} - \frac{N}{2}| < |T_{min} - \frac{N}{2}|$, the attacker selects the the key candidate corresponding to T_{min} and guesses $K[\chi_K] = 1$ when $\epsilon > 0$ or 0 when $\epsilon < 0$.

The algorithm can be used to collect extra key bits from Feistel ciphers by inverting the roles of the plaintext and ciphertext and running it again [110].

Guessing key bits associated with round keys forces consideration of the role the key schedule plays, in choosing Type II approximations. The number of known-plaintexts required to execute a type II attack is roughly $8\epsilon^{-2}$ texts [137]. By using approximations that are limited to a only one non-linear component, the pool of key candidates can be kept small, reducing the complexity of the attack. For either attack method, the success rate is defined as $\Phi(2\sqrt{N}\epsilon)$, where Φ is the normal cumulative distribution function [110].

Attacks using linear cryptanalysis can be prevented by ensuring a sufficiently large input size of non-linear components [137]. This produces approximations with lower probabilities. Additionally, the values within the LAT of each non-linear component should be roughly uniform so that there are no prime candidates for generating approximations. Alternatively, the number of rounds of the cipher can be increased, resulting in a reduced probability for the resulting best approximation.

Using Multiple Linear Approximations

By increasing the number of plaintext-ciphertext pairs used in a linear cryptanalytic attack, the variance of the statistic T is diminished, and the key bits predicted by the statistic are more reliable - the chance of success increases. However, the variance can also be decreased by simultaneously using multiple approximations on the same key bits. If collection of the statistic T_i is carried out for i approximations, statistic $U = \sum_{i=1}^{n} a_i T_i$ can be calculated, where the weight of approximation i is $a_i = \epsilon_i / \sum_{i=1}^{n} \epsilon_i$. The statistic U has a comparable bias to each T_i but a reduced variance. The success rate of the attack improves to $\Phi(2\sqrt{N}\sqrt{n\epsilon})$, where n is the number of approximations [110].

For each non-linear component with b output bits, there are $2^b - 1$ linear approximations that have the same input mask, and operate on the same round key bits. However, multiple linear approximations have to operate on the same master key bits, so the key schedule has to be considered in calculating compatible approximations.

Using multiple linear approximations in an attack is quite effective if each linear approximation is linearly independent and has near maximal probability [210].

Cipher	Key Size	Type	Rounds	Complexity	
				Data	Time
FROG [222]	All	Linear	All	2^{56} KP	$< 2^{56}$
LOKI97 [136]	All	Linear	All	2^{56} KP	2^{56}
NUSH [228]	All	Linear	All	2^{62} KP	2^{53}
Q [119]	256	Linear	All	2^{97} KP	
RC6 [210]	256	Multiple Linear	14	$2^{119.68}$ KP	$2^{185.86}$
SAFER++ [111] (1 in 2^{-13} keys)	256	Linear	3.5	2^{81} KP	2^{178}
SC2000 [229]	All	Linear	4.5	$2^{115.17}$ KP	2^{42}
Serpent [19]	192, 256	Linear	11	2^{118} KP	2^{187}

Table A.5: Linear Cryptanalysis Against Recent Ciphers

Using Non-linear Approximations

For a given non-linear component, there are far more non-linear than linear approximations, including those with much better probabilities than for the best linear approximations. This means that using non-linear approximations can significantly reduce the number of ciphertexts required to launch an attack. However, there are difficulties in using non-linear approximations. For example, one-round non-linear approximations do not concatenate effectively [110].

Nevertheless, non-linear approximations can be used in the first and last rounds of a cipher, and in some cases, the second and penultimate rounds. These approximations are concatenated with linear approximations, to extend across the cipher with greater effectiveness than pure multiple-round linear equivalents.

The methodology of attacking ciphers with non-linear approximations is similar to that for standard linear cryptanalysis. In addition to those key bits normally retrieved in linear cryptanalysis, further key bits can be recovered when input bits to which they are related are involved as products in the approximation. By guessing the bits associated with the products, and observing the frequency of the resultant outputs in relation to the approximation probability, the attacker can determine the veracity of the guesses.

Despite the potential for this attack, no contemporary ciphers appear to have been studied, or at least successful attacked, from this angle.

Partitioning Cryptanalysis

In [90], Harpes et al. describe a generalization of linear cryptanalysis that replaces the linear approximation with an *I/O sum*. The I/O sum for the i^{th} round of a cipher is calculated as:

$$S^{(i)} = f_i(Y^{(i-1)}) \oplus g_i(Y^{(i)})$$

where f_i is a balanced binary function of the round input $Y^{(i-1)}$ and g_i is a balanced binary function of the round output $Y^{(i)}$. Standard linear cryptanalysis fixes the f and g functions to the parity function.

As with linear approximations, I/O sums can be composed, producing the sum:

$$S^{(1..\rho)} = \bigoplus_{i=1}^{\rho} S^{(i)} = g_0(Y^{(0)}) \oplus g_\rho(Y(^{(\rho)})$$

The bias of a linear approximation is replaced by the *key-dependent imbalance* of the I/O sum, calculated as $|2P[V = 0]-1|$, where V is $S^{(1...\rho)}|k^{(1...\rho)}$, conditioned upon k being the round key. The larger the imbalance (up to a maximum of 1), the more effective is the I/O sum.

An attack uses an I/O sum (for all but the last round), and N randomly chosen plaintext-ciphertext pairs. For a pair, each possible last round key is used to decrypt the ciphertext by one round. If the I/O sum of the plaintext and the decrypted ciphertext is 0, a counter is incremented for that key. This is repeated for the remaining pairs, and the keys chosen by the attacker are the ones for which the distance between the corresponding counter and $\frac{N}{2}$ is greatest.

A shortcut can be applied to this technique, by finding equivalence classes in which any two keys, k and k', of the class satisfy $g_{r-1}(F_{k'}^{-1}(y)) = g_{r-1}(F_k^{-1}(y)) \oplus c$ for some c and all ciphertexts y, where F is the round function and $r-1$ is the penultimate round number. By guessing only one key from each class, the efficiency of the attack is significantly increased. The attack predicts the class in which the correct key lies, with accuracy proportional to the imbalance of the I/O sum.

Partitioning cryptanalysis is a further generalization of this technique, in which chosen-plaintexts from a single partition are considered. The attack strategy is similar to that of the linear generalization. It is applied to ciphers in which bits in the output partition are non-uniformly distributed for random selections of plaintexts in the input partition. This implies the imbalance in the output partition is greater when the guessed key is correct. The computational complexity of the attack is proportional to the number of equivalence classes [91].

One example of partitioning cryptanalysis is the technique developed in [124], in which the partitions are simply formed modulo some prime integer. The technique is used to attack a modified form of RC5 in which the exclusive-ors are replaced by modular additions. The rotations of RC5P are approximated by

$$X \lll n \bmod 3 = \begin{cases} 2X\text{mod } 3 \text{ if } n \text{ odd} \\ X \bmod 3 \text{ if } n \text{ even} \end{cases}$$

and the additions by

$$(X+Y) \bmod\ 2^{32} \bmod 3 \quad = \begin{cases} X+Y \bmod 3 \text{ if no carry} \\ X+Y-1 \bmod 3 \text{ if carry} \end{cases}$$

A chosen-plaintext attack based on these approximation works by exhaustively searching on some key bits, and the partitions into which the last round key falls. For each guess, the partition into which the last round's input text falls is predicted, and a χ-square test applied. The guess to which the highest score applies is assumed to be the correct guess, and these key bits are used to aid in the guessing of further key bits in the same manner.

The generalization of linear cryptanalysis is claimed to be more powerful than linear cryptanalysis, and in turn, partitioning cryptanalysis is stronger still [107]. Both variants require small numbers of texts for successful identification of the correct key class. They have been theoretically applied to a number of ciphers, including six-round DES [91] and LOKI97 [136], although no complexities are given. In [124], it is suggested that a variant of RC5 that uses addition modulo some value other than 2^{32} may be vulnerable to the *mod n* range of attacks. However, partitioning cryptanalysis has not been responsible for any concrete attacks on contemporary block ciphers.

A.4 Differential-linear Cryptanalysis

Differential-linear (DL) cryptanalysis [146] is a chosen-plaintext attack that uses both differentials and linear approximations.

In the technique, a cipher of r rounds is divided into three segments, E_0 of m rounds, E_1 of $n = r - m - 1$ rounds, and the final round. The first segment E_0 is covered by a a truncated differential $\alpha \rightarrow \beta$ with probability $p = 1$, and the second E_1 by a linear approximation $\gamma \rightarrow \delta$ with probability q. The final round is used for guessing key bits.

Because both the differential and the approximation are linear operations that use invertible combiners such as exclusive-or, in some cases they can be concatenated to cover all but the last round. The approximation can be successfully joined to the differential if the input γ to the linear approximation does not use bits changed by the differential $\alpha \rightarrow \beta$. The bits used by the approximation $\gamma \rightarrow \delta$ remain unchanged at the output of the approximation with probability $q^2 + (1 - q)^2$, according to the piling up lemma. By using a differential of probability one, the cipher appears to have only n rounds, which can be attacked in the standard manner for linear cryptanalysis.

In [146], Langford applies this technique to six-round reduced DES, and retrieves ten key bits with a comparable complexity to differential and linear cryptanalysis, but with many fewer text pairs (approximately 640 pairs). The use of structures enables a further reduction in required text pairs.

In [21], Biham et al. relax the assumption that bits incorporated into the linear approximation need to be unchanged by the differential, and also that the probability of the differential needs to be 1. In the case that the differential bits do change or that the differential is not satisfied, the input and output parities behave randomly and do not suggest key bits correctly. Biham et al. successfully use a 4-round differential with probability $p < 1$ in conjunction with a 3-round linear approximation, to launch a key recovery attack on 9 rounds of DES. The probability of an enhanced DL-attack is $\frac{1}{2} + 4pq^2$. In [22], Biham et al. attack Serpent with a 256-bit master key by joining a 3-round differential with $p = 2^{-6}$ and a 6-round approximation with $q = 2^{-27}$. The attack covers 11 rounds and has a time complexity of $2^{139.2}$ and a data complexity $2^{125.3}$ chosen plaintexts.

A.5 Key Schedule Cryptanalysis

Differential and linear techniques are statistical attacks, in which the probability of success deteriorates as the number of cipher rounds grow. As processor speeds increase, it becomes easier to defeat these techniques by designing ciphers with a large number of rounds.

However, there is a class of techniques that jointly use homogeneity within the cipher algorithm and within the key schedule to engage in key recovery or deduction attacks. The success of these techniques is independent of the number of times a cipher round is iterated. Each technique relies on discovering the input and output to a small number of rounds to recover the round keys.

Three categories are presented here: related-key attacks, which rely on simplistic key schedules; slide attacks, which depend upon periodic key schedules and round functions; and related-cipher attacks, in which separate instances of a cipher are used with slightly different numbers of rounds.

Related-key Cryptanalysis

Related-key cryptanalysis [14] is a class of chosen-plaintext known- or chosen-key attacks in which plaintext-ciphertext pairs are obtained, such that the relationship between the keys used in each pair is well-known. In contrast to differential and linear cryptanalysis, the methodology of the attack changes between ciphers.

The first well-known related key attack [14] exploits the simple key schedule of LOKI89, in which each round key is derived from its predecessor using left rotations. The attack involves pairs of encryptions, in which the first round key of encryption X^* equals the second round key of encryption X. This implies that the second round key of encryption X^* equals the third of X and so on. Thus, if the plaintext of encryption X^* equals the plaintext of X encrypted over one round, then the ciphertext of X is identical to the ciphertext of X^* decrypted by one round. This observation enables the derivation of a linear equation involving plaintext, ciphertext and key bits, from which key bits can be easily acquired.

A particularly powerful form of related-key cryptanalysis uses differentials. This is similar to the resynchronization attack discussed in Section 4.2.8. In [122], Kelsey, Schneier and Wagner attack the cipher 3-Way [57]. The cipher has eleven rounds consisting of three layers, including a parallel application of thirty-two 3×3-bit s-boxes, a linear layer L, and a key addition layer in which a combined 96-bit round key and constant are added to the output of the linear layer. By inducing a difference $\alpha \oplus L(\beta)$ in the round key, they extend the characteristic across the round as the iterative characteristic $\alpha \rightarrow \alpha$. This allows them to build a large high-probability differential, with which they break the cipher. The attack succeeds on the full cipher, with one related-key query and 2^{22} chosen plaintexts.

