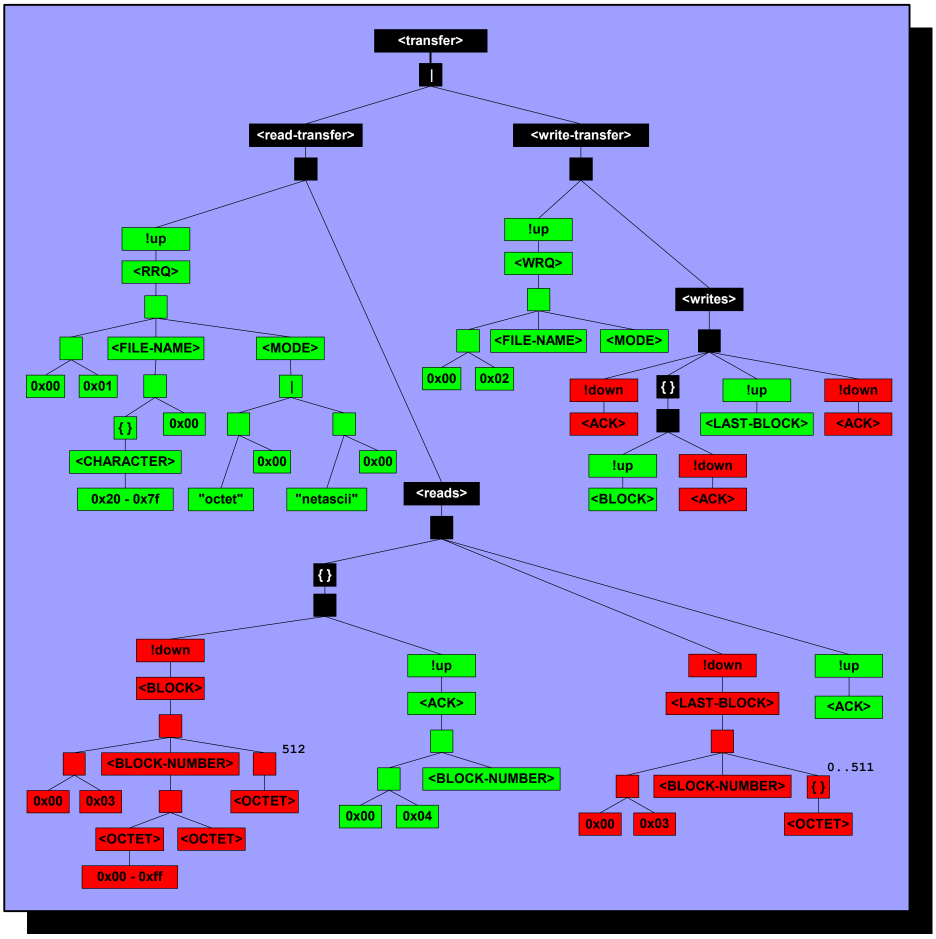
4 4 8

V T T   P U B L I C A T I O N S

Rauli Kaksonen

A Functional Method for Assessing Protocol Implementation Security



TECHNICAL RESEARCH CENTRE OF FINLAND ESPOO 2001

VTT PUBLICATIONS 448

A Functional Method for Assessing Protocol Implementation Security

Rauli Kaksonen

VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 2001

ISBN 951–38–5873–1 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5874–X (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Tietoliikennejärjestelmät, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Telekommunikationssystem, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Telecommunication Systems, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Leena Ukskoski

Otamedia Oy, Espoo 2001

Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security. Espoo. Technical Research Centre of Finland, VTT Publications 448. 128 p. + app. 15 p.

Keywords information security, automated testing, software quality, implementation vulnerabilities, programming mistakes, mini-simulation method

Abstract

Serious information security vulnerabilities are discovered daily and reported from already deployed software products. Customers have no feasible means for estimating the security level of the products they purchase. The few generally applicable methods require the source code, which is often not delivered with a product. Many of the reported vulnerabilities are robustness problems. Robustness can be functionally assessed without the source code by injecting anomalies, unexpected input elements, to the tested component. The component passes the tests if it can securely handle the injected anomalies.

The methods generally applied for software testing and modelling were found to be too complex and rigid for functional robustness assessment. A new mini-simulation method using attribute grammar to model both input syntax and software behaviour was proposed. Means for the systematic creation of a large number of test cases was presented. The method was used to test the robustness of 49 software products. A total of 40 tested products were found to be vulnerable to denial-of-service problems, and 14 of them were proven to contain vulnerabilities making it possible to execute remotely supplied code on the host system.

Applications of the method include quantitative comparisons and the benchmarking of software components, but it has some limitations. The proportion of the flaws found using the method compared to the actual number of flaws is difficult to assess and the tests may favour some components over others. However, if the method can help to eliminate the most obvious vulnerabilities, it would be much more difficult to find serious flaws using unsystematic methods. This could cut down on the number of publicly disclosed vulnerabilities.

Preface

This publication presents some of the results from the work done in the project *Security Testing of Protocol Implementations* (PROTOS) during 1999–2001. At the same time this publication acts as my licentiate thesis. The supervisor of the thesis was professor Juha Röning from the Department of Electrical Engineering of the University of Oulu. The thesis was inspected by professor Veikko Seppänen. Professor Petri Mähönen supported the work.

Back in 1998 the inspiration for the PROTOS-project was the large number of security problems found from software on a daily basis. It seemed clear that at least some of these problems could be revealed using functional testing, but there did not seem to be much research on the topic. The PROTOS-project was initiated in the beginning of 1999 as a joint effort of the University of Oulu and VTT Electronics. The original industrial partners were Nokia Networks and Oulun Puhelin, later Nokia Mobile Phones joined the project. The main part of the project funding was provided by the National Technology Agency (Tekes).

The PROTOS-project concept was originally proposed by Marko Laakso, who is the founder of the *Oulu University Secure Programming Group* (OUSPG). He and Ari Takanen, also from OUSPG, have been participating in the PROTOS project since its beginning. In VTT the PROTOS-project has been supported by Marko Heikkinen, Jussi Paakkari, and Aija Kotila. Virtually all OUSPG employees have been involved in the testing conducted on the PROTOS-project: Juhani Eronen, Mikko Hiltunen, Jani Kenttälä, Jarkko Lämsä, Tomi Nylund, Mikko Varpiola, and Joachim Viide, as well as exchange students Raymond Hofman, Christian Wieser, and Alexander List. Antti Häyrynen from Nokia Mobile Phones and Jarmo Mustonen, Teemu Särkelä, and Mikko Tienhaara from Nokia Networks have also taken part in the testing activities.

Personally I have been responsible for the development of the mini-simulation method and implementation of the prototype toolkit, called the “Bugbear”. In that work I have exploited ideas and experiences from the other PROTOS-project participants. For the results-part of this study, I have used the material from the tests done in the OUSPG. In my mind the PROTOS-project has succeeded in providing useful ideas and new information for the public good, like the material presented in this publication. However, giving out information about security holes is always a two-edged sword, since the pieces of information may also give new arsenal to “crackers”. Still, I strongly feel that it is good to bring the issues out in the open: The public must know that software does contain vulnerabilities. Developers must know what causes the vulnerabilities and how to avoid them. Testers must have methods and tools to search for the vulnerabilities.

I thank all the aforementioned people for their contribution to this study, which quite incorrectly carries my name only. I also thank Zach Shelby for the final proof-reading of the text. Finally, I must thank by parents, Aune and Veijo Kaksonen, for their support since the beginning.