Related-key cryptanalysis is a potent tool and has theoretically attacked many ciphers, including reduced round variants of the AES candidates DEAL [134] and SAFER+ [123]. It is a contentious topic as to whether the attack model, in which the attacker can choose related keys, is realistic. Master keys should be generated randomly so the attacker should have limited ability to launch related-key attacks; however cipher and protocol implementation details such as key management issues should not be the first line of defence. That job belongs to the block cipher key schedule. From this perspective, the realism of an attack is unimportant.

It is easy to protect against related key attacks, by adopting a Type 2C key schedule [122], [45]. Observing the number of Type 1 key schedules in Table 2.2 suggests this is a lesson that will only be learnt after related-key attacks inflict more casualties.

Slide-attacks

Slide attacks [32] apply to homogeneous ciphers, in which a weak function F is iterated several times. The function F is weak because given two instances $F(x, k) = y$ and $F(x^*, k) = y^*$, it is easy to deduce the key. The F function is generalized from the round function - it is p self-similar if it consists of p iterations of the round function [32]. This usually implies that the key schedule is periodic over p rounds. In the case of ciphers with a Type 1A key schedule, $p = 1$. This makes these ciphers particularly vulnerable to slide attacks.

The attack requires either known- or chosen-plaintexts, and involves identifying *slid pairs* within those texts. For text pairs (P, C) and (P^*, C^*), a slid pair occurs when $F(P) = P^*$ and $F(C) = C^*$, as if for two parallel encryptions, one has slipped forward a round.

According to the birthday paradox, one slid pair can be recovered from $2^{\frac{n}{2}}$ texts, where n is the block size of the encrypting cipher. Slid pairs can be identified by checking that $F(P_i) = P'_i$ and $C(P_i) = C'_i$ both hold for some key k. This has a complexity of 2^{n-1} operations, which is equivalent to exhaustive search. Improvements on this complexity can be based on particular function weaknesses, or when plaintexts contain redundancy. Identification of a slid pair rapidly leads to recovery of n round key bits.

Homogenous Feistel ciphers, and more generically, ciphers in which the length of the round key is smaller than the length of the master key, are particularly vulnerable to the slide attack. In the case of the homogenous Feistel cipher, the round only operates on half the block size. This means that vulnerable ciphers can be attacked with at most $2^{\frac{n}{2}}$ texts and work.

In [33], Biryukov introduces further optimizations for the attack against Feistel ciphers, in which they can be executed at the round level, even for self-similar structures where $p > 1$. These are the *complementation slide* and *sliding with a twist*.

The complementation slide applies to ciphers in which the period p of the key schedule is two; that is, the key schedule alternates between keys K_0 and K_1. With the complementation slide, the cipher is treated as though $p = 1$. The difference between the round keys $\triangle = K_0 \oplus K_1$ is cancelled out by introducing the difference between the chosen plaintexts. This is similar to the attack on 3-Way by Kelsey et al. (described earlier in this section). The complementation slide attack can be converted to a known-plaintext attack with complexity $2^{\frac{n}{2}}$. The advantage of this technique is obvious, since it is easier to retrieve key bits from a single weak round of the cipher than from two.

The sliding with a twist technique also applies to Feistel ciphers with self-similarity $p = 2$, and follows because decryption under keys K_1, K_0 is the same as encryption under K_0, K_1. By sliding a decryption one round against an encryption, parallel rounds use the same round key. This amplifies the self-similarity to one round. Again, this technique also permits a known-plaintext attack with complexity $2^{\frac{n}{2}}$.

These techniques can be combined to ciphers of self-similarity $p = 4$, with keys K_0, K_1, K_2, and K_3. After an encryption is slid by one round, keys K_0 and K_2 always meet in parallel at the odd rounds, but encryptions at the even rounds differ by $K_1 \oplus K_3$. By inducing a slid difference of $[0, K_1 \oplus K_3]$, the differences cancel. This technique allows a chosen-plaintext attack with complexity $2^{\frac{n}{4}}$.

The slide attack does not apply to heterogenous ciphers or those with Type 2 key schedules. It can be defeated by removing homogeneity within the round function and within the key schedule (for example, introducing counters into the key schedule, so that each round key is calculated in a slightly different way).

In [202], Saarinen presents slide attacks against some block ciphers based upon hash functions. He examines slide attacks against the NESSIE candidate SHACAL-1, and although he identifies slid pairs with complexity 2^{32}, he is unable to launch a practical attack on the cipher.

Related Cipher Attacks

In [226], Wu introduces the "Related Cipher Attack". This attack works on the same principle as slide attacks [32]: by gaining the text produced by the cipher just a few rounds from the production of the ciphertext, the intermediate weak rounds can be broken, exposing the round keys.

Slide attacks rely on homogenous round functions to acquire 'slid pairs'. Related cipher attacks rely on using two versions of the cipher, one of which has m rounds, and the other n rounds, $m \neq n$. In the case of slide attacks, the key schedule needs to be periodic to produce a slid pair. But in the case of related cipher attacks, one of the sequences of round keys needs to be a subset of the round key sequence of the related cipher. Given that for most practical ciphers, the number of rounds is fixed, or varies only as a function of the key length, the conditions under which the attack succeeds are contrived. For one example in which the attack is claimed to work, but which we refute due to an impractical attack model, see Section 3.3.2.

A.6 Algebraic Attacks

Algebraic attacks operate by reducing ciphers, or cipher rounds, to systems of equations in which key bits are unknown. By solving the system, some key bits are recovered. There are two classes of attacks discussed in this section:

interpolation attacks, and XLS attacks, which are related to the XL attacks that have devastated bit-based stream ciphers (see Section 4.2.9).

Interpolation Attack

The interpolation attack [107] applies to ciphers consisting of components in which the output is of a low non-linear order of the input. The attack is based upon Lagrange's interpolation formula:

$$f(x) = \sum_{i=1}^{n} y_i \prod_{i \leq j \leq n, j \neq i} \frac{x - x_j}{x_i - x_j}$$

for $2n$ elements $x_1, \ldots, x_n, y_1, \ldots, y_n$ in a finite field.

The attack applies to ciphers where the ciphertext can be expressed as a polynomial of the plaintext such that the number of coefficients n in the polynomial is less than or equal to 2^m, m being the block-size of the cipher. In this case, a global deduction algorithm can be devised with complexity n and requiring n known plaintexts.

Alternatively a key recovery algorithm can be devised, in which the polynomial is expressed in terms of the output of the next to last round. By guessing b key bits of the final round, the key recovery will succeed with complexity $2^{b-1}(n+1)$ and $n+1$ texts.

In [107], two simple Feistel ciphers are broken using the interpolation attack: $PURE$ and KN, both of which are defined around a low-order function $f : GF(2^{32}) \rightarrow GF(2^{32}), f(x) = x^3$. Both ciphers are defined to be immune to first order differential and linear cryptanalysis.

The same paper contains an attack on a modified version of the cipher SHARK, an ancestor of AES. SHARK uses s-boxes generated by inversion in a Galois field, which have a high algebraic degree, but could be simply expressed using rational expressions.

It is easy to avoid the interpolation attack by using components with complicated algebraic expressions. For example, although AES also uses s-boxes generated by inversion in a Galois field, it overcomes the problems experienced by the SHARK cipher, by modifying the inversion with a complex affine transformation. Consequently only a few ciphers have been broken using the interpolation attack, such as SNAKE [174].

XLS

In 2002, Courtois and Piepryzk [56] published an attack – XLS – that they claimed theoretically broke the 256-bit version of the Advanced Encryption standard. Without any concrete proof, this was bound to generate a lot of controversy [173]. In an early version [56], attacking AES with a 256-bit key required only 2^{87} operations, but after redoing their calculations, Courtois and Piepryzk retracted this claim. Nevertheless, the cryptographic community has cautiously embraced the attack in theory, and a related attack (XL) has been shown to break some stream ciphers (see Section 4.2.9).

The crux of the attack is that ciphers can be represented as multivariate equations in a field. The multi-quadratic (MQ) problem is known to be NP-complete, but when there are more equations than unknowns (the system is over-defined), then the complexity of the MQ problem drops substantially [209]. In [56], Courtois and Pieprzyk extend this result by developing a new algorithm - the XLS algorithm - for systems of over-defined equations that are regular and sparse. This algorithm is applied to SPN ciphers consisting of alternating layers of key addition, substitution boxes and linear transformations. Its designers conjecture that the algorithm may be applied to SA-ciphers (which alternate s-box layers and permutations) or even Feistel ciphers.

Two ciphers that Courtois et al. [56] specifically target are AES [62] and Serpent [4]. The only non-linear components in these ciphers are the s-boxes. In these ciphers, the s-boxes are constructed according to conventional wisdom that explicit equations with the output bits $y_i, 0 \le i < n$ should have high algebraic degree in the input bits $x_i, 0 \le i < m$. However, this does not discount that there can be equations of the form $P(x_1, ..., x_m, y_1, ..., y_n)$ of low algebraic degree. The number of explicit equations is necessarily fewer than n; however there can be many more implicit equations, of the latter form. When the number of equations r is close to the number of monomials t in the implicit equations, most of the terms can be eliminated to produce a system of simpler equations that are sparse and even linear.

In [56], Courtois and Pieprzyk show that 4×4 s-boxes always produce systems of over-defined equations. They also show that 8×8 s-boxes, when chosen randomly, are almost never over-defined. However, because the s-box for AES is not random, but instead defined according to exponentiation in a Galois field, it is strongly over-defined. They state a well-designed s-box should have a ratio of

$\frac{t}{r} \approx O(m^{d-1})$

The roots of XLS occur in relinearization [126]. The classical algorithm for solving over-defined equations is Buchberger's algorithm for constructing Grobner bases. This has a complexity of $O(2^{2.7n})$ and cannot handle systems with more than 15 variables. In [126], Kipnis and Shamir develop the process of relinearization. This simplifies the system of over-defined equations enabling it to be solved more efficiently by Gaussian elimination, by adding nonlinear equations that express the fact that some of the variables are related rather than independent.

XL [55] is an extension of relinearization. XL is a simple parameterized algorithm that takes the system of m equations in n variables, each of which has degree K. The parameter D determines the degree of the linearized equations in the final system. The algorithm has three steps:

1. Using all possible monomials as multiplicands, multiply all equations within the set by those monomials, such that the resulting equations have degree less than or equal to D.

2. Consider each monomial in the system as a new variable and linearize the system. The monomials must be ordered so that all terms containing a single variable are eliminated last.

3. If only one univariate equation remains, solve this equation in the field using Berlekamp's reduction algorithm.

XLS differs from XL primarily in step 1 by carefully choosing the monomials used as multiplicands to generate the sparseness within the system. The system of equations on the s-boxes are extended to the entire cipher, by incorporating linear equations for the diffusion layers. The linear components in the cipher add extra equations but do not generate new variables in the system.

Courtois and Pierprzyk [56] infer that ciphers that possess key schedules that are similar to the cipher algorithm may be targeted by a more specific form of XLS, but this theory is never developed.

Attacking a cipher using XLS requires only a few known plaintexts. Courtois and Pierpzryk [56] now claim that the complexity for the AES with a 256-bit key is 2^{298}, although they speculate that in combination with the result of [176], this could fall to 2^{87} which would be a spectacular break of the new standard. However, three years later, someone has yet to show how to do this. They claim a marginal break on Serpent with a complexity of 2^{210}. No concrete result has been shown on reduced-round versions of either of these ciphers, because the complexity of the attack grows very slowly with the number of rounds. In [28], Biryukov et al. develop equations for an XLS attack on Camellia, but do not expand it to an attack. Given the similarity of Camellia and other ciphers (see Section 2.2), this is not a surprising result.

Appendix B

Fast Implementation of Dragon

Unoptimized Dragon

The following code is the header file for an unoptimized version of Dragon.

```
#include "libcipher.h"
#include "dragon_sboxes.h"

#define  DRAGON_WORD_SIZE       32
#define  DRAGON_STATE_SIZE      32
#define  DRAGON_OUTPUT_SIZE     64

#define  DRAGON_KEY_SIZE        256
#define  DRAGON_IV_SIZE         256

typedef struct
{
    word32  w[DRAGON_STATE_SIZE];
    word32  m[2];
    word32  i;
} DragonCtx;

#define STATE_OFFSET(ctx, offset) \
    ctx->w[(ctx->i + offset) % DRAGON_STATE_SIZE]

/**
 * Initialize the Dragon cipher
 */
int dragon_init(DragonCtx *ctx, word32 *k, word32 *iv);
```

```
/**
 * Generate a new word of keystream
 * @param  ctx      [In/Out]  the dragon context
 * @param  ks       [Out]     preallocated array containing ks_size words of
 *                            keystream
 * @param  ks_size  [In]      size of the ks array in words
 */
void dragon_update(DragonCtx *ctx, word32 *ks, word32 size);
#endif _DRAGON_H_
```

The following code implements an unoptimized version of Dragon.

```
#include "dragon.h"

word32 g1(word32 x) {
   return (sbox2[BYTE3(x)] ^ sbox1[BYTE2(x)] ^
           sbox1[BYTE1(x)] ^ sbox1[BYTE0(x)]);
}

word32 g2(word32 x) {
   return (sbox1[BYTE3(x)] ^ sbox2[BYTE2(x)] ^
           sbox1[BYTE1(x)] ^ sbox1[BYTE0(x)]);
}

word32 g3(word32 x) {
    return (sbox1[BYTE3(x)] ^ sbox1[BYTE2(x)] ^
            sbox2[BYTE1(x)] ^ sbox1[BYTE0(x)]);
}

word32 h1(word32 x) {
    return (sbox1[BYTE3(x)] ^ sbox2[BYTE2(x)] ^
            sbox2[BYTE1(x)] ^ sbox2[BYTE0(x)]);
}

word32 h2(word32 x) {
    return  (sbox2[BYTE3(x)] ^ sbox1[BYTE2(x)] ^
             sbox2[BYTE1(x)] ^ sbox2[BYTE0(x)]);
}

word32 h3(word32 x) {
    return  (sbox2[BYTE3(x)] ^ sbox2[BYTE2(x)] ^
             sbox1[BYTE1(x)] ^ sbox2[BYTE0(x)]);
```

```
}

int update(word32 *v) {
    v[1] ^= v[0];
    v[3] ^= v[2];
    v[5] ^= v[4];

    v[2] += v[1];
    v[4] += v[3];
    v[0] += v[5];

    v[3] ^= g1(v[0]);
    v[5] ^= g2(v[2]);
    v[1] ^= g3(v[4]);

    v[0] ^= h1(v[1]);
    v[2] ^= h2(v[3]);
    v[4] ^= h3(v[5]);

    v[1] += v[4];
    v[3] += v[0];
    v[5] += v[2];

    v[0] ^= v[5];
    v[2] ^= v[1];
    v[4] ^= v[3];
}

/**
 * Initialize the Dragon cipher
 * @param  ctx  [Out]  the dragon context (state)
 */
int dragon_init(DragonCtx *ctx, word32 *k, word32 *iv) {
    word32  v[6] = { 0, 0, 0, 0, 0x00004472, 0x61676F6E };
    word32  i;

    ctx->i = 0;

    for (i = 0; i < 8; i++) {
        ctx->w[i]    = k[i];
        ctx->w[i+8]  = k[i] ^ iv[i];
        ctx->w[i+24] = iv[i];
        ctx->w[i+16] = ctx->w[i+8] ^ 0xFFFFFFFF;
    }

    for (i = 0; i < 16; i++) {
        v[0] = STATE_OFFSET(ctx, 0)  ^ STATE_OFFSET(ctx, 24) ^
               STATE_OFFSET(ctx, 28);

        v[1] = STATE_OFFSET(ctx, 1)  ^ STATE_OFFSET(ctx, 25) ^
               STATE_OFFSET(ctx, 29);

        v[2] = STATE_OFFSET(ctx, 2)  ^ STATE_OFFSET(ctx, 26) ^
               STATE_OFFSET(ctx, 30);

        v[3] = STATE_OFFSET(ctx, 3)  ^ STATE_OFFSET(ctx, 27) ^
               STATE_OFFSET(ctx, 31);

        update(v);

        ctx->i -= 4;
        if (ctx->i < 0) {
            ctx->i += 0x20;
        }

        STATE_OFFSET(ctx, 0) = v[0] ^ STATE_OFFSET(ctx, 20);
        STATE_OFFSET(ctx, 1) = v[1] ^ STATE_OFFSET(ctx, 21);
        STATE_OFFSET(ctx, 2) = v[2] ^ STATE_OFFSET(ctx, 22);
        STATE_OFFSET(ctx, 3) = v[3] ^ STATE_OFFSET(ctx, 23);
    }
    ctx->i    = 0;
    ctx->m[0] = v[4];
    ctx->m[1] = v[5];
    return 0;
}

/**
 * Generate a new word of keystream
 * @param  ctx      [In/Out]  the dragon context
 * @param  ks       [Out]     preallocated array containing ks_size words of
 *                            keystream
 * @param  ks_size  [In]      size of the ks array in words
 */
void dragon_update(DragonCtx *ctx, word32 *ks, word32 size) {
    word32 v[6] = {0, 0, 0, 0, ctx->m[0], ctx->m[1]};
    word32  i;
    for (i = 0; i < size;)  {
        v[0] = STATE_OFFSET(ctx, 0);
        v[1] = STATE_OFFSET(ctx, 9);
        v[2] = STATE_OFFSET(ctx, 16);
        v[3] = STATE_OFFSET(ctx, 19);
        v[4] = STATE_OFFSET(ctx, 30) ^ ctx->m[0];
        v[5] = STATE_OFFSET(ctx, 31) ^ ctx->m[1];

        update(v);

        ctx->i -= 2;

        STATE_OFFSET(ctx, 0) = v[1];
        STATE_OFFSET(ctx, 1) = v[2];

        ks[i++] = v[0];
        ks[i++] = v[4];

        ctx->m[1]++;
        ctx->m[0] += ((ctx->m[1] == 0) & 0x1);
    }
}
#endif
```

Optimized Dragon in C

The following code implements an optimized version of Dragon in C.

```
/**
 * @file dragon.c
 * Optimized implementation of the ISRC cipher Dragon
 * @author Matt Henricksen, copyright asserted 2004
 */

#include "dragon.h"

#define XOR(t1, t2, t3, t4, t5, t6) \
       t1 ^= t2; t3 ^= t4; t5 ^= t6;
#define ADD(t1, t2, t3, t4, t5, t6) \
        t1 += t2; t3 += t4; t5 += t6;
```

```
#define SBOX_G(t1, t2, t3, t4, t5, t6) \
        t1 ^= G1(t2); t3 ^= G2(t4); t5 ^= G3(t6);

#define SBOX_H(t1, t2, t3, t4, t5, t6) \
        t1 ^= H1(t2); t3 ^= H2(t4); t5 ^= H3(t6);

#define UPDATE(a, b, c, d, e, f) \
        XOR(b, a, d, c, f, e); \
        ADD(a, f, c, b, e, d); \
        SBOX_G(d, a, f, c, b, e); \
        SBOX_H(a, b, c, d, e, f); \
        ADD(b, e, d, a, f, c); \
        XOR(a, f, c, b, e, d);

/**
 * Initialize the Dragon cipher
 * @param  ctx  [Out]  the dragon context (state)
 */
int dragon_init(DragonCtx *ctx, word32 *k, word32 *iv) {
    word32  a, b, c, d;
        word32  e = 0x00004472, f = 0x61676F6E;
    word32  i;

    ctx->i = 0;

    for (i = 0; i < 8; i++) {
        ctx->w[i]    = k[i];
        ctx->w[i+8]  = k[i] ^ iv[i];
        ctx->w[i+24] = iv[i];
        ctx->w[i+16] = ctx->w[i+8] ^ 0xFFFFFFFF;
    }

    for (i = 0; i < 16; i++) {
        a = STATE_OFFSET(ctx, 0)  ^
            STATE_OFFSET(ctx, 24) ^
            STATE_OFFSET(ctx, 28);
        b = STATE_OFFSET(ctx, 1)  ^
            STATE_OFFSET(ctx, 25) ^
            STATE_OFFSET(ctx, 29);
        c = STATE_OFFSET(ctx, 2)  ^
            STATE_OFFSET(ctx, 26) ^
            STATE_OFFSET(ctx, 30);
        d = STATE_OFFSET(ctx, 3)  ^
```

```
            STATE_OFFSET(ctx, 27) ^
            STATE_OFFSET(ctx, 31);

        UPDATE(a, b, c, d, e, f);
        ctx->i -= 4;
        ctx->i &= 0x1F;


        STATE_OFFSET(ctx, 0) = a ^ STATE_OFFSET(ctx, 20);
        STATE_OFFSET(ctx, 1) = b ^ STATE_OFFSET(ctx, 21);
        STATE_OFFSET(ctx, 2) = c ^ STATE_OFFSET(ctx, 22);
        STATE_OFFSET(ctx, 3) = d ^ STATE_OFFSET(ctx, 23);
    }
    ctx->i    = 0;
    ctx->m[0] = e;
    ctx->m[1] = f;
    return 0;
}

/**
 * Generate a new word of keystream
 * @param  ctx      [In/Out]  the dragon context
 * @param  ks       [Out]     preallocated array containing ks_size words of
 *                            keystream
 * @param  ks_size  [In]      size of the ks array in words
 */
void dragon_update(DragonCtx *ctx, word32 *ks, word32 size) {
    word32 a, b, c, d, e, f;
    word32  i;

    for (i = 0; i < size;)  {
        a = STATE_OFFSET(ctx, 0);
        b = STATE_OFFSET(ctx, 9);
        c = STATE_OFFSET(ctx, 16);
        d = STATE_OFFSET(ctx, 19);
        e = STATE_OFFSET(ctx, 30) ^ ctx->m[0];
        f = STATE_OFFSET(ctx, 31) ^ ctx->m[1];

        UPDATE(a, b, c, d, e, f);

        ctx->i -= 2;

        STATE_OFFSET(ctx, 0) = b;
        STATE_OFFSET(ctx, 1) = c;

        ks[i++] = a;
        ks[i++] = e;

        ctx->m[1]++;
        ctx->m[0] += ((ctx->m[1] == 0) & 0x1);
    }
}
```

Dragon Update Function in Intel x86 Assembly

The following code implements the Dragon function F in Intel x86 assembly language.

```
  %include "c32.mac"

global _dragon_update

extern _sbox_1
extern _sbox_2

extern _a;
extern _b;
extern _c;
extern _d;
extern _e;
extern _f;

section .text

proc _dragon_update

; now the roles of e and a are swapped around
;   eax = e
;   ebx = b
;   edx = c
;   edi = d
;   esi = a
;   ebp = f

    mov  esi, [_a]
    mov  ebx, [_b]
    mov  edx, [_c]
    xor  ebx, esi

    mov  edi, [_d]
    xor  edi, edx

    mov  eax, [_e]
    mov  ebp, [_f]

    xor  ebp, eax
    mov  [_b],  ebx

    add  edx, ebx
    mov  [_d],  edi

    add  esi, edi
    mov  [_f],  ebp

    add  eax, ebp
    mov  [_c],  edx

    mov  [_e],  eax

; strict scheme for s-boxes
; 32-bit word is broken like this
; +-----------------------+
; | ebx | edx | edi | esi |
; +-----------------------+
;
; don't use ecx because it is being
; used as a counter
;
;   d ^= G1(a)

    mov  esi, [_a]
    mov  edi, [_a]

    and  esi, 0xFF
    shr  edi, 0x8

    mov  edx, [_a]
    and  edi, 0xFF
    mov  eax, [_d]
    shr  edx, 0x10

    mov  ebx, [_a]
    xor  eax, DWORD [_sbox_1 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_1 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_1 + edx * 4]

    and  ebx, 0xFF
    mov  esi, [_c]

    xor  eax, DWORD [_sbox_2 + ebx * 4]
    mov  edi, [_c]

;   f ^= G2(c)
    mov  [_d], eax
    and  esi, 0xFF

    shr  edi, 0x8
    mov  edx, [_c]

    and  edi, 0xFF
    mov  eax, [_f]

    shr  edx, 0x10

    mov  ebx, [_c]
    xor  eax, DWORD [_sbox_1 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_1 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_2 + edx * 4]

    and  ebx, 0xFF
    mov  esi, [_e]
    xor  eax, DWORD [_sbox_1 + ebx * 4]
    mov  edi, [_e]

; b ^= G3(e)
    mov  [_f], eax
    and  esi, 0xFF

    shr  edi, 0x8
    mov  edx, [_e]

    and  edi, 0xFF
    mov  eax, [_b]

    shr  edx, 0x10

    mov  ebx, [_e]
    xor  eax, DWORD [_sbox_1 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_2 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_1 + edx * 4]

    and  ebx, 0xFF
    mov  esi, [_d]

    xor  eax, DWORD [_sbox_1 + ebx * 4]
    mov  edi, [_d]

; c ^= H2(d)
    mov  [_b], eax
    and  esi, 0xFF

    shr  edi, 0x8
    mov  edx, [_d]

    and  edi, 0xFF
    mov  eax, [_c]

    shr  edx, 0x10

    mov  ebx, [_d]
    xor  eax, DWORD [_sbox_2 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_2 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_2 + edx * 4]

    and  ebx, 0xFF
    mov  esi, [_b]

    xor  eax, DWORD [_sbox_1 + ebx * 4]
    mov  edi, [_b]

; a ^= H1(b)
    mov  [_c], eax
    and  esi, 0xFF

    shr  edi, 0x8
    mov  edx, [_b]

    and  edi, 0xFF
    mov  eax, [_a]

    shr  edx, 0x10

    mov  ebx, [_b]
    xor  eax, DWORD [_sbox_2 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_2 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_1 + edx * 4]

    and  ebx, 0xFF
    mov  esi, [_f]

    xor  eax, DWORD [_sbox_2 + ebx * 4]
    mov  edi, [_f]

; e ^= H3(f)
    mov  [_a], eax
    and  esi, 0xFF

    shr  edi, 0x8
    mov  edx, [_f]

    and  edi, 0xFF
    mov  eax, [_e]

    shr  edx, 0x10

    mov  ebx, [_f]
    xor  eax, DWORD [_sbox_2 + esi * 4]

    and  edx, 0xFF
    xor  eax, DWORD [_sbox_1 + edi * 4]

    shr  ebx, 0x18
    xor  eax, DWORD [_sbox_2 + edx * 4]

    and  ebx, 0xFF
    mov  ebp, [_f]

    xor  eax, DWORD [_sbox_2 + ebx * 4]
    mov  edx, [_c]

; now the roles of e and a are swapped around
;   eax = e
;   ebx = b
;   edx = c
;   edi = d
;   esi = a
;   ebp = f

    mov  ebx, [_b]
    add  ebp, edx

    mov  esi, [_a]
    add  ebx, eax

    mov  edi, [_d]
    add  edi, esi

    mov  [_f],  ebp
    xor  edx, ebx

    mov  [_d],  edi
    xor  eax, edi

    mov  [_c],  edx
    xor  esi, ebp

    mov  [_a],  esi
    mov  [_e],  eax
    mov  [_d],  edi

endproc
```