Oulu, 28th September 2001

Rauli Kaksonen

Contents

List of Symbols

ACM	Advanced Computer Machinery
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
AVA	Adaptive Vulnerability Analysis
BER	Basic Encoding Rules
BNF	Backus Naur Form
CTMF	Conformance Testing and Methodology Framework
DOM	Document Object Model
DTD	Document Type Declaration
EPA	Extended Propagation Analysis
FIST	Fault Injection Security Tool
FSM	Finite State Machine
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force

IP	Internet Protocol
IUT	Implementation Under Test
LOTOS	Language for Temporal Ordering Specification
LDAP	Lightweight Directory Access Protocol
MSC	Message Sequence Chart
Perl	Practical Extraction and Reporting Language
PCO	Point of Control and Observation
POSIX	Portable Operating System Interface
PDU	Protocol Data Unit
PROTOS	Project: Security Assessment of Protocol Implementations
RTAG	Real-Time Asynchronous Grammar
SDL	Specification and Description Language
SNMP	Simple Network Management Protocol
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTCN	Tree and Tabular Combined Notation
UDP	Unreliable Datagram Protocol

UML	Unified Modelling Language
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WMLC	Wireless Markup Language Compiled
WSP	Wireless Session Protocol
WWW	World Wide Web
XML	Extensible Markup Language

1. Introduction

The infrastructure of modern society relies heavily on computer systems and networks for most of its critical functions. Emerging e-commerce and other applications rely on the trusted handling of digital information. The failure of software is a great risk for this infrastructure, either through inadvertent failures or by malicious attacks. The latter risk has become more severe under recent years since the Internet connection has lowered both the financial and intellectual barriers to launch effective attacks against critical systems. Furthermore, systems previously protected from external access are being equipped with network connectivity, exposing them to external intrusion attempts.

The security of a computer system is often mistakenly described only by the properties of its cryptographic algorithms, e.g. key lengths [61]. In practice most security compromises bypass cryptographic protections by exploiting weaknesses in the system software. This conclusion can be drawn, e.g. by looking at the annals of the *BugTraq* mailing list dedicated to the public disclosure of information security vulnerabilities [14]. The rate at which vulnerabilities are reported to the public is high. This causes a continuous stream of updates and security advisories. Administrators have difficulty keeping systems up-to-date and serious vulnerabilities remain even though there are fixes available [48]. The problem is more severe for home users, who lack the time or expertise to continuously maintain their systems.

Instead of considering security as something we can add into a system as an afterthought, we should view security as a *quality attribute* determined during development work [22]. Software *vulnerabilities*, security-critical flaws in software, can be introduced during both the design and implementation of a product [61]. *Design vulnerabilities* can result from a poor design caused by a lack of expertise, e.g. a bad encryption algorithm is selected. An unfortunate trend is to ship products with the default configuration set to minimal security and leave the hardening to the users or administrators. In many cases decisions leading to vulnerabilities are made to add novel features, e.g. executable scripts are embedded into e-mail messages [62]. It is not easy to say when a feature is a vulnerability and when it is a reasonable design choice [30, p. 127]. Sometimes a feature is highly desirable, sometimes the dangers it conveys cannot be justified.

Modern software products tend to be complex. A complex system is difficult to implement and use [5]. High assurance of the correctness of a simple system is easier to get than that of a complex one [30, p. 14]. A simpler system is also more likely to be used appropriately. Still, complexity can be justified if no simple solution fulfils the system requirements.

Implementation vulnerabilities are introduced during the implementation phase [32; 61]. They are security hazards resulting from programming mistakes. Even if the software design is flawless and used protocols and algorithms provide strong theoretical security, implementation vulnerabilities can make a software component vulnerable. A design may be formally modelled and proven to contain some desirable security properties, but implementation complexity greatly exceeds the capabilities of formal analysis [22]. The more complex a product is, the more likely it is to contain implementation mistakes [63]. An intruder who has access to a vulnerable software component through, e.g. a network interface, can exploit the vulnerability to compromise the computer system.

Implementation vulnerabilities are caused by dangerous program constructs. They often have directly observable *failure modes* like crashes or hangs resulting from corruption of the internal state [42; 50]. Many implementation vulnerabilities can be unambiguously detected, in contrast to design problems which are on a more abstract level and open for interpretation. The automatic detection of implementation vulnerabilities should be possible, at least to a limited extent. Some tools for this do exist, but many require access to the source code [64; 74, p. 226; 81]. Functional assessment is possible without the source code, but the existing tools are few in numbers and limited in scope [42; 50; 53]. Only a few generally applicable functional security analysis tools seems to be available.

Security has not sold products in the past. Time-to-market, new features, and performance have been more attractive properties [19]. Even if a customer considers security as a requirement and is willing to pay for it, there have been only a few feasible means for assessing the true security of the product. The existing methods usually rely on source code. In many cases the source code is not part of the delivered package and the buyer is at the mercy of the vendor.

The purpose of this study is to present a new assessment method for finding software implementation vulnerabilities through functional testing. The source code is not required and no co-operation with the software vendor is needed to analyse a product. The focus of the study is on the method itself, vulnerabilities, design of effective tests, and the treatment of vulnerabilities are only briefly introduced for understanding.

Chapter 2 gives an overview to implementation vulnerabilities. It is followed by a summary of software assessment techniques in chapter 3 and behavioural modelling methods in chapter 4. In chapter 5 a new method for software modelling is introduced and further refined in chapter 6 to be used for vulnerability assessment. Chapter 7 show results from tests using the method. Finally, an analysis of the proposed method and a summary are given.

2. Implementation Vulnerabilities

Implementation vulnerabilities result from mistakes made by software programmers. There are no reliable means for preventing these mistakes from remaining in software and current software error rates are high [11]. Some of the mistakes inevitably lead to vulnerabilities. They exist despite the use of rigorous software processes, maturity models, and formally verified protocols [74, p. 14]. Security is hard to measure and the security properties of many contemporary software products are largely unknown [26].

Compared to normal implementation flaws (or faults), implementation vulnerabilities have some unique properties. Normal flaws prevent applications from functioning correctly and are observable by the users. Vulnerabilities enable an attacker to gain privileges or interfere with proper functionality. They do not manifest themselves during everyday operation and can lay dormant for years. Still, after a vulnerability is exploited there is a sudden need to get it fixed.

A program or procedure taking advantage of a vulnerability is called an *exploit*. Exploits may have varying impacts from denial-of-service to the total compromise of a system. A vulnerability in a shipped software products requires *patching* if the implications of the vulnerability cannot be tolerated. A *patch* must be created, tested, distributed, and deployed to installed systems before the problem is fixed [44]. This may be an expensive process for both vendors and customers. Many systems are left vulnerable since their administrators lack expertise and/or time for continuous patching. This is especially true for home users. Embedded systems without upgrade capability cannot be patched at all.

2.1 Types

There are many different types of implementation vulnerabilities. Common mistakes include failure to verify the validity of input from a mistrusted source, use of an insecure library function, or use of the function in an insecure way. An interested reader should consult some of the available secure programming resources for further information about implementation vulnerabilities and avoiding them [3; 10; 70; 80].

The C and C++ programming languages are especially problematic from the security point of view since it is relatively easy to write insecure code inadvertently. Careless use of many standard library functions, e.g. `gets`, `sprintf`, and `strcpy`, opens up security holes. Programmers make these mistakes because of a lack of knowledge or they prefer the easiest way despite of the hazards [73]. Despite the programming language used an implementation vulnerability can exist in an employed external component.

A common and serious vulnerability type is *buffer overflow*, but other types of vulnerabilities exists as well. A buffer-overflow vulnerability is a failure to ensure that a processed chunk of input data fits into the space reserved for it [13; 28; 43; 66]. The oversized input chunk overwrites the memory content beyond the reserved buffer. Arbitrary input usually leads to a crash of the program, but carefully constructed data embedded into the input chunk may be executed with the privileges of the victim process. Instructions for exploiting buffer overflows are publicly available [72]. Enough details for exploiting a buffer overflow vulnerability can be obtained without access to the source code. When present, buffer overflows are often exploitable.

2.2 Vulnerability Management

Overall there are three alternatives for managing software faults proactively, before taking a system into use: *fault avoidance*, *fault elimination*, and *fault tolerance* [74, p. 40]. Similar categorisation may be applied for vulnerability management as well, for this study we define the following vulnerability management activities:

1. *Vulnerability avoidance*: Developing a software component using techniques which prevent the introduction of (specific types of) vulnerabilities into the component.

2. *Vulnerability elimination*: Searching for the vulnerabilities from a component using testing or other activities and removing the problems.