Appendix C

Dragon Test Vectors

Dragon test vectors for a 256-bit master key and 256-bit IV are presented below.

```
KEY:
00001111 22223333 44445555 66667777 88889999 AAAABBBB CCCCDDDD EEEEFFFF
IV:
00001111 22223333 44445555 66667777 88889999 AAAABBBB CCCCDDDD EEEEFFFF
KEYSTREAM:
BC020767 DC48DAE3 14778D8C 927E8B32 E086C6CD E593C008 600C9D47 A488F622
3A2B94D6 B853D644 27E93362 ABB8BA21 751CAAF7 BD316595 2A37FC1E A3F12FE2
5C133BA7 4C15CE4B 3542FDF8 93DAA751 F5710256 49795D54 31914EBA 0DE2C2A7
8013D29B 56D4A028 3EB6F312 7644ECFE 38B9CA11 1924FBC9 4A0A30F2 AFFF5FE0

KEY:
00112233 44556677 8899AABB CCDDEEFF 00112233 44556677 8899AABB CCDDEEFF
IV:
00112233 44556677 8899AABB CCDDEEFF 00112233 44556677 8899AABB CCDDEEFF
KEYSTREAM:
8D3AB9BA 01DAA3EB 5CBD0F6D E3ECFCAB 619AF808 CF9C4A42 E2877766 6D2D7037
EE6F94AC 29D1EEE5 340DB047 8E91A679 480D8D88 2367CE2A 31C96AD4 49E70756
815EBEB2 290DBA7A 3CCB76A2 257BD122 2B0B7AED 917FAFFF 6B58B2B2 B05F24F6
E271A016 9E897BEF F5C22451 DA6F9E40 52B78BE5 6C97C1A5 C6F8E791 0F7B9C98
```

Appendix D

Dragon S-Boxes

The 8×8 s-boxes which are used at the core of Dragon's virtual 32×32 G and H s-boxes are presented below.

```
unsigned_word_32 sbox1[256]={
    0x393BCE6B,0x232BA00D,0x84E18ADA,0x84557BA7,0x56828948,0x166908F3,
    0x414A3437,0x7BB44897,0x2315BE89,0x7A01F224,0x7056AA5D,0x121A3917,
    0xE3F47FA2,0x1F99D0AD,0x9BAD518B,0x99B9E75F,0x8829A7ED,0x2C511CA9,
    0x1D89BF75,0xF2F8CDD0,0x2DA2C498,0x48314C42,0x922D9AF6,0xAA6CE00C,
    0xAC66E078,0x7D4CB0C0,0x5500C6E8,0x23E4576B,0x6B365D40,0xEE171139,
    0x336BE860,0x5DBEEEFE,0x0E945776,0xD4D52CC4,0x0E9BB490,0x376EB6FD,
    0x6D891655,0xD4078FEE,0xE07401E7,0xA1E4350C,0xABC78246,0x73409C02,
    0x24704A1F,0x478ABB2C,0xA0849634,0x9E9E5FEB,0x77363D8D,0xD350BC21,
    0x876E1BB5,0xC8F55C9D,0xD112F39F,0xDF1A0245,0x9711B3F0,0xA3534F64,
    0x42FB629E,0x15EAD26A,0xD1CFA296,0x7B445FEE,0x88C28D4A,0xCA6A8992,
    0xB40726AB,0x508C65BC,0xBE87B3B9,0x4A894942,0x9AEECC5B,0x6CA6F10B,
    0x303F8934,0xD7A8693A,0x7C8A16E4,0xB8CF0AC9,0xAD14B784,0x819FF9F0,
    0xF20DCDFA,0xB7CB7159,0x58F3199F,0x9855E43B,0x1DF6C2D6,0x46114185,
    0xE46F5D0F,0xAAC70B5B,0x48590537,0x0FD77B28,0x67D16C70,0x75AE53F4,
    0xF7BFECA1,0x6017B2D2,0xD8A0FA28,0xB8FC2E0D,0x80168E15,0x0D7DEC9D,
    0xC5581F55,0xBE4A2783,0xD27012FE,0x53EA81CA,0xEBAA07D2,0x54F5D41D,
    0xABB26FA6,0x41B9EAD9,0xA48174C7,0x1F3026F0,0xEFBADD8E,0x387E9014,
    0x1505AB79,0xEADF0DF7,0x67755401,0xDA2EF962,0x41670B0E,0x0E8642F2,
    0xCE486070,0xA47D3312,0x4D7343A7,0xECDA58D0,0x1F79D536,0xD362576B,
    0x9D3A6023,0xC795A610,0xAE4DF639,0x60C0B14E,0xC6DD8E02,0xBDE93F4E,
    0xB7C3B0FF,0x2BE6BCAD,0xE4B3FDFD,0x79897325,0x3038798B,0x08AE6353,
    0x7D1D20EB,0x3B208D21,0xD0D6D104,0xC5244327,0x9893F59F,0xE976832A,
    0xB1EB320B,0xA409D915,0x7EC6B543,0x66E54F98,0x5FF805DC,0x599B223F,
    0xAD78B682,0x2CF5C6E8,0x4FC71D63,0x08F8FED1,0x81C3C49A,0xE4D0A778,
    0xB5D369CC,0x2DA336BE,0x76BC87CB,0x957A1878,0xFA136FBA,0x8F3C0E7B,
    0x7A1FF157,0x598324AE,0xFFBAAC22,0xD67DE9E6,0x3EB52897,0x4E07E855,
    0x87CE73F5,0x8D046706,0xD42D18F2,0xE71B1727,0x38473B38,0xB37B24D5,
    0x381C6AE1,0xE77D6589,0x6018CBFF,0x93CF3752,0x9B6EA235,0x504A50E8,
    0x464EA180,0x86AFBE5E,0xCC2D6AB0,0xAB91707B,0x1DB4D579,0xF9FAFD24,
    0x2B28CC54,0xCDCFD6B3,0x68A30978,0x43A6DFD7,0xC81DD98E,0xA6C2FD31,
    0x0FD07543,0xAFB400CC,0x5AF11A03,0x2647A909,0x24791387,0x5CFB4802,
    0x88CE4D29,0x353F5F5E,0x7038F851,0xF1F1C0AF,0x78EC6335,0xF2201AD1,
    0xDF403561,0x4462DFC7,0xE22C5044,0x9C829EA3,0x43FD6EAE,0x7A42B3A7,
    0x5BFAAAEC,0x3E046853,0x5789D266,0xE1219370,0xB2C420F8,0x3218BD4E,
    0x84590D94,0xD51D3A8C,0xA3AB3D24,0x2A339E3D,0xFEE67A23,0xAF844391,
    0x17465609,0xA99AD0A1,0x05CA597B,0x6024A656,0x0BF05203,0x8F559DDC,
    0x894A1911,0x909F21B4,0x6A7B63CE,0xE28DD7E7,0x4178AA3D,0x4346A7AA,
    0xA1845E4C,0x166735F4,0x639CA159,0x58940419,0x4E4F177A,0xD17959B2,
    0x12AA6FFD,0x1D39A8BE,0x7667F5AC,0xED0CE165,0xF1658FD8,0x28B04E02,
    0x1FA480CF,0xD3FB6FEF,0xED336CCB,0x9EE3CA39,0x9F224202,0x2D12D6E8,
    0xFAAC50CE,0xFA1E98AE,0x61498532,0x03678CC0,0x9E85EFD7,0x3069CE1A,
    0xF115D008,0x4553AA9F,0x3194BE09,0xB4A9367D,0x0A9DFEEC,0x7CA002D6,
    0x8E53A875,0x965E8183,0x14D79DAC,0x0192B555};

unsigned_word_32 sbox2[256]={
    0xA94BC384,0xF7A81CAE,0xAB84ECD4,0x00DEF340,0x8E2329B8,0x23AF3A22,
    0x23C241FA,0xAED8729E,0x2E59357F,0xC3ED78AB,0x687724BB,0x7663886F,
    0x1669AA35,0x5966EAC1,0xD574C543,0xDBC3F2FF,0x4DD44303,0xCD4F8D01,
    0x0CBF1D6F,0xA8169D59,0x87841E00,0x3C515AD4,0x708784D6,0x13EB675F,
    0x57592B96,0x07836744,0x3E721D90,0x26DAA84F,0x253A4E4D,0xE4FA37D5,
    0x9C0830E4,0xD7F20466,0xD41745BD,0x1275129B,0x33D0F724,0xE234C68A,
    0x4CA1F260,0x2BB0B2B6,0xBD543A87,0x4ABD3789,0x87A84A81,0x948104EB,
    0xA9AAC3EA,0xBAC5B4FE,0xD4479EB6,0xC4108568,0xE144693B,0x5760C117,
    0x48A9A1A6,0xA987B887,0xDF7C74E0,0xBC0682D7,0xEDB7705D,0x57BFFEAA,
    0x8A0BD4F1,0x1A98D448,0xEA4615C9,0x99E0CBD6,0x780E39A3,0xADBCD406,
    0x84DA1362,0x7A0E984B,0xBED853E6,0xD05D610B,0x9CAC6A28,0x1682ACDF,
    0x889F605F,0x9EE2FEBA,0xDB556C92,0x86818021,0x3CC5BEA1,0x75A934C6,
    0x95574478,0x31A92B9B,0xBFE3E92B,0xB28067AE,0xD862D848,0x0732A22D,
    0x840EF879,0x79FFA920,0x0124C8BB,0x26C75B69,0xC3DAAAC5,0x6E71F2E9,
    0x9FD4AFA6,0x474D0702,0x8B6AD73E,0xF5714E20,0xE608A352,0x2BF644F8,
    0x4DF9A8BC,0xB71EAD7E,0x6335F5FB,0x0A271CE3,0xD2B552BB,0x3834A0C3,
    0x341C5908,0x0674A87B,0x8C87C0F1,0xFF0842FC,0x48C46BDB,0x30826DF8,
    0x8B82CE8E,0x0235C905,0xDE4844C3,0x296DF078,0xEFAA6FEA,0x6CB98D67,
    0x6E959632,0xD5D3732F,0x68D95F19,0x43FC0148,0xF808C7B1,0xD45DBD5D,
    0x5DD1B83B,0x8BA824FD,0xC0449E98,0xB743CC56,0x41FADDAC,0x141E9B1C,
    0x8B937233,0x9B59DCA7,0xF1C871AD,0x6C678B4D,0x46617752,0xAAE49354,
    0xCABE8156,0x6D0AC54C,0x680CA74C,0x5CD82B3F,0xA1C72A59,0x336EFB54,
    0xD3B1A748,0xF4EB40D5,0x0ADB36CF,0x59FA1CE0,0x2C694FF9,0x5CE2F81A,
    0x469B9E34,0xCE74A493,0x08B55111,0xEDED517C,0x1695D6FE,0xE37C7EC7,
    0x57827B93,0x0E02A748,0x6E4A9C0F,0x4D840764,0x9DFFC45C,0x891D29D7,
    0xF9AD0D52,0x3F663F69,0xD00A91B9,0x615E2398,0xEDBBC423,0x09397968,
    0xE42D6B68,0x24C7EFB1,0x384D472C,0x3F0CE39F,0xD02E9787,0xC326F415,
    0x9E135320,0x150CB9E2,0xED94AFC7,0x236EAB0F,0x596807A0,0x0BD61C36,
    0xA29E8F57,0x0D8099A5,0x520200EA,0xD11FF96C,0x5FF47467,0x575C0B39,
    0x0FC89690,0xB1FBACE8,0x7A957D16,0xB54D9F76,0x21DC77FB,0x6DE85CF5,
    0xBFE7AEE9,0xC49571A9,0x7F1DE4DA,0x29E03484,0x786BA455,0xC26E2109,
    0x4A0215F4,0x44BFF99C,0x711A2414,0xFDE9CDD0,0xDCE15B77,0x66D37887,
    0xF006CB92,0x27429119,0xF37B9784,0x9BE182D9,0xF21B8C34,0x732CAD2D,
    0xAF8A6A60,0x33A5D3AF,0x633E2688,0x5EAB5FD1,0x23E6017A,0xAC27A7CF,
    0xF0FC5A0E,0xCC857A5D,0x20FB7B56,0x3241F4CD,0xE132B8F7,0x4BB37056,
    0xDA1D5F94,0x76E08321,0xE1936A9C,0x876C99C3,0x2B8A5877,0xEB6E3836,
    0x9ED8A201,0xB49B5122,0xB1199638,0xA0A4AF2B,0x15F50A42,0x775F3759,
    0x41291099,0xB6131D94,0x9A563075,0x224D1EB1,0x12BB0FA2,0xFF9BFC8C,
    0x58237F23,0x98EF2A15,0xD6BCCF8A,0xB340DC66,0x0D7743F0,0x13372812,
    0x6279F82B,0x4E45E519,0x98B4BE06,0x71375BAE,0x2173ED47,0x14148267,
    0xB7AB85B5,0xA875E314,0x1372F18D,0xFD105270,0xB83F161F,0x5C175260,
    0x44FFD49F,0xD428C4F6,0x2C2002FC,0xF2797BAF,0xA3B20A4E,0xB9BF1A89,
    0xE4ABA5E2,0xC912C58D,0x96516F9A,0x51561E77};
```

Appendix E

Implementation of MUGI-M

The following code is the header file for an implementation of MUGI-M, the variant of MUGI described in Chapter 6

```
/**
 * @file mugi.h
 * Definitions for the MUGI PRNG
 * @author Matt Henricksen
 */
#ifndef _MUGI_H_
#define _MUGI_H_

#ifdef __cplusplus
extern "C" {
#endif

#include "libcipher.h"
#include "mugi_sboxes.h"

/* MUGI constants */
#define C0  0x6A09E667F3BCC908
#define C1  0xBB67AE8584CAA73B
#define C2  0x3C6EF372FE94F82B

#define MUGI_WORD_SIZE  64
#define INT_STATE_SIZE  3
#ifdef _STANDARD_MUGI_
    #define BUFFER_SIZE     16
#else /* _MUGI_M_  */
```

```
    #define BUFFER_SIZE      8
#endif
#define MUGI_STATE_SIZE (BUFFER_SIZE + MUGI_STATE_SIZE)

#define MUGI_KEY_SIZE    4
#define MUGI_IV_SIZE     4
#define MUGI_OUTPUT_SIZE 1

#ifdef _STANDARD_MUGI_
    /* ----------- STANDARD MUGI BUFFER UPDATE ----------- */
    /* b_i[t+1]    = b_{i-1}[t] (i \neq 0,4,10)           */
    /* b_0[t+1]    = b_{15}[t] \oplus a_0[t]              */
    /* b_4[t+1]    =    b_3[t] \oplus b_7[t]              */
    /* b_{10}[t+1] =    b_9[t] \oplus (b_{13}[t] \lll 32)  */
    /*                                                    */
    /* Increment taps because buffer has already shifted   */

    #define MUGI_TAP1         8
    #define MUGI_TAP2        14
    #define MUGI_TAP1_TARGET  4
    #define MUGI_TAP2_TARGET 10
#else /* MUGI_M */

    /* -------------- MUGI-M BUFFER UPDATE --------------- */
    /* b_i[t+1]    = b_{i-1}[t] (i \neq 0,4,10)           */
    /* b_0[t+1]    = b_7[t] \oplus a_0[t]                 */
    /* b_2[t+1]    = b_1[t] \oplus b_3[t]                 */
    /* b_5[t+1]    = b_4[t] \oplus b_6[t] \lll 32)        */
    /*                                                    */
    /* Increment taps because buffer has already shifted   */

    #define MUGI_TAP1        4
    #define MUGI_TAP2        7
    #define MUGI_TAP1_TARGET 2
    #define MUGI_TAP2_TARGET 5
#endif

/* MUGI context */
typedef struct {
    int     offset;
    word64  a[INT_STATE_SIZE];
    word64  b[BUFFER_SIZE];
} MugiCtx;
```

```
/**
 * Initialize the MUGI PRNG
 * @param  m    [In/Out]  MUGI generator
 * @param  key  [In]      128 bit key
 * @param  IV   [In]      128 bit IV
 * @returns MUGI_OK on success
 */
int mugi_init(MugiCtx* m, word32* key, word32* IV);

/**
 * Extract a 64-bit word from the MUGI PRNG
 * @param  m [In/Out]  MUGI generator
 * @returns pseudo-random word
 */
word64 mugi_update(MugiCtx *m);

#ifdef __cplusplus
}
#endif
#endif
```

The following code is the source file for an implementation of MUGI-M described in Chapter 6

```
/**
 * @file mugi.c
 * Reference implementation of cipher MUGI
 * @author Matt Henricksen, copyright asserted 2005
 */
#include "mugi.h"

#define ROTATE_LEFT(x, r)   ((x << r) | (x >> (64-r)))

/* Core function to the MUGI cipher */
word64 F(word64 a, word64 b)
{
    word64 x = a ^ b;
    word64 y = 0, y0, y1, z;

    /* starting from the right-most byte, push the first
     * first four bytes through the combined s-boxes/MDS */
    FOUR_SBOXES(x, y)
    y0 = y;
    y  = 0;

    /* push the last four bytes through the s-boxes/MDS */
    FOUR_SBOXES(x, y)
    y1 = y;

    /* execute the permutation */
    z   = (y0 << 32 | y0) & 0xFFFF00000000FFFF;
    y1  = (y1 << 32 | y1) & 0x0000FFFFFFFF0000;
    z  |= y1;

    return z;
}

/* RHO: state update function       */
/* inputs: a   - internal state     */
/*         b1  - buffer word 1      */
/*         b2  - buffer word 2      */
/*         tmp - temporary word     */
#define RHO(a, b1, b2, tmp) \
    tmp = a[2]; \
    a[2] = a[0] ^ F(a[1], ROTATE_LEFT(b2, 17)) ^ C2; \
    a[0] = a[1]; \
    a[1] = tmp ^ F(a[1], b1) ^ C0;

#define OFFSET(base, offset)  ((base + offset) & 0xF)
#define BUF(m, x)             m->b[OFFSET(x, m->offset)]

/* LAMBDA: buffer update function   */
/* inputs: m   - context            */
/*         tmp - temporary word     */
#define LAMBDA(m, tmp) \
    tmp = ROTATE_LEFT(BUF(m, MUGI_TAP2), 32); \
    m->offset--; \
    BUF(m, MUGI_TAP2_TARGET) ^= tmp; \
    BUF(m, MUGI_TAP1_TARGET) ^= BUF(m, MUGI_TAP1); \
    BUF(m, 0)                ^= m->a[0];

/**
 * Initialize the MUGI PRNG
 * @param  m    [In/Out]  MUGI generator
 * @param  key  [In]      192 bit key
 * @param  IV   [In]      192 IV
 * @returns MUGI_OK on success
 */
int mugi_init(MugiCtx* m, word32* k, word32* IV, int )
{
    word64  temp;
    int     idx = 0;

    m->offset = 0;

    /* Phase 1: Master Key Injection */
    m->a[0] = (k[0] << 32) | k[1];
    m->a[1] = (k[2] << 32) | k[3];
    m->a[2] = ROTATE_LEFT(m->a[0], 7) ^ ROTATE_LEFT(m->a[0], 57) ^ C0;

    /* Phase 2: State Mixing and Buffer Initialization */
    for (idx = 1; idx <= BUFFER_SIZE; idx++) {
         RHO(m->a, 0, 0, temp);
         m->b[BUFFER_SIZE-idx] = m->a[2];
    }

    /* Phase 3: Initialization Vector Injection */
    temp     = *(word64*)IV;
    m->a[0] ^= temp;
    m->a[2] ^= ROTATE_LEFT(temp, 7);

    temp     = *(word64*)(IV+1);
    m->a[1] ^= temp;
    m->a[2] ^= ROTATE_LEFT(temp, 57) ^ C1;

#ifdef _STANDARD_MUGI_ /* omitted from MUGI-M */
    /* Phase 4: Further State Mixing */
    for (idx = 0; idx < BUFFER_SIZE; idx++) {
        RHO(m->a, 0, 0, temp);
    }
#endif

    /* Phase 5: State and Buffer Mixing */
    for (idx = 0; idx < BUFFER_SIZE; idx++) {
        mugi_update(m);
    }
    return 0;
}

/**
 * Extract a 64-bit word from the MUGI PRNG
 * @param  m [In/Out]  MUGI generator
 * @returns pseudo-random word
 */
word64 mugi_update(MugiCtx *m)
{
    word64 t1   = BUF(m, MUGI_TAP1_TARGET);
    word64 t2   = BUF(m, MUGI_TAP2_TARGET);
    word64 res  = m->a[2];
    word64 temp = 0;

    LAMBDA(m, temp);
    RHO(m->a, t1, t2, temp);
    return res;
}
```

Bibliography

[1] Carlisle Adams. The CAST-128 encryption algorithm, May 1997. RFC 2144, Available at www.faqs.org/rfcs/rfc2144.html.

[2] Carlisle Adams. Designing aginst the Overdefined System of Equations Attack, May 2004. Available at http://eprint.iacr.org/2004/110/.

[3] Carlisle Adams and Jeff Gilchrist. The CAST-256 encryption algorithm, June 1999. RFC 2612, Available at www.faqs.org/rfcs/rfc2612.html.

[4] Ron Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the Advanced Encryption Standard. Available at http://www.cl.cam.ac.uk/ rja14/serpent.html.

[5] Anonymous. RC4 algorithm revealed. Posting to sci.crypt usenet group on 14 September, 1994. Available at ftp://idea.sec.dsi.unimi.it/pub/security/crypt/code/rc4.revealed.gz.

[6] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakahima, and Toshio Tokito. Camellia: A 128-bit block cipher suitable for multiple platforms. In Doug Stinson and Stafford Tavares, editors, *Selected Areas in Cryptography - SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, 2002.

[7] Frederik Armknecht. A linearization attack on the Bluetooth keystream generator. *Cryptology ePrint Archive, Report 2002/191*, 2002. Available at http://eprint.iarc.org/2002/191.

[8] Frederik Armknecht. Improving fast algebraic attacks. In Bimal Roy and Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 65–82. Springer-Verlag, 2004.

[9] Paulo Barreto and Vincent Rijmen. The ANUBIS Block Cipher, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[10] Paulo Barreto, Vincent Rijmen, Jorge Nakahara Jr., Bart Preneel, Joos Vandewalle, and Hae Kim. Improved SQUARE attacks against reduced-round HIEROCRYPT. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 165–173. Springer-Verlag, 2001.