3. *Vulnerability tolerance*: Building tolerance to the (potential) vulnerabilities in a component and ensuring that acceptable results are produced despite of them.

As this study is concerned with implementation vulnerabilities, the following discussion is limited to them.

2.2.1 Avoidance

Vulnerability avoidance means the use of an implementation technique which does not suffer from a particular type of vulnerability at all. Completely avoiding some vulnerability types leaves more resources to deal with other kinds of vulnerabilities.

For example, a Java *virtual machine* provides a protected execution environment for programs. Effects of the failure of a program is limited by the actions allowed by the virtual machine [19]. Java programs do not have, e.g. buffer-overflow problems, since all memory access is controlled by the virtual machine. Java also contains a *bytecode verifier*, which enforces the integrity properties of Java programs. Still, all kinds of faults in the virtual machine itself are possible.

Buffer-overflows, illegal memory accesses, and memory leaks are major problems in C and C++ software. A compiler can embed integrity checks into programs for invalid variable access and perform bounds checking for memory accesses [57; 40; 20]. Unfortunately such extra precautions have in most cases a negative performance impact [13]. Problems can be avoided also by introducing *coding conventions* which deny the use of vulnerable functions, e.g. `strcpy`, and mandate other safe programming practices. The supervision of such rules requires inspections or reviews.

2.2.2 Elimination

Vulnerability elimination means the identification of vulnerabilities from software and fixing them. Elimination must include searching for vulnerabilities and can only address problems which are found. The elimination of vulnerabilities before shipment is cheap compared to the patching of deployed products [44]. Elimination of all flaws is not usually possible or cost effective. At some point the product must be shipped despite that it still may contain vulnerabilities. The shipping point must be based on cost estimations of using more resources to eliminate the existing vulnerabilities compared to fixing them after shipment.

The finding of the vulnerabilities is problematic. The size and complexity of code is proportional to the number of faults it may contain [63]. An elimination process can be promoted by limiting the amount of code in a system, which cuts down the number of potentially vulnerable locations. The amount of critical code is limited if systems are made as simple as possible and all non-essential software is removed [19].

2.2.3 Tolerance

Vulnerability tolerance strategies recognise that software does have vulnerabilities and tries to limit their impact to system security. Tolerance is inferior for avoidance and elimination in the sense that it requires the addition of some kind of *protection system*. The protection system brings complexity which has a negative impact to performance and cost. Ironically enough, the protection system itself may contain vulnerabilities. Still, tolerance often is the only available approach if the system is built from external components of which security cannot be ensured [19]. *Defensive programming* assumes that flaws may exist in the developed components and attempts are made to detect problems and remedy the detected inconsistencies [68, p. 302].

The access of hostile entities to a computer system can be limited by *firewalls* and *wrappers* [19]. Firewalls allow only limited communication between the external world and the system internals. A wrapper inspects input to a protected component and blocks any suspicious interactions before the component is corrupted. Execution of the component may also be terminated if a vulnerable condition is observed. For example, integrity checks could be included into vulnerable library functions to prevent buffer-overflows [13]. Alternatively the operating system could be modified to have a *non-executable stack* [83]. Finally, an *intrusion detection system* (IDS) can be used to detect ongoing attacks by monitoring network and host activity [7]. Still, no bullet-proof solutions seem to exists as different exploitation strategies are devised to bypass the protection systems.

The impact of software flaws in a component can be reduced by limiting the access rights of the component to the minimum required. Abuse of the component is limited by the access rights. The principle is not always followed, e.g. many UNIX programs are running unnecessarily as *root* [25]. Root programs have all privileges and the potential to do anything (good and evil) in the system. As said earlier, Java and other virtual machines protect against a set of problems [71]. The security features of Java allow non-trusted programs to be executed under tighter restrictions. For example, Java *applets* loaded from the Internet are not allowed to access critical local resources. In *Practical Extraction and Reporting Language* (Perl), a mechanism called *tainting*, tries to ensure that input provided to a program is not used insecurely [55]. Input data is tainted and attempt to use it in a vulnerable fashion, e.g. as a name of a file opened for writing, is aborted. This prevents an intruder from exploiting the vulnerability.

2.3 Vulnerability Analysis

Vulnerability elimination and vulnerability tolerance activities require information about the location of the vulnerabilities. Vulnerabilities can be searched individually from components or from an aggregate system.

The software components themselves are increasingly built from sub-components and the aggregated behaviour may lead to insecurities, even if each component appears to be free of vulnerabilities. However, a secure system is easier to build from secure components than from insecure components [27]. The vulnerability analysis can be done statically from source code (sometimes from object code) or dynamically by observing an executing component or system. Time of the analysis may be before or after the release of the product or activation of the system.

2.3.1 Component Analysis

The traditional software project activities for achieving a high quality product, promote discovery of the implementation vulnerabilities as well. Inspections, reviews, conformance testing, structural testing, coding conventions, etc. are all useful and required for the production of high quality software.

Source code inspections can be automated into tools searching for vulnerable constructs [73; 75]. For example, *ITS4*, *Flawfinder*, and *RATS* are tools for scanning C and C++ source code [17; 64; 81]. One problem is that the scanning tools tend to create many *false positives*, i.e. alarms for statements which do not contain real vulnerabilities. There are also problems which the source code scanners do not find.

Traditional testing tends to be concerned only whether the software component performs the behaviour specified for it [29]. Testing for the ability of the component to stand anomalous events and malicious attacks requires a different approach. *Fault-injection* analysis purposefully injects faults into a system and observes the resulting behaviour [26; 27; 28]. Software fault injection mutates the source code, or data at software interfaces. The former requires access to the source code, but the latter does not. Fault injection gives information about *robustness*, the ability of software to function correctly in the presence of invalid inputs or stressful environmental conditions [33, p. 64]. Robustness is especially important in safety-critical and security-critical applications [74, pp. 45, 275].

There are few tools available for robustness evaluation through software interfaces. Three tools are given as examples.

- The *Fuzz* tool is used for testing the robustness of different UNIX utility commands [50]. The tested programs were subjected to random input. Somewhat comparable tests were conducted in 1990 and 1995, results indicate improved robustness. Still, significant failure rates of up to 40% were observed in some categories.

- The *Ballista* tool has been used to test the ability of POSIX-compatible operating systems to handle exceptional input parameters in system calls [42]. The selection of input was not random, but based on the type and meaning of the call parameters. The tests consisted of over 1 million tests cases and resulted in failure rates were between 10% to 23% depending on the operating system. Also Microsoft Windows systems have been tested with Ballista [65].

- The *IP Stack Integrity Checker* exercises the robustness of the IP stack and related stacks [53]. The tool creates pseudo-random packets which are then fed into the tested stack.

Ghosh proposes a *Component Security Certification* pipeline for a comprehensive analyse of the security of a software component [27]. The pipeline includes test planning, structural testing, and functional testing activities. As a result the component is assigned a rating based on metrics collected throughout the analysis pipeline.

2.3.2 System Scanning

Vulnerabilities can be searched also from a deployed system. In *tiger-team testing* a group of experts attempt to break into an installed system [74, p. 230]. A vulnerability is found if the break-in succeeds. The problem is that the tiger-team testing is manual work based on the experience of the team, rather than a systematic and reproducible effort. Tiger teams do also vary in their point of view and expertise.

Security scanner tools systematically scan hosts of the inspected system for known vulnerabilities [7]. These tools capture the tiger team testing process [74, p. 230]. Normally a security scanner probes only for known vulnerabilities, vulnerabilities unknown by the scanner are not noticed.

3. Assessing Software Components

Software component assessment probes whether a software component meets the stated requirements or holds some other desirable properties. Security is one such property of a software component. In the following some traditional and non-traditional assessment techniques are described and their relevance for security analysis is discussed. The discussion is limited to the testing of an already implemented software component, the *implementation under test*, IUT.

3.1 Software Testing

The IEEE Standard Glossary of Software Engineering Terminology defines *testing* as [33, p. 76].

> *The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.*

It is important to note that the testing is defined to evaluate some aspect of the tested system or component, rather than vague "goodness" or "badness". A test should always have a clear purpose, which exactly defines the aspects of the IUT that must be assessed.