[11] Larry Bassham. Efficiency testing of ANSI C implementations of round 2 candidate algorithms for the Advanced Encryption Standard. In *Proceedings from the Third Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, April 2000. Available at http://csrc.nist.gov/encryption/aes/.

[12] Olivier Baudron, Henri Gilbert, Louis Granboulan, Helena Handschuh, Antoine Joux, Phong Nguyen, Fabrice Noilhan, David Pointcheval, Thomas Pornin, Guillaume Poupard, Jacques Stern, and Serge Vaudenay. Report on the AES Candidates, 1999. Available at http://www.nist.gov/aes.

[13] Elwyn Berlekamp, Hal Fredricksen, and R Proto. Minimum conditions for uniquely determining the generator of a linear sequence. *Utilitas Math*, 5:305–315, 1974.

[14] Eli Biham. New types of cryptanalytic attacks using related keys. *Journal of Cryptology*, 7(4):229–246, 1994.

[15] Eli Biham. On Matsui's linear cryptanalysis. In Alfredo de Santis, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag, 1994.

[16] Eli Biham, Alex Biryukov, Niels Ferguson, Lars Knudsen, and Bruce Schneier. Cryptanalysis of MAGENTA. In *Proceedings from the Second Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, March 1999. Available at http://csrc.nist.gov/encryption/aes/.

[17] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In Jacques Stern, editor,

Advances in Cryptology - Proceedings of EUROCRYPT 99, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer-Verlag, 1999.

[18] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the middle attacks on IDEA and Khufu. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 124–137. Springer-Verlag, 1999.

[19] Eli Biham, Orr Dunkelman, and Nathan Keller. Linear cryptanalysis of reduced round Serpent. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 16–27. Springer-Verlag, 2001.

[20] Eli Biham, Orr Dunkelman, and Nathan Keller. The Rectangle Attack - Rectangling the Serpent. In Birgit Pfitzmann, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 2001*, volume LNCS 2045 of *Lecture Notes in Computer Science*, pages 340–357. Springer-Verlag, 2001.

[21] Eli Biham, Orr Dunkelman, and Nathan Keller. Enhancing differential-linear cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology - Proceedings of Asiacrypt 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 254–266. Springer-Verlag, 2002.

[22] Eli Biham, Orr Dunkelman, and Nathan Keller. Differential-linear cryptanalysis of Serpent. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 9–21. Springer-Verlag, 2003.

[23] Eli Biham, Orr Dunkelman, and Nathan Keller. New results on boomerang and rectangle attacks. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2003.

[24] Eli Biham, Orr Dunkelman, and Nathan Keller. Rectangle attacks on 49-round SHACAL-1. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*,

volume 2365 of *Lecture Notes in Computer Science*, pages 22–35. Springer-Verlag, 2003.

[25] Eli Biham, Vladimir Furman, Michael Misztal, and Vincent Rijmen. Differential cryptanalysis of Q. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 174–186. Springer-Verlag, 2001.

[26] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[27] Alex Biryukov. *Methods of Cryptanalysis*. PhD thesis, Technion Institute of Technology, Israel, 1999.

[28] Alex Biryukov and Christophe De Cannière. Block ciphers and systems of quadratic equations. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 274–289. Springer-Verlag, 2003.

[29] Alex Biryukov, Christophe De Cannière, and Gustaf Dellkrantz. Cryptanalysis of SAFER++. In Dan Boneh, editor, *Advances in Cryptology - Proceedings of CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 195–211. Springer-Verlag, 2003.

[30] Alex Biryukov and Adi Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology - Proceedings of Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2000.

[31] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2000.

[32] Alex Biryukov and David Wagner. Slide attacks. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 1999.

[33] Alex Biryukov and David Wagner. Advanced slide attacks. In Bart Preneel, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer-Verlag, 2000.

[34] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: a new high-performance stream cipher. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 325–344. Springer-Verlag, 2003.

[35] Nikita Borisov, Monica Chew, Rob Johnson, and David Wagner. Multiplicative differentials. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 17–33. Springer-Verlag, 2003.

[36] Johan Borst. The Block Cipher: GRAND CRU, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[37] Marc Briceno, Ian Goldberg, and David Wagner. A Pedagogical Implementation of A5/1, May 1999. Available at http://www.scard.org.

[38] Laurie Brown, Matthew Kwan, Josef Pieprzyk, and Jennifer Seberry. Improving resistance to differential cryptanalysis and the redesign of LOKI. In Hideki Imai, Ronald Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology - Proceedings of ASIACRYPT '91*, volume 739 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 1991.

[39] Laurie Brown and Josef Pieprzyk. Introducing the new LOKI97 block cipher. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[40] Laurie Brown, Josef Pieprzyk, and Jennifer Seberry. LOKI - a cryptographic primitive for authentication and secrecy applications. In Jennifer Seberry and Josef Pieprzyk, editors, *Advances in Cryptology - Proceedings of AUSCRYPT '90*, volume 753 of *Lecture Notes in Computer Science*, pages 229–236. Springer-Verlag, 1990.

[41] Linda Burnett, Gary Carter, Ed Dawson, and William Millan. Efficient methods for generating MARS-like s-boxes. In Bruce Schneier, editor, *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2000.

[42] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen Matyas Jr, Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS — A Candidate Cipher for AES. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[43] Christophe De Cannière. Guess and determine attack on SOBER. *Public report, NESSIE, NES/DOC/KUL/WP5/010*, 2001.

[44] Christophe De Cannière, Joseph Lano, Bart Preneel, and Joos Vandewalle. Distinguishing attacks on SOBER-t32. *Proceedings of the Third NESSIE workshop*, 2002.

[45] Gary Carter. *The Design, Analysis and Categorization of Block Ciphers and Their Components*. PhD thesis, Information Security Research Centre, Queensland University of Technology, May 1999.

[46] Gary Carter, Ed Dawson, and Lauren Nielsen. Key schedules of iterative block ciphers. In Colin Boyd and Ed Dawson, editors, *Proceedings of Information Security and Privacy - 3rd Australasian Conference, ACISP'98*, volume 1438 of *Lecture Notes in Computer Science*, pages 80–89. Springer-Verlag, July 1998.

[47] William Chambers. On random mappings and random permutations. In Bart Preneel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 22–28. Springer-Verlag, 1995.

[48] Kevin Chen, Matt Henricksen, Leonie Simpson, William Millian, and Ed Dawson. Dragon: A fast word based cipher. In *Information Secu-*

rity and Cryptology - ICISC '04 - Seventh International Conference, 2004. To appear in Lecture Notes in Computer Science.

[49] Don Coppersmith, Shai Halevi, and Charanjit Jutla. Cryptanalysis of stream ciphers with linear masking. In Moti Yung, editor, *Advances in Cryptology - Proceedings of CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 515–532. Springer-Verlag, 2002.

[50] Don Coppersmith, Hugo Krawczyk, and Yishay Mansour. The Shrinking Generator. In Doug Stinson, editor, *Advances in Cryptology - Proceedings of CRYPTO 93*, volume 773 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1994.

[51] Nicholas Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. *Cryptology ePrint Archive, Report 2002/087*, 2002. Available at http://eprint.iarc.org/2002/087.

[52] Nicolas Courtois. Algebraic attacks on combiners with memory and several outputs, 2003. Available at http://eprint.iacr.org/2003/125.pdf.

[53] Nicolas Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In Dan Boneh, editor, *Advances in Cryptology - Proceedings of CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 177–194. Springer-Verlag, 2003.

[54] Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 2003.

[55] Nicolas Courtois and Jacques Patarin. About the XL Algorithm over GF(2). In Marc Joye, editor, *Proceedings of RSA Conference 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 141–157. Springer-Verlag, 2003.

[56] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology - Proceedings of Asiacrypt 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer-Verlag, 2002.

[57] Joan Daemen. *Cipher and hash function design strategies based upon on linear and differential cryptanalysis.* PhD thesis, K.U. Leuven, March 1995.

[58] Joan Daemen and Craig Clapp. Fast hashing and stream encryption with PANAMA. In Serge Vaudenay, editor, *Proceedings of the 5th International Workshop on Fast Software Encryption*, volume 1372 of *Lecture Notes in Computer Science*, pages 60–74. Springer-Verlag, 1998.

[59] Joan Daemen, René Govaerts, and Joos Vandewalle. Weak keys for IDEA. In Doug Stinson, editor, *Advances in Cryptology - Proceedings of CRYPTO 93*, volume 773 of *Lecture Notes in Computer Science*, pages 224–231. Springer-Verlag, 1994.

[60] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher SQUARE. In Eli Biham, editor, *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer-Verlag, 1997.

[61] Joan Daemen, Michael Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie Proposal: NOEKEON, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[62] Joan Daemen and Vincent Rijmen. Rijndael. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[63] Ed Dawson, Gary Carter, Helen Gustafson, Matt Henricksen, William Millan, and Leonie Simpson. Evaluation of the MUGI psuedo-random number generator. Technical report, CRYPTREC, Information Technology Promotion Agency (IPA), Tokyo, Japan, 2002. Available at www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1035_IPA-MUGI_report_final.pdf.

[64] Ed Dawson, Andrew Clark, Jovan Golic, William Millan, Lyta Penna, and Leonie Simpson. The LILI-128 Keystream Generator. In Doug Stinson and Stafford Tavares, editors, *Selected Areas in Cryptography - SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 2002.

[65] Carl D'Halluin, Gert Bijnens, Vincent Rijmen, and Bart Preneel. Attack on six rounds of Crypton. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 46–59. Springer-Verlag, 1999.

[66] Markus Dichtl and Marcus Schafheutle. Linearity properties of SOBER-t32 key loading. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 225–230. Springer-Verlag, 2003.

[67] Patrik Ekdahl and Thomas Johansson. Snow - a new stream cipher, 2000. Available at http://www.it.lth.se/cryptology/snow/.

[68] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography - SAC'2002*, volume 2592 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2003.

[69] Patrik Ekdahl and Thomas Johansson. Distinguishing attacks on SOBER-t16 and t32. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 2003.

[70] Electronic Frontier Foundation. DES Cracker, July 1998. Available at http://www.eff.org/Privacy/Crypto_misc/DESCracker.

[71] The European Commission Community Research Information Societies Technology Programme. NESSIE-call for cryptographic primitives, 2000. Available at www.cosic.esat.luleuven.ac.be/nessie.

[72] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schenier, Mike Stay, David Wagner, and Doug Whiting. Improved cryptanalysis of Rijndael. In Bruce Schneier, editor, *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer-Verlag, 2000.

[73] Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, and Tadayoshi Kohno. Helix: fast encryption and authentication in a single cryptographic primitive. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 345–361. Springer-Verlag, 2003.

[74] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2001.

[75] Scott Fluhrer and David McGrew. Statistical analysis of the alleged RC4 keystream generator. In Bruce Schneier, editor, *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2000.

[76] Joanne Fuller and William Millan. Linear redundancy in s-boxes. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 74–86. Springer-Verlag, 2003.

[77] Joanne Fuller, William Millan, and Ed Dawson. Efficient implementation for analysis of cryptographic boolean functions. In *13th Australian Workshop On Combinatorial Algorithms*, Fraser Island, Australia, 2002.

[78] Dianelous Georgoudis, Damian Leroux, and Billy Simon Chaves. The FROG encryption algorithm. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[79] Richard Gerber. *The Software Optimization Cookbook*. Intel Press, 2002.

[80] Brian Gladman. AES second round implementation experience. In *Proceedings from the Second Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, March 1999. Available at http://csrc.nist.gov/encryption/aes/.

[81] Jovan Golic. Cryptanalysis of Alleged A5 Stream Cipher. In Walter Fumy, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1997.

[82] Jovan Golic. Linear statistical weakness of alleged RC4 keystream generator. In Walter Fumy, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 97*, volume 1233 of *Lecture Notes in Computer Science*, pages 226–238. Springer-Verlag, 1997.

[83] Jovan Golic. Security evaluation of MUGI. Technical report, CRYPTREC, Information Technology Promotion Agency (IPA), Japan, Tokyo, 2002.

[84] Jovan Golic and Luke O'Connor. Embedding and Probabilistic Correlation Attacks on Clock-Controlled Shift Registers. In Alfredo de Santis, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1994.

[85] Louis Granboulan, Phong Q. Nguyen, Fabrice Noilhan, and Serge Vaudenay. DFCv2. In Doug Stinson and Stafford Tavares, editors, *Selected Areas in Cryptography - SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 2002.

[86] Alex Grosul and David Wallach. A related key cryptanalysis of RC4. Technical report, TR-00-358, Department of Computer Science, Rice University, 2002.

[87] Helen Gustafson, Ed Dawson, Lauren Nielsen, and William Caelli. A computer package for measuring the strength of ciphers. *Journal of Computers and Security*, 13(8):687–697, 1997.

[88] Shai Halevi, Don Coppersmith, and Charanjit Jutla. Scream: a software-efficient stream cipher. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 195–209. Springer-Verlag, 2003.

[89] Helena Handschuh and David Naccache. SHACAL, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/tweaks.html/shacal_tweak.pdf.

[90] Carlo Harpes, Gerhard Kramer, and James Massey. A generalization of linear cryptanalysis and the applicability of Matsui's Piling-up lemma. In Louis Guillou and Jean-Jacques Quisquater, editors, *Advances in Cryptology - Proceedings of EUROCRYPT 95*, volume 921 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, 1995.

[91] Carlo Harpes and James Massey. Partitioning cryptanalysis. In Eli Biham, editor, *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 13–27. Springer-Verlag, 1997.

[92] Johan Håstad and Mats Näslund. BMGL: synchronous keystream generator with provable security, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[93] Philip Hawkes and Greg Rose. Primitive specification and supporting documentation for Sober t-32, 2000. Submission to NESSIE at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sober-t32.zip.

[94] Philip Hawkes and Greg Rose. Primitive specification and supporting documentation for the SOBER-t32 submission to NESSIE, 2000. Available at http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sobert-32.zip.

[95] Philip Hawkes and Greg Rose. Guess and determine attacks on SNOW. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography - SAC'2002*, volume 2592 of *Lecture Notes in Computer Science*, pages 37–46. Springer-Verlag, 2003.

[96] Philip Hawkes and Gregory Rose. Rewriting variables: the complexity of fast algebraic attacks on stream ciphers, 2004. Available at http://eprint.iacr.org/2004/081.pdf.

[97] Matt Henricksen. LibCipher software library for the Intel Pentium family, 2004. Available on request.

[98] Deukjo Hong, Jaechul Sung, Shiho Moriai, SangjinLee, and Jongin Lim. Impossible differential cryptanalysis of Zodiac. In Mitsuru Matsui, editor,

Proceedings of the 8th International Workshop on Fast Software Encryption, volume 2355 of *Lecture Notes in Computer Science*, pages 300–311. Springer-Verlag, 2001.

[99] Russ Housely and Doug Whiting. Wep fix using rc4 fast packet keying, 2002. Available at http://www.rsasecurity/rsalabs/technotes.

[100] IEEE Standards Association. IEEE 802.11 Specification, 1999. Available at http://standards.ieee.org/getieee802/802.11.html.

[101] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual Volume 1: Basic Architecture*. Intel Press, 2001.

[102] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual Volume 2: Instruction Set Reference*. Intel Press, 2001.

[103] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual Volume 3: System Programming Guide*. Intel Press, 2001.

[104] Intel Corporation. *Intel Architecture Optimization Reference Manual*. Intel Press, 2001.

[105] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*. Intel Press, 2001.

[106] Internet Engineering Task Force. IP Security Protocol (IPsec), 2004. Available at http://www.ietf.org/html.charters/ipsec-charter.html.

[107] Thomas Jakobsen and Lars Knudsen. The interpolation attack on block ciphers. In Eli Biham, editor, *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer-Verlag, 1997.

[108] Japanese Information Techology Promotion Agency. CRYPTREC: Call for Cryptographic Techniques in 2001, June 2000. Available at http://www.ipa.go.jp/security/enc/CRYPTREC/index-e.html#P2.

[109] Japanese Information Techology Promotion Agency. CRYPTREC Report 2002, 2003. Available at http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/ cryptrec20030829_report01e.html.

[110] Burt Kaliski Jr and Matt Robshaw. Linear cryptanalysis using multiple approximations. In Bart Preneel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 249–264. Springer-Verlag, 1995.

[111] Jorge Nakahara Jr. *Cryptanalysis and Design of Block Ciphers*. PhD thesis, Katholieke Universiteit Leuven, June 2003.

[112] Jorge Nakahara Jr, Paulo Barreto, Bart Preneel, Joos Vandewalle, and Hae Kim. SQUARE attacks against Reduced-Round PES and IDEA Block Ciphers, 2001. IACR Cryptology ePrint Archive, Report 2001/068.

[113] Michael Jacobson Jr and Klaus Huber. The Magenta block cipher algorithm. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[114] Pascal Junod. On the complexity of Matsui's attack. In Serge Vaudenay and A M Youssef, editors, *Selected Areas in Cryptography - SAC'2001*, volume 2259 of *Lecture Notes in Computer Science*, pages 199–211. Springer-Verlag, 2002.

[115] Pascal Junod and Serge Vaudenay. FOX: a new family of block ciphers. In *Selected Areas in Cryptography (SAC'04)*, pages 15–29, 2004. To appear.

[116] Charanjit Jutla. Encryption modes with almost free message integrity, 2000. Available at http://eprint.iacr.org/2000/039.

[117] John Kam and George Davida. Structured design of substitution-permutation encryption networks. *IEEE Transactions on Computers*, 28(10):747–753, October 1979.

[118] Takeshi Kawabata and Toshinobu Kaneko. A study on higher order differential attack on Camellia. In *Proceedings of the 2nd Open NESSIE Workshop*, September 2001.

[119] Liam Keliher, Henk Meijer, and Stafford Tavares. High probability linear hulls in Q. In *Proceedings of the 2nd Open NESSIE Workshop*, September 2001.

[120] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified boomerang attacks against reduced MARS and Serpent. In Bruce Schneier, editor, *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 75–93. Springer-Verlag, 2000.

[121] John Kelsey, Bruce Schneier, and David Wagner. Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In Neal Koblitz, editor, *Advances in Cryptology - Proceedings of CRYPTO 96*, volume 1109 of *Lecture Notes in Computer Science*, pages 237–251. Springer-Verlag, 1996.

[122] John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-Way, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communications Security, First International Conference Proceedings*, pages 233–246. Springer-Verlag, 1997.

[123] John Kelsey, Bruce Schneier, and David Wagner. Key-schedule weakness in SAFER+. In *Proceedings from the Second Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, March 1999. Available at http://csrc.nist.gov/encryption/aes/.

[124] John Kelsey, Bruce Schneier, and David Wagner. Mod n Cryptanalysis, with Applications against RC5P and M6. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 139–155. Springer-Verlag, 1999.

[125] Jongsung Kim, Dukjae Moon, Wonil Lee, Seokhie Hong, Sangjin Lee, and Seokwon Jung. Amplified boomerang attack against reduced-round SHA-CAL. In Yuliang Zheng, editor, *Advances in Cryptology - Proceedings of Asiacrypt 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 243–253. Springer-Verlag, 2002.

[126] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. In Michael Wiener, editor, *Advances in Cryptology - Proceedings of CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 1999.

[127] Vlastimil Klima. Cryptanalysis of Hiji-Bij-Bij (HBB), January 2005. Available at http://eprint.iacr.org/2005/03/.

[128] Alexander Klimov and Adi Shamir. New Cryptographic Primitives Based on Multiword T-Functions. In Bimal Roy and Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2004.

[129] Lars Knudsen. Cryptanalysis of LOKI. In *Advances in Cryptology - Proceedings of AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 196–208. Springer-Verlag, 1992.

[130] Lars Knudsen. New potentially weak keys for DES and LOKI. In Alfredo de Santis, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 419–424. Springer-Verlag, 1994.

[131] Lars Knudsen. Practically secure Feistel ciphers. In *Proceedings of the 1st International Workshop on Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 211–221. Springer-Verlag, 1994.

[132] Lars Knudsen. A key-schedule weakness in SAFER K-64. In Don Coppersmith, editor, *Advances in Cryptology - Proceedings of CRYPTO 95*, volume 963 of *Lecture Notes in Computer Science*, pages 274–286. Springer-Verlag, 1995.

[133] Lars Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 1995.

[134] Lars Knudsen. Deal: A 128-bit block cipher. Technical Report 151, Department of Informatics,University of Bergen, Norway, Feb 1998. Available at citeseer.ist.psu.edu/knudsen98deal.html.

[135] Lars Knudsen and Vincent Rijmen. On the Decorrelated Fast Cipher (DFC) and its theory. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 1999.

[136] Lars Knudsen and Vincent Rijmen. Weaknesses in LOKI97. In *Proceedings from the Second Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, pages 168–174, March 1999. Available at http://csrc.nist.gov/encryption/aes/.

[137] Lars Knudsen and Matt Robshaw. Non-linear approximations in linear cryptanalysis. In Ueli Maurer, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 224–236. Springer-Verlag, 1996.

[138] Lars Knudsen, Matt Robshaw, and David Wagner. Truncated differentials and Skipjack. In Michael Wiener, editor, *Advances in Cryptology - Proceedings of CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 1999.

[139] Lars Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer-Verlag, 2003.

[140] Nick Komninos, Bahram Honary, and Michael Darnell. An Efficient Stream Cipher for Mobile and Wireless Devices. In Brian Honary, editor, *Proceedings of Cryptography and Coding - 8th IMA International Conference, Cirencester UK*, volume 2260 of *LNCS*, pages 294–300. Springer-Verlag, December 2001.

[141] Korea Information Security Agency. SEED algorithm specification, 2000. Available at http://www.kisa.or.kr/seed/data/Document_pdf/ SEED_Specification_english.pdf.

[142] Xue Lai. *Communications and Cryptography: two sides of one tapestry*, chapter Higher order derivatives and differential cryptanalysis, pages 227–233. Kluwer Academic Publishers, 1994.

[143] Xue Lai and James Massey. A proposal for a new block encryption standard. In Ivan Damgard, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1991.

[144] Xue Lai, James Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Joan Feigenbaum, editor, *Advances in Cryptology - Proceedings of CRYPTO 91*, volume 576 of *Lecture Notes in Computer Science*, pages 17–38. Springer-Verlag, 1991.

[145] LAN Crypto. NUSH specification, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[146] Susan Langford and Martin Hellman. Differential-linear cryptanalysis. In Yvo Desmedt, editor, *Advances in Cryptology - Proceedings of CRYPTO 94*, volume 839 of *Lecture Notes in Computer Science*, pages 17–24. Springer-Verlag, 1994.

[147] Dong Hoon Lee, Jaeheon Kim, Jin Hong, Jae Woo Han, and Dukjae Moon. Algebraic Attacks on Summation Generators. In Bimal Roy and Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 34–48. Springer-Verlag, 2004.

[148] Hoonjae Lee and Sangjae Moon. Parallel stream cipher for secure high-speed communications. *Signal Proceesing*, 82(2):137–143, February 2002.

[149] Marcus Leech. A Feistel cipher with hardened key scheduling. In *Selected Areas in Cryptography - SAC'96*, pages 15–29, 1996.

[150] Chae Hoon Lim. A revised version of Crypton - Crypton V1.0. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 1999.

[151] Helgar Lipmaa. AES Ciphers: Speed. Available at http://www.tcs.hut.fi/ helger/aes/round2.html, October 2001.

[152] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology - Proceedings of CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[153] Stefan Lucks. The saturation attack - a bait for Twofish. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software*

Encryption, volume 2355 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2001.

[154] Itsik Mantin. Analysis of the Stream Cipher RC4. Master's thesis, Weizmann Institute of Science, Israel, November 2001.

[155] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer-Verlag, 2001.

[156] James Massey, Gurgen Khachatrian, and Melsik Kuregian. Safer++, 2000. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[157] James Massey and Charles William. Safer+. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[158] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1993.

[159] Mitsuru Matsui. The first experimental cryptanalysis of the Data Encryption Standard. In Yvo Desmedt, editor, *Advances in Cryptology - Proceedings of CRYPTO 94*, volume 839 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1994.

[160] Mitsuru Matsui. New Block Encryption Algorithm MISTY. In Eli Biham, editor, *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 1997.

[161] Mitsuru Matsui and Toshio Tokia. Cryptanalysis of a reduced version of the block cipher E2. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 71–80. Springer-Verlag, 1999.

[162] Mitsuru Matsui and A Yamagishi. A new method for known plaintext attack of FEAL cipher. In Rainer Rueppel, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 92*, volume 658 of *Lecture Notes in Computer Science*, pages 81–91. Springer-Verlag, 1992.

[163] Lauren May. *Design, Analysis and Implementation of Symmetric Block Ciphers*. PhD thesis, Information Security Research Centre, Queensland University of Technology, January 2002.