The following overview to software testing is based on books from Beizer and Baumgarten [9; 8]. Some terms are from the IEEE glossary [33].

3.1.1 Basic Concepts

The IEEE glossary defines the terms *mistake*, *fault*, and *failure* [33, p. 31]. A developer may make a mistake which causes a fault, which is an incorrect step, statement or data definition. A failure is the inability of a software to perform its required functionality. Alternatively the term *flaw* is used instead of the term fault.

The fundamental unit of testing is a *test case*. A test case demonstrates certain behaviour or the realisation of some capability [8, p. 71]. Test cases are grouped into *test group*s and further into *test suites*. A test case has a specific *test purpose* and a group may have a *test group objective*. The result of a test case is called the *verdict* [8, p. 32]. It is either *pass*, *fail*, or *inconclusive*. In a test case with the pass verdict the tested component fulfils the test purpose and displays only valid behaviour. In a test case with the fail verdict some requirement is violated, i.e. a failure is observed. Inconclusive means the inability to assign either the pass or fail verdict.

An *oracle* specifies the expected outcome for a test case [9, p. 23]. An *input/output oracle* specifies the expected output from the IUT. An oracle may be created manually during test design. An automated oracle can be a simplified implementation, an older version of the same program or a program from another source. The test output of an initial program version can be collected and manually validated, and the testing of subsequent versions can be done against the collected data.

Testing can be divided into *structural testing* (*white-box testing*) and *functional testing* (*black-box testing*) [9, p. 10]. In functional testing software internals are not considered. Test input is constructed and the output verified solely according to the specified behaviour. Structural testing is based on implementation details, such as source code. Structural testing and functional testing complement each other [9, p. 10]. Functional testing is theoretically sufficient, but it may take an infinite time to detect all flaws since the analysis is limited to the software interfaces. Complete (in some sense) structural testing takes a finite time, but cannot address problems which are visible only during execution.

Static analysis is structural testing conducted while the tested component is not running [9, p. 8]. Modern compilers perform a variety of static analysis, e.g. type checking. *Dynamic analysis* is done while the tested component is executing [9, p. 74].

3.1.2 Testing Process

Figure 1 shows the *V-model* of the software development process and information flows within the process. The model combines the software construction process from the *waterfall model* with the testing process [68, pp. 5, 376]. In the presented V-model software construction is divided into *requirement analysis*, *architecture design*, *module design*, and *programming*. Each phase uses specifications provided by the earlier phase. Development-time testing of the product is divided into *unit testing* (*component testing*), *integration testing* and *system testing* [9, p. 20]. Unit testing addresses the individual program components. Integration testing proceeds with checking the co-operation of components. The overall system conformance to the architectural design is assured during system testing. The end product is evaluated against the requirements during *acceptance testing* [9, p. 16].

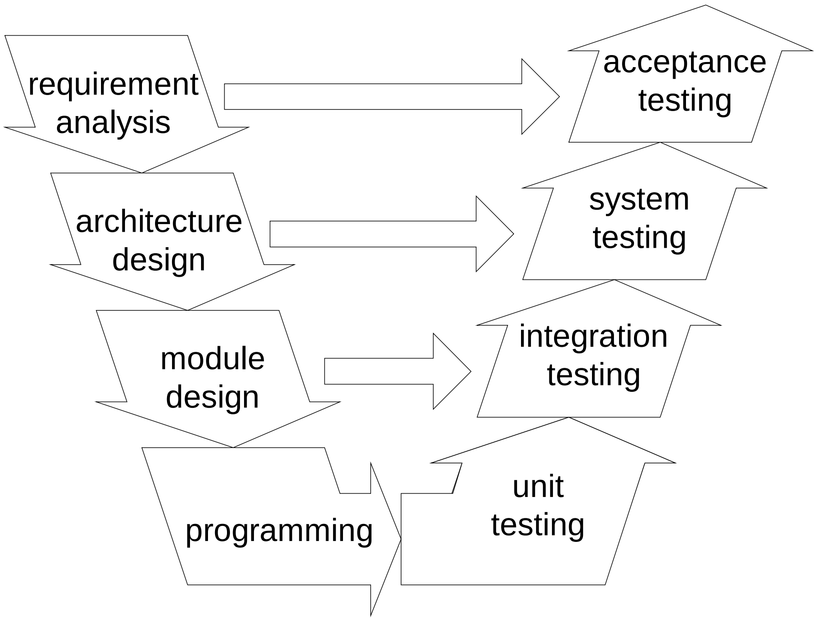


Figure 1. V-model for software development.

Structural testing tends to dominate the earlier testing activities, especially unit testing [9, p. 428]. Unit testing is usually done by the programmers themselves, who are familiar with the program structure. Also, the amount of code in the program units is relatively small, so a high code coverage up to 100% is feasible and desirable [9, p. 75]. The later testing phases, like system testing, usually takes advantage of functional testing techniques [9, p. 428].

3.1.3 Functional Testing

Functional testing is based on the specification, not program internals [9, p. 10]. It is applicable to all software components regardless of whether they are white-box or black-box components. In this respect it is an universal approach.

Functional testing should exercise all required functionality of the tested component. The functionality can be modelled using a special *graph*, a *transaction flowgraph*, drawn from the software specification. This graph defines the expected behaviour of the component using *nodes* and *edges* [9, p. 121]. A node stands for an identifiable state or status of the modelled entity, an edge is a labelled transfer from one node to another. A single sequence of events, a *path*, forms a route through the graph. Each execution of the component selects one path. Usually testing is designed to *cover* the graph using some coverage criteria. For example, a test suite can be designed to exercise all edges of the graph at least once.

Figure 2 shows a simple graph, with the nodes numbered from 1 to 9 and the edges labelled with event names. Input events are underlined to separate them from output events. The graph is drawn from the specification of the *Trivial File Transfer Protocol* (TFTP)[1] server [67]. The protocol provides a simple method for transferring files between a TFTP server and client. For brevity, the TFTP protocol error handling and timeout mechanism is omitted. TFTP is intended to be a continuously running service, but the graph represents only a single transfer, either a file read or a file write.

[1] TFTP is chosen to be the example protocol of this study, it is used in this and the following chapters. TFTP was chosen because of its simplicity, the available space is enough to present meaningful TFTP sessions.

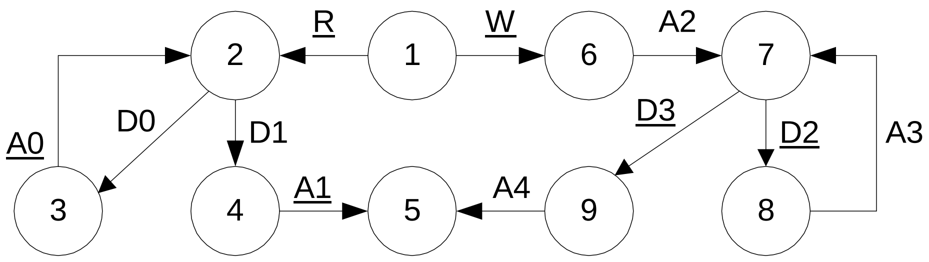


Figure 2. TFTP transfer without error handling.

A TFTP transfer is initiated by the client. The client requests either a read or write transfer from a TFTP server. In a read transfer the client downloads a file from the server. In a write transfer the client uploads a file to the server. A file is transferred in data blocks, the size of all but the last one being 512 octets[2]. The last data block is shorter than 512 octets, possibly zero octets. Each data block is acknowledged before the next one is sent. Table 1 explains the edge labels used in Figure 2.

Functional testing should exercise the specified functionality and verify that is it correctly implemented. A simple coverage criteria would be to cover each edge at least once. Paths (**R D0 A0 D1 A1**) and (**W A2 D2 A3 D3 A4**) cover the graph in Figure 2.

A graph can be represented as a *path expression* [9, p. 243]. For example, the graph shown in Figure 2 is represented by the path expression (1). In path expressions braces are used to group nodes or paths. The "+" sign separates alternative nodes or paths. An exponent following a node or a path indicates a loop in the graph. In the expression (1), exponents *n* and *m* indicate potentially unlimited looping.

$$\mathbf{\underline{R}(D0\ \underline{A0})^{n}\,D1\ \underline{A1} + \underline{W}\ A2(\underline{D2}\ A3)^{m}\,\underline{D3}\ A4} \qquad (1)$$

[2] Octet is 8-bit byte of data.

Table 1. Edge labels of the graph in Figure 2.

Label	Explanation
R	Client requests a file read.
D0	Server sends a 512-octet data block.
A0	Client acknowledges the previous 512-octet data block.
D1	Server sends the final data block, less than 512 octets.
A1	Client acknowledges the final data block.
W	Client requests a file write.
A2	Server acknowledges the readiness to receive.
D2	Client sends a 512-octet data block.
A3	Server acknowledges the previous 512-octet data block.
D3	Client sends the final data block, less than 512 octets.
A4	Server acknowledges the final data block.

3.1.4 Domain Testing and Syntax Testing

In *domain testing* the test cases are based on the specification of input values accepted by a software component [9, p. 173]. The possible input values are divided into *input domains* which are ranges of values treated uniformly. The component is likely to process values in one input domain similarly and the tests should sample each domain at least once. The tester should especially concentrate on domain boundaries, since they are the more likely to be misunderstood or badly handled.

Usually tests verify the correctness of the IUT responses for valid input. As a complementary approach one should also verify the responses for illegal input, i.e. “garbage” [9, p. 284]. Systems that have an interface to a potentially hostile environment should always perform strict input validation. Unless this is done, garbage input may drive the system to an invalid internal state.

Syntax testing is an another technique for generating test cases from the input specification. In syntax testing the tester uses a syntax description of the *input languages* accepted by the IUT [9, p. 284]. Virtually all input can be modelled as a language and syntax testing is usually applicable for a software interface. The syntax testing cases assess the input validation routines of the software components by erroneous input elements. Beizer proposes the following kinds of erroneous elements [9, p. 296].

1. Elements with illegal contents, i.e. “garbage”.

2. Misplaced and missing elements, elements in wrong order, too many elements.

3. Correct element, but in wrong context.

4. Illegal delimiters (such as spaces or braces), if delimiters exists between elements. Apply all presented faults to delimiters as well, try also unbalanced braces, if possible.

5. Broken inter-element dependencies. If a value of one element is used in another elements as reference, design cases where this dependency is broken.

6. Empty input, only the end of input character, if any.

The best test cases are those which are difficult to categorise either as valid or invalid [9, p. 299]. A programmer has had a hard time determining what to do with them, and she or he may have made mistakes there. For effective tests all other parts of the input data, except the error element, should be legal [9, p. 297].

3.1.5 Conformance Testing

The OSI *conformance testing methodology and framework* (CTMF) defines a methodology for the conformance testing of protocol implementations [38; 8, p. 15]. The purpose of conformance testing is to ensure that the behaviour of a protocol implementation fulfils the standardised conformance requirements.

Conformance testing uses the functional testing approach, because the same tests must be applicable to test different implementations from different vendors.

Conformance testing can be done by a *third* party: an entity independent from the software provider (the *first party*) and procurer (the *second party*). Before third party testing the provider claims the conformance of a product to the protocol by creating a *conformance statement*. The statement describes the functionality that is supposed to meet the conformance requirements. The third party does the actual testing and verifies whether the product fulfils the conformance requirements or not.

Interoperability tests are different from conformance tests [8, p. 19]. In the interoperability testing two or more different implementations from potentially different vendors are used together to see if and how they interoperate. The interoperability tests may reveal misunderstandings about the protocol specification, especially if the used conformance tests are not extensive or the protocol specification is ambiguous.

3.1.6 Structural Testing

Structural testing is based on implementation details, such as source code [9, p. 11]. Sometimes the structural methods can be applied to the object code of a component as well [13].

Statements and branches form the *control flowgraph* of component [9, p. 60]. In *control flowgraph testing* the test cases are designed to cover the control flowgraph according some coverage criterion [9, p. 72]. For example, all statements and branches from the tested component must be executed at least once. In *data-flow testing* the test design is based on a *data flowgraph* [9, p. 145]. The data flowgraph shows the life-cycle and dependencies of the data objects in a component. The tests are designed to find *data-flow anomalies*, i.e. flaws in the manner the component is handling data objects. Modern programming language compilers perform versatile data-flow analysis [9, p. 156].

Property-based testing concentrates the testing effort to the important parts of the tested component rather than to full coverage [25]. Property-based testing takes advantage of program *slices*. A slice is a truncated version of the program which has the same behaviour as the full program in respect to the tested property. The slice is considerably smaller than the original program which greatly reduces the testing effort. The observed behaviour of the slice during tests is evaluated against the test purpose. The assumption is that the testing of the slice is equal to the testing of the whole program, as far as the tested property is concerned. Property-based testing has been applied to white-box security analysis [25].

3.2 Fault Injection

A real-world system is always likely to contain faults. Rather than (desperately) trying to remove the faults, the aim can be the building of a system which provides the required service despite them. *Fault injection* purposefully injects artificial faults into a system for analysis purposes. Fault injection has been proposed to be used to understand the effects of real faults, to get feedback for system correction or enhancement and to forecast the expected system behaviour [16].

Fault injection can be applied to both hardware and software [16]. In hardware *transient faults* change system state but hardware is not damaged, but *permanent faults* are irreversible and hardware must be replaced to fix them. In contrast, software faults result only from incorrect design or implementation. The fault is always there, but it may require a rare condition to cause a failure. The fault may be latent in a software for years, emerge once or a few times and then disappear again. Such faults are extremely difficult to pin-point and repair.

It is relatively straightforward to validate system behaviour in the presence of a foreseen set of faults [16]. The faults are injected into the system and the behaviour is monitored. The evaluation of the system behaviour more broadly is difficult, since the actual faults that are going to occur during the system life-cycle are unknown.

3.2.1 Injection

Software fault injection analyses what happens after an *anomaly* [74, pp. 26, 201]. An anomaly is an unexpected event that appears to have the potential to alter software behaviour through the alteration or corruption of its internal state [74, p. 40]. Fault injection provides information on the behaviour of the software component when faults do happen. This enables the analyser to know to some extend how the component will behave when it faces unknown situations.

In *software fault injection*, anomalies are injected into a software component which is executed [74, p. 45]. The anomalies can be injected to the component inputs, or directly into the component code or data structures. When only inputs are used, then fault injection is a functional technique.

3.2.2 Output Monitoring

In fault injection, as in normal testing, software output is monitored. The monitored output should include not only the normal results of a component, but also the forms of output not usually categorised as output. For example, a system call made by the observed component should be considered as output. There are two fundamental approaches for output monitoring in fault injection [74, p. 48].

- Monitoring for deviation between the output of the normal component version and versions with injected anomalies.

- Monitoring of the component output events using a *postcondition checker* [74, p. 272]. The postcondition checker analyses the output events and declares a failure when unacceptable output is observed.

A component may hide failures, if the monitoring is too narrow. The component may also have some unidentified input or output points. The monitoring of software output may be augmented with the monitoring of the internal state of the component or the state of the whole system [74, p. 51].

Software fault injection can be seen as a non-traditional software testing paradigm [74, p. 34]. Traditional testing usually tries to ascertain that the tested program conforms to stated requirements. Software fault injection is generally incapable of determining this conformance. If the source code is mutated, it is impossible to say whether the result is a product of the software itself or caused by the injected fault. If the software input is mutated, the requirements may specify the expected output for the mutated input.

3.2.3 Fault Injection Examples

Extended Propagation Analysis (EPA) is a source code software fault injection technique developed into a tool by *Reliable Software Technologies*[3]. Extended propagation analysis is composed of four steps: [74, p. 191]

1. Insert *instruments* into the source code of the analysed program. Instruments are software constructs for injecting artificial faults into program state and monitoring program behaviour. The locations of instruments can be selected either manually of automatically.

2. Compile the instrumented program.

3. Run the compiled program once for each test case and simulated fault. The simulated faults are injected by the instruments. The execution results including the instrumentation data is logged.

4. The collected logs are analysed and various metrics are calculated.

The metrics may, for example, indicate that some portions of the program were particularly sensitive to the injected faults and unacceptable behaviour were observed. These portions should probably receive additional effort to make them more tolerant.

Another method, *Adaptive Vulnerability Analysis* (AVA) assesses information system security quantitatively. AVA is used by commercial *Fault Injection*

[3] Reliable Software Technologies has been since renamed to *Cigital* [17].

Security Tool (FIST) [26]. The method is applicable for simulation of many known security threats, although it may fail to take account new threats [74, p. 226]. AVA is further developed from EPA [74, p. 234]:

- The ability to simulate code weaknesses though fault injection is strengthened. This includes the simulation of known threats, but also random mutations to simulate unknown threats.

- Automatic test-case generation is added. The rarely used region of input space is used to reveal inputs problematic from the security point of view.

- The postcondition checker is tuned to determine which events represent security breaches.

AVA has the following steps, browsed here using *Hypertext Transport Protocol* (HTTP) server password authentication as an example. The example uses only a single input scenario and single intrusion detection predicate [74, p. 238]:

1. Define what is analysed. In this case a web-page requiring authentication is fetched. The provided username is correct, but password is invalid. The analysis probes if a potential flaw can be exploited to access the web-page without proper authentication, i.e. without knowing the password.

2. Add instruments to the program. This step is done automatically.

3. Execute test cases as in EPA.

4. Analyse the results.

The locations where mutations caused security breaches, i.e. the web-page was fetched, are noted from the test results. These locations can be then manually inspected to see if they are in fact problematic and should be fixed.

3.3 Testing Security Properties

With a narrow view, testing is considered solely as the process of ensuring that a software product satisfies the stated requirements [6]. The requirements indicate what the *product is supposed to do* and this is reflected by the testing activities. However, many security properties are negative, they describe what the *program is not supposed to do* [25]. With a broader view the true mission of testing is to bring light to the risk of having serious problems in the product [6]. The stated requirements are just one source for identifying these risks, there are others. Not all risks are equal, the higher risk areas should receive more attention.

For example, in an approach presented by G. Flink and K. Levitt special *security specification* is constructed to be used in testing [25]. The security specification states the set of requirements which a program must not violate. A requirement may state, e.g. that the password file must not be accessed by the tested program. Other requirements may cover common programming mistakes which cause vulnerabilities. During testing a test case receives the fail verdict if any of the security requirements are violated.

A secure software component must be robust to maliciously formatted input. Testing activities should also include robustness testing. Robustness testing has a very attractive property: *there is no need for an input/output oracle.* The correctness of output from the IUT does not matter, if only the robustness is tested. This greatly reduces the required test design effort since only input must be crafted. A large number of test cases are feasible and desirable.

The security assessment of black-box software components must rely on component interfaces, although sometimes the monitoring facilities of the used operating system or special instrumentation may be available to access the internals of the analysed program. The design of test cases must be done only using the behavioural specification and knowledge of relevant vulnerabilities. On the positive side, the same functional tests can be used in assessing multiple implementations based on the same behavioural specification.

3.4 Summary of the Presented Techniques

The Table 2 summarises the presented software testing techniques. The techniques are categorised by whether they require access to the source code and whether the primary focus is in the conformance to functional requirements or in software quality. The techniques probing quality aspects without requiring source code access are the most interesting from the functional security analysis point of view.

Table 2. Summary of presented software testing techniques.

Technique	Requires source code?	Probes mainly conformance or quality?
Transaction flowgraph testing	No	Requirements
Domain testing and syntax testing	No	Both
Data and control flowgraph testing	Yes	Both
Property-based testing	Yes	Both
Code and data fault injection	Yes	Quality
Fault injection using interfaces	No	Quality

4. Functional Software Modelling

Functional testing of a software implementation requires modelling of the peer components which the IUT communicates with. For example, the testing of a server requires the model of a client. A real-world software component can be used, if it is available and allows the execution of the test cases as designed. On many cases this is not possible because there is a need to test exceptional scenarios which cannot be produced by the component. The component may also require a human operator to perform some test steps, which prevents fully automated testing.

A *model* of the peer component is required, if no software components are available or suitable. Generally, a model is an abstraction of the modelled system. A model concentrates on some properties of the system while suppressing other aspects [59, p. 15; 51, p. 1-3]. Models are built in order to understand complex systems, to simulate and visualise systems and to be used as communication aids between people. Many different models can be constructed from a single system.

The conformance testing methodology and framework recognises two major alternatives for realisation of the peers for functional testing: the *encoder/decoder* technique and *enhanced implementation* (CTMF calls them *lower testers*) [8, p. 79]. In the encoder/decoder technique a fixed data set contains the test cases and execution is straightforward. In the latter alternative the tests are executed by the enchanted implementation capable of producing the required behaviour.

The following discussion provides a pragmatic introduction to some techniques and methods for describing the behaviour and syntax of protocols. The applicability of the techniques and methods for functional security assessment is emphasised. Features for horizontal decomposition (modularization) and vertical decomposition (layering) are ignored.

4.1 Basic Techniques

This chapter presents *finite state machines* as the fundamental technique for functional modelling and an alternative approach, *regular expressions,* having the same modelling power. The techniques are extended into *extended finite state machines, context-free grammars,* and *attribute grammars*.

4.1.1 Regular Expressions

A regular expression consists of rules defining a *language* over some *alphabet* [2, p. 94]. The language is the union of all input strings accepted by the regular expression. Path expressions and directed graphs, which were presented earlier, can be converted into regular expressions by assigning *link weights* to edges [9, p. 243].

Regular expressions are traditionally used for lexical analysis in compilers, but they are widely applicable e.g. for behavioural modelling. Figure 3 shows a modified version of the earlier transaction flowgraph of the TFTP specification with edge names as link weights. The names are listed in Table 3. The corresponding regular expression is written down as expression (2).

$$\mathbf{\underline{R}\{D\underline{a}\}d\underline{a} \mid \underline{W}a\{\underline{D}a\}\underline{d}a} \tag{2}$$

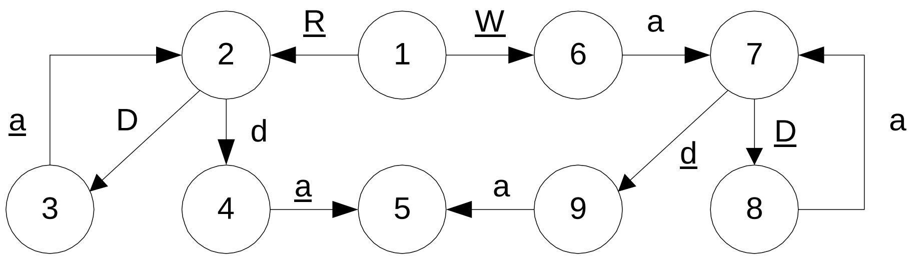


Figure 3. Regular expression for TFTP transaction without error handling.

There are some notational differences between the path expression and regular expression. In regular expressions alternative symbols are separated by vertical bar "**|**". Zero or more instances of a symbol are indicated by enclosing the symbol in curly braces "**{...}**". The language defined by the regular expression contains all possible TFTP transfers (without error handling). As before, underlining is used to separate received messages from sent messages.

Table 3. Explanations of link names of Figure 3.

Link name	Explanation
R	Request to read a file from the server.
W	Request to write a file to the server.
D	A 512 octet data block.
d	A data block shorter than 512 octets.
a	Acknowledgement of a data block.

When the modelled behaviour is complex, the regular expression becomes long and complicated. A regular expression can be divided into more manageable parts by using *regular definitions* [2, p. 96]. For example, expression (3) is equivalent to expression (2), but uses regular definitions of the *identifiers* **<TFTP>**, **<RRQ>**, and **<WRQ>**. The identifier **<TFTP>** denotes either a read transfer or a write transfer, **<RRQ>** denotes a read transfer and **<WRQ>** a write transfer. A regular definition has a *left-hand side* identifier and a *right-hand side* regular expression separated by an equals sign "**=**" in the middle. Recursion is not allowed, a left-hand side identifier must not be used in the right-hand side of the same definition or any definition transitively used in the right-hand side.

$$\begin{aligned}
\langle\mathbf{TFTP}\rangle &= \langle\mathbf{RRQ}\rangle \mid \langle\mathbf{WRQ}\rangle \\
\langle\mathbf{RRQ}\rangle &= \underline{\mathbf{R}}\{\mathbf{D}\underline{\mathbf{a}}\}\mathbf{d}\underline{\mathbf{a}} \\
\langle\mathbf{WRQ}\rangle &= \underline{\mathbf{W}}\mathbf{a}\{\underline{\mathbf{D}}\mathbf{a}\}\underline{\mathbf{d}}\mathbf{a}
\end{aligned} \qquad (3)$$

Fusion, an object-oriented development method, uses regular expressions called *life-cycle expressions* [18, p. 31]. A life-cycle expression is used to describe the possible interactions of the modelled system. Life-cycle expressions have notations for input events and output events, for sequential events, alternative events, repeating events, optional events, and concurrent events. A life-cycle expression can take advantage of regular definitions. Apart from notation, the only difference between life-cycle expressions and the presented approach is that life-cycle expressions can represent concurrent events.

Regular expressions are used in *scripting languages* and other similar tools for extracting portions from a textual (also binary) stream for searching or manipulation. For example, consider the definition of a TFTP write request message in Figure 4. The message has a two-octet *operation code*, character string *filename* terminated by a zero octet, and character string *data mode* terminated by a zero octet as well [].

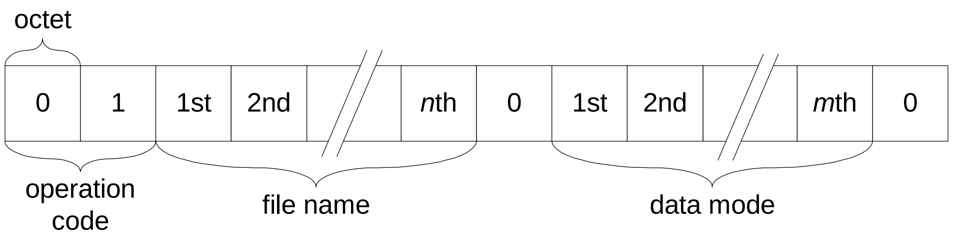


Figure 4. TFTP write request message.

Using notation from the scripting language Perl the write request would be identified and matched using regular expressions [55; 77]:

```
^\x00\x01([^\x00]*)\x00([^\x00]*)\x00$
```

The notation used in Perl is somewhat different from expressions (2) and (3). A backslash followed by letter x “\x*XX*“ indicates an octet with hexadecimal value *XX*, “`[^\x00]`“ indicates any character with null, postfix asterisk “*“ indicates that the proceeding symbols is repeated zero or more times. The initial character “^” and final character “$” indicate the beginning and end of data (actually end of line), respectively. Braces are used for grouping.

4.1.2 Finite State Machine

A finite state machine (FSM) (or finite automata) is capable of recognising a regular language [2, p. 113; 31, p. 16]. A FSM is constructed from *states, state transitions,* and *input symbols*. A FSM has a single *initial state* and set of *accepting states*. FSM execution starts from the initial state and goes through state transitions caused by the consumed input symbols. A string is accepted only if the string symbols lead from the initial state to one of the accepting states.

A FSM recognising the same language as a regular expression can be constructed from the regular expression. Figure 3 can be seen as a FSM recognising legal TFTP behaviour by considering the nodes as states and the edges as state transitions. The initial state is **1** and there is a single accepting state **5**.

An extended *finite-state machine* is a FSM with *variables* [23, p. 228; 60, p. 92]. The use of variables effectively limits the number of explicit states required for the modelling of such elements as numerical identifiers or counters. The used variables must be considered as part of the global state of an extended FSM.

4.1.3 Context-Free Grammars

Context-free grammar, also called *Backus-Naur Form* (BNF), is an extension to regular expressions [2, pp. 25, 172; 31, p. 8]. BNF is widely used, but there is no standard or consensus on the notation. Most of the notation used in this study is adopted from Wirth[4], but some modifications are made to conform to commonly seen conventions and to facilitate extensibility [82]. The resulting notation is described in Table 4, the term “closure” is taken from Gough [31, p. 78].

[4] Wirth’s article was written in 1977 especially to unify used BNF notations. This intention is clearly broken by introducing modifications.

Table 4. Context-free grammar symbols..

Symbol	Description	Notation
string	A string of characters enclosed in double quotes.	`"xyz" "Hello, World!"`
identifier	An identifier enclosed in angle braces. The identifier can be replaced by the right-hand side of the corresponding production.	`<a> <Name>`
production	Production consisting of the left hand side identifier, an equals sign, and right hand side symbol(s).	`<Name> = "xyz"`
sequence	Sequence (or concatenation) of symbols. Normal braces are used to group symbols.	`(<a> <b> <c>)` `<d> <e> (<f> <g>)`
null	Empty sequence.	`()`
selection	Selection (or alteration) from alternative symbols separated by a vertical bar.	`<a> | <b> | <c>`
closure	Selection between zero, one or multiple repeats of a symbol enclosed in curly braces, e.g. { <a> } equals to <a> |<a> <a> |<a> <a> <a> |... etc.	`{ <a> }`
option	Optional symbol, `[ <a> ]` equals to ( ) | <a>.	`[ 0xff ]`

The TFTP specification from Figure 3 is shown below using context-free grammar. Identifiers `<RRQ>`, `<WRQ>`, `<BLOCK>`, `<ACK>`, and `<LAST-BLOCK>` correspond to read request, write request, intermediate data block, acknowledgement, and last data block, respectively. The direction of messages is not shown.

```
<transfer> = <read-transfer> |<write-transfer>
<read-transfer> = <RRQ> <reads>
```

```
<write-transfer> = <WRQ> <ACK> <writes>
<reads> = {<BLOCK> <ACK>} <LAST-BLOCK> <ACK>
<writes> = {<BLOCK> <ACK>} <LAST-BLOCK> <ACK>
```

Context-free grammars can contain recursive structures forbidden in regular definitions [2, p. 165]. For example, consider a trivial language consisting merely of balanced braces:

```
<balanced> ::= "(" <balanced> ")" | "()"
```

There are as many close braces as there are open braces. This language cannot be described by regular expressions. Consequently a FSM cannot recognise all context free languages. A *push-down automata* is an extension to FSM which is able to recognise a context-free grammar [31, p. 149]. A push-down automata contains a stack for storing symbols.

BNF is used to describe the message formats of various Internet-related protocols [21]. Beizer suggests a method for creating test cases from the BNF specification of input language accepted by the IUT [9, p. 284]. Maurer has used BNF for the creation of test cases for hardware testing [49]. Maurer takes advantage of variables to distribute the same value into many parts of a single sentence. This resembles the variables in extended FSMs. He also used *action routines* to extend the capabilities of BNF. An action routine is a programming language routine performing calculations and other operations difficult to model using only a context-free grammar.

4.1.4 Attribute Grammars

Attribute grammars are an extension of context-free grammars having *attributes* associated to the grammar symbols [2, p. 279; 54]. In *first-order* attribute grammars attributes are separate from grammar, but in *higher-order* attribute grammars the attributes may themselves be grammar symbols. Attribute values are calculated by a traversal through the grammar using *semantic rules*. A semantic rule may also perform any other actions during the traversal besides calculating attribute values. A semantic rule is similar to action routines proposed by Maurer. The first-order attribute grammar attributes resemble the variables in extended FSM.

Real-time asynchronous grammar (RTAG) uses attribute grammars for a specification of protocols and for an automatic implementation of protocol entities [4]. A RTAG specification contains symbols for input messages, output messages, and special purposes (e.g. a timer). Externally supplied C-routines are used for communication with the outside world. RTAG has been proposed as a method for easier implementation of protocol stacks.

4.2 Modelling Standards

This chapter overviews some standard methods for software modelling. They take advantage of the techniques presented in the previous chapter. The applicability of the standard techniques for functional vulnerability assessment is discussed in the end of this chapter.

4.2.1 Specification and Description Language

Specification and Description Language (SDL) is a standard language for specifying systems [34; 23, p. 1]. The aspects covered by SDL are behaviour, structure and data. SDL can be used to describe the co-operation of a system with the environment, as well as, the internal system structure. SDL supports stepwise refinement from higher-level models into more detailed models. An SDL description may contain enough information to be directly executable or to be converted into programming language code. SDL was originally designed for telecommunication system modelling, but it is applicable to any kind of computer system. SDL includes both graphical and textual notation.

The overall behaviour of an SDL system is the joint behaviour of all *process* instances [23, p. 8]. Process behaviour is modelled using extended FSM [23, p. 227]. The state machines are further extended with *decisions* and *signals*. *Timers* are used to specify time sensitive actions, such as timeouts. *Procedures* are used in a similar manner as in programming languages [23, p. 125].

The SDL state machine of a partial TFTP server is shown in Figure 5. More understandable state and event labels are introduced compared to the earlier single-letter labels. The conditions related to the decisions between nondeterministic states are added to the diagram. Explanations of the labels used are given in Table 5.

For data definitions SDL uses the concept of *abstract data types* [23, p. 13]. Optionally an SDL specification can take advantage of ASN.1 data types. Abstract data types define data in a hardware and software platform independent manner. For example, an integer in SDL is defined as a mathematical concept and not limited to any particular implementation or range.

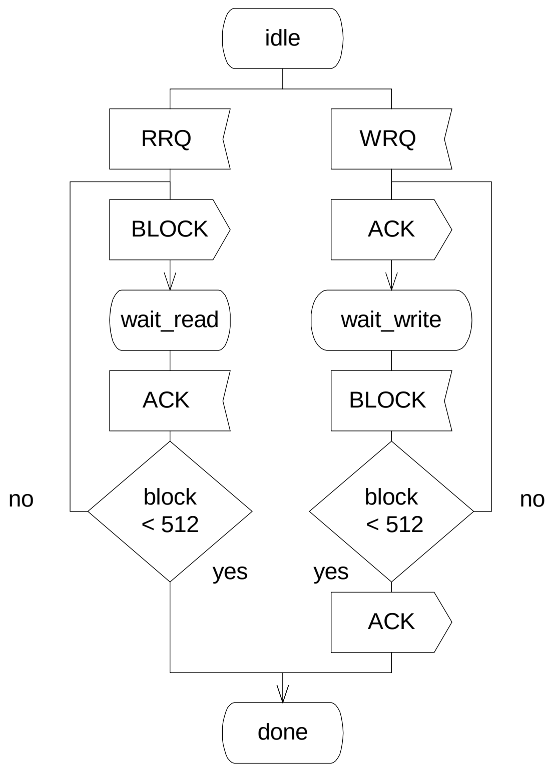


Figure 5. SDL state machine for TFTP transfer without error handling.

Table 5. Explanations of the events and states of Figure 5.

Label	Type	Explanation
idle	state	Waiting for a request.
RRQ	event	Request to read a file from the server.
BLOCK	event	A data block.
wait_read	state	Waiting for the client to acknowledge a block.
ACK	event	Acknowledgement of a data block.
WRQ	event	Request to write a file to the server.
wait_write	state	Waiting for the client to write the next data block.

4.2.2 Message Sequence Chart

A companion for SDL is a standardised *Message Sequence Chart* (MSC) language for visualising message exchanges [35; 23, p. 250; 58]. As SDL, MSC also includes both graphical and textual notations. An MSC shows one specific transaction compared to a state machine which captures the complete behaviour of an entity, and is usually easier to understand than a state machine. However, an MSC cannot be outright used as a specification since it only shows one possible interaction from many or an unlimited number of different interactions.

Figure 6 shows an error-free TFTP transfer of a 1050 octet file from server to client using MSC. The file name is “sample.txt” and transfer mode is “octet”. The file is transferred in three messages, two 512 octet messages and a final 26 octet message.

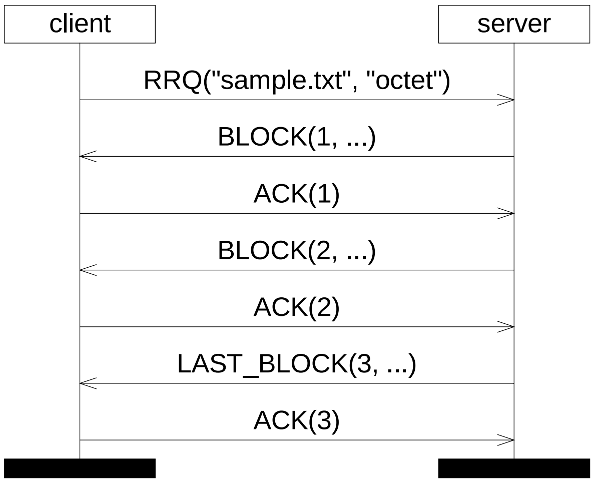


Figure 6. MSC of successful TFTP file read transfer.

4.2.3 Unified Modelling Language

Unified Modelling Language (UML) is a visual object-oriented modelling language to develop and exchange models [51, p. 1-5]. It is programming language independent. UML supports high-level concepts such as components, collaborations, frameworks, and patterns. UML is meant for specifying, constructing, visualising, and documenting software systems. It is not a programming language. In a software project UML modelling and programming are separate tasks.

UML provides *use cases*, *collaborations*, *state machines*, and *activity graphs* for behavioural specification [51, p. 2-83]. State machines are essentially finite state machines [51, p. 2-129]. The FSMs are described using graphs called *statechart diagrams* [51, p. 3-131]. *Activity graphs* are FSMs intended for the modelling of computational processes in terms of control-flows and object-flows [51, p. 2-160]. Apart from the differences in terminology and notation UML and SDL provide similar tools for behavioural modelling. Figure 7 shows a UML statechart diagram of a TFTP transfer. The black dot is the start state and black dot surrounded by white binding is the end state. The general form of a UML state transition is:

event [*guard condition*] / *action*

An *event* is the triggering condition for the state transition. A *guard condition*, when present, must also be satisfied. An *action* is taken when a state transition occurs. Figure 7 uses link names from Table 3 as events (received messages) and actions (send messages).

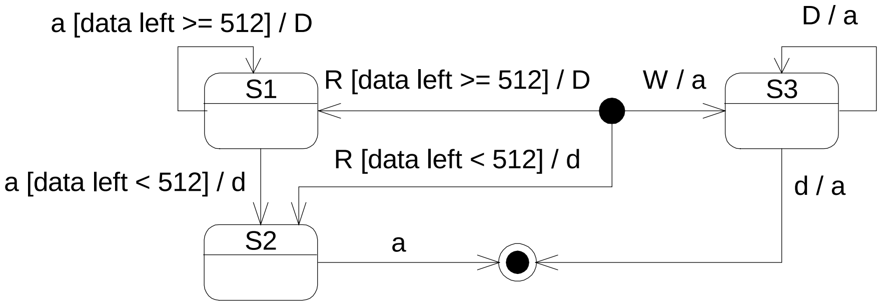


Figure 7. UML statechart diagram of TFTP transfer without error handling.

UML uses *interaction diagrams* to show the interaction between two or more instances [51, p. 3-97]. There are two variants, namely *sequence diagrams* and *collaboration diagrams*. A sequence diagrams shows interaction as messages arranged as a time sequence. A collaboration diagram emphasises more the relationships between communicating peers. UML sequence diagrams have a lot in common with MSC diagrams, the differences are mostly notational.

4.2.4 Tree and Tabular Combined Notation

Tree and Tabular Combined Notation (TTCN) is a standard notation for describing test cases [39; 8, p. 95]. TTCN has both a machine readable notation and a graphical notation suitable for human comprehension.

A TTCN test case is formed from a *behaviour tree* [8, p. 164]. A behaviour tree is a conceptual tree and is usually composed from several *TTCN trees*. There can be multiple concurrently executed behavioural trees. Behaviour tree nodes