[164] Leslie McBride. Q: A Proposal for NESSIE v2.00, 2001. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions.html.

[165] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic attacks and decomposition of boolean functions. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - Proceedings of EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 474–491. Springer-Verlag, 2004.

[166] Mihodrag Mihaeljevic. Report on security evaluation of MUGI stream cipher. Technical report, CRYPTREC, Information Technology Promotion Agency (IPA), Tokyo, Japan, 2002.

[167] Miodrag Mihaeljevic. Report on security evaluation of RC4 stream cipher. Technical report, CRYPTREC, Information Technology Promotion Agency (IPA), Japan, Tokyo, 2002.

[168] William Millan, Andrew Clark, and Ed Dawson. Smart hill climbing finds better boolean functions. In *Selected Areas in Cryptography - SAC'1997*, pages 50–63. Springer-Verlag, 1997.

[169] William Millan, Joanne Fuller, and Ed Dawson. New concepts in evolutionary search for boolean functions in cryptology. *The 2003 Congress on Evolutionary Computation - CEC '03*, 3:2157–2164, 2003.

[170] Ivan Miranov. (Not so) random shuffles of RC4. In Moti Yung, editor, *Advances in Cryptology - Proceedings of CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 2002.

[171] Chris Mitchell. Remarks on the Security of the Alpha1 Stream Cipher. Technical Report RHUL-MA-2001-8, Royal Holloway, University of London, December 2001. Available at http://www.ma.rhul.ac.uk/techreports/2001/RHUL-MA-2001-8.pdf.

[172] Shoji Miyaguchi, Akira Shiraishi, and Akihiro Shimizu. Fast Data Encryption Algorithm FEAL-8. *Review of Electrical Communications Laboratories*, 36(4):433–437, 1988.

[173] Tzuong-Tsieng Moh. On the Courtois-Pieprzyk's attack on Rijndael, 2002. Available at http://www.usdsi.com/aes.html.

[174] Shiho Moriai, Takeshi Shimoyama, and Toshinobu Kaneko. Interpolation attacks of the block cipher: SNAKE. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 275–290. Springer-Verlag, 1999.

[175] Frédéric Muller. Differential attacks against the Helix stream cipher. In Bimal Roy and Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 2004.

[176] Sean Murphy and Matthew Robshaw. Essential algebraic structure within the AES. In Moti Yung, editor, *Advances in Cryptology - Proceedings of CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2002.

[177] National Bureau of Standards. Data Encryption Standard. Federal Information Processing Standard (FIPS), Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C, January 1977.

[178] National Institute of Standards and Technology. Announcing the Advanced Encryption Standard, November 2001. Available at http://csrc.nist.gov/CryptoToolkit/aes/.

[179] National Institute of Standards and Technology (NIST) (Computer Security Division). Announcing request for candidate algorithm nominations for the Advanced Encryption Standard (AES), 1997. available at www.nist.gov/aes.

[180] James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, and Edward Robak. Status report on the first round of the development of the Advanced Encryption Standard. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1999. Available at http://csrc.nist.gov/encryption/aes/.

[181] NESSIE Committee. NESSIE Security Report version 2, February 2003. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/D20V2.pdf.

[182] NESSIE Committee. Portfolio of recommended cryptographic primitives, February 2003. Available at https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/decision-final.pdf.

[183] Nippon Telegraph and Telephone Corporation. Specification of E2 - a 128-bit block cipher, June 1998. Available at http://info.isl.ntt.co.jp/e2/E2spec.pdf.

[184] Kaisa Nyberg. Differentially uniform mappings for cryptography. In Tor Helleseth, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Springer-Verlag, 1993.

[185] Kaisa Nyberg and Lars Knudsen. Provable security against differential cryptanalysis. In Rainer Rueppel, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 92*, volume 658 of *Lecture Notes in Computer Science*, pages 556–574. Springer-Verlag, 1992.

[186] Kenji Ohkuma, Hirofumi Muratani, Fumihiko Sano, and Shinichi Kawamura. Specification on a block cipher: Hierocrypt-3. In Doug Stinson and Stafford Tavares, editors, *Selected Areas in Cryptography - SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, 2002.

[187] OpenSSL. Available at http://www.openssl.org.

[188] Souradyuti Paul and Bart Preneel. A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher. In

Bimal Roy and Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 245–249. Springer-Verlag, 2004.

[189] Lyta Penna. Implementation issues in symmetric ciphers. Master's thesis, Information Security Research Centre, Queensland University of Technology, 2002.

[190] Gilles Piret and Jean-Jacques Quisquater. Integral cryptanalysis on reduced-round SAFER-++. *Cryptology ePrint Archive, Report 2003/033*, 2003. Available at http://eprint.iacr.org/2003/033.pdf.

[191] W Price and D Davies. *Security for computer networks: an introduction to data security in teleprocessing and electronic funds transfer.* Wiley Series In Computing, 1984.

[192] Jean-Jacques Quisquater and Chantal Couvreur. Fast decipherment for RSA public-key cryptosystem. *Electronic Letters*, 18:905–907, 1982.

[193] Vincent Rijmen, Joan Daemen, Bart Preneel, Anton Bosselaers, and Erik DeWin. The cipher SHARK. In Dieter Gollman, editor, *Proceedings of the 3th International Workshop on Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 99–111. Springer-Verlag, 1996.

[194] Ron Rivest. A description of the RC2 encryption algorithm. File draft-rivest-rc2desc-00.txt available from ftp://ftp.ietf.org/internet-drafts/.

[195] Ron Rivest. RSA Security Response to weaknesses in key scheduling algorithm of RC4, 2001. Technical note available from RSA Security Inc. site. http://www.rsasecurity.com/rsalabs/technotes/wep.html.

[196] Ronald Rivest. The RC5 encryption algorithm. In Bart Preneel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer-Verlag, 1995.

[197] Ronald Rivest, Matt Robshaw, Ray Sidney, and Yi Qin Lisa Yin. The RC6 block cipher. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.

[198] Phil Rogaway and Don Coppersmith. A software-optimized encryption algorithm. In Bart Preneel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 56–63. Springer-Verlag, 1995.

[199] Philip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. A block-ciper mode of operation for efficient authenticated encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 196–205, August 2001.

[200] Andrew Roos. Weak keys in RC4. Posting to sci.crypt usenet group on 22 September, 1995.

[201] Gregory Rose and Philip Hawkes. Turing: a fast stream cipher. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 307–324. Springer-Verlag, 2003.

[202] Markku-Juhani O. Saarinen. Cryptanalysis of block ciphers based on SHA-1 and MD5. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 26–44. Springer-Verlag, 2003.

[203] Palash Sarkar. Hiji-bij-bij: a new stream cipher with a self-synchronous mode of operation. *Cryptology ePrint Archive, Report 2003/014*, 2003. Available at http://eprint.iarc.org/2003/014.

[204] Bruce Schneier. Description of a new variable-length key 64-bit block cipher Blowfish. In *Proceedings of the 1st International Workshop on Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204. Springer-Verlag, 1994.

[205] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley & Sons, Inc., 1999.

[206] Bruce Schneier and Doug Whiting. Fast software encryption: Designing encryption algorithms for optimal software speed on the Intel Pentium Processor. In Eli Biham, editor, *Proceedings of the 4th International Workshop*

on Fast Software Encryption, volume 1267 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[207] Rick Schroeppel. The Hasty Pudding Cipher, 1999. Available at http://www.cs.arizona.edu/ rcs/hpc/.

[208] Jennifer Seberry, XianMo Zhang, and Yuilang Zheng. On constructions and nonlinearity of correlation immune functions. In Tor Helleseth, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 181–199. Springer-Verlag, 1993.

[209] Adi Shamir, Jacques Patarin, Nicholas Courtois, and Alexander Klimov. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - Proceedings of EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag, 2000.

[210] Takeshi Shimoyama, Masahiko Takenaka, and Takeshi Koshiba. Multiple linear cryptanalysis of a reduced round RC6. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 76–88. Springer-Verlag, 2003.

[211] Takeshi Shimoyama, Hitoshi Yanami, Kazuhiro Yokoyama, Masahiko Takenaka, Kouichi Itoh, Jun Yajima, Naoya Torii, and Hidema Tanaka. The block cipher SC2000. In Mitsuru Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 312–328. Springer-Verlag, 2001.

[212] Thomas Siegenthaler. Correlation Immunity of Nonlinear Combining Fnctions for Cryptographic Applications. *IEEE Transactions on Information Theory*, 30(5):776–780, September 1984.

[213] Thomas Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, C-34(1):81–85, January 1985.

[214] Leonie Simpson. *Divide and Conquer Attacks on Register Based Stream Ciphers*. PhD thesis, Information Security Research Centre, Queensland University of Technology, January 2000.

[215] Softforum. Block cipher proposal of XENON to ISO/IEC/JTC1/SC27, 2001. Available at http://www.softforum.co.kr/files/board/download/Xenon_SWIST.pdf.

[216] Softforum. Block cipher proposal of ZODIAC to ISO/IEC/JTC1/SC27, 2001. Available at http://www.softforum.co.kr/files/board/download/Zodiac_SWIST.pdf.

[217] Adam Stubblefield, John Ioannidis, and Avi Rubin. Using the Fluhrer, Mantin, and Shamir Attack to break WEP. Technical report, TD-4ZCPZZ AT and T Labs Technical Report, 2001.

[218] Makoto Sugita, Kazukuni Kobara, and Hideki Imai. Security of reduced version of the block cipher Camellia against Truncated and Impossible Differential Cryptanalysis. In Colin Boyd, editor, *Advances in Cryptology - Proceedings of Asiacrypt 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 2001.

[219] Yukiyasu Tsunoo, Hiroyasu Kubo, Hiroshi Miyauchi, and Kazuo Nakamura. A new 128-bit block cipher CIPHERUNICORN-A. Technical report, The Institute of Electronics, Information and Communication Engineers, 2000. ISEC2000-5.

[220] David Wagner. Weak keys in RC4. Posting to sci.crypt usenet group on September 26, 1995.

[221] David Wagner. The boomerang attack. In Lars Knudsen, editor, *Proceedings of the 6th International Workshop on Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer-Verlag, 1999.

[222] David Wagner, Niels Ferguson, and Bruce Schneier. Cryptanalysis of FROG. In *Proceedings from the Second Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, March 1999. Available at http:/csrc.nist.gov/encryption/aes/.

[223] Dai Watanabe, Soichi Furuya, Hirotaka Yoshida, and Kazuo Takaragi. MUGI psuedorandom number generator, self evaluation, 2001. Available at http://www.sdl.hitachi.co.jp/crypto/mugi/index-e.html.

[224] Dai Watanabe, Soichi Furuya, Hirotaka Yoshida, and Kazuo Takaragi. A new keystream generator MUGI. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 2003.

[225] Hongjun Wu. Cryptanalysis of Stream Cipher Alpha1. In Lynn Batten and Jennifer Seberry, editors, *Proceedings of Information Security and Privacy - 7th Australasian Conference, ACISP'02*, volume 2384 of *Lecture Notes in Computer Science*, pages 169–175. Springer-Verlag, July 2002.

[226] Hongjun Wu. Related-cipher attacks. In Robert Deng, Sihan Qing, Feng Bao, and Jianying Zhou, editors, *Information and Communications Security - 4th International Conference*, volume 2513 of *Lecture Notes in Computer Science*, pages 447–455. Springer-Verlag, 2002.

[227] Hongjun Wu. A New Stream Cipher HC-256, 2004. Available at http://eprint.iacr.org/2004/092.pdf.

[228] Wenling Wu and Dengguo Feng. Linear cryptanalysis of the NUSH block cipher. *Science in China series*, 45(11):59–67, February 2002.

[229] Hitoshi Yanami, Takeshi Shimoyama, and Orr Dunkelman. Differential and linear cryptanalysis of a reduced-round SC2000. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 34–48. Springer-Verlag, 2003.

[230] Yongjin Yeom, Sangwoo Park, and Iljun Kim. On the Security of CAMELLIA against the Square Attack. In Joan Daemen and Vincent Rijmen, editors, *Proceedings of the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 89–99. Springer-Verlag, 2003.

[231] Eric Young. Re: More LTC timings... Posting to sci.crypt usenet group on 18 June, 2003.

[232] Amr Youssef and Stafford Tavares. On Some Algebraic Structures in the AES Round Function. Cryptology ePrint Archive, Report 2002/144, 2002. Available at http://eprint.iacr.org/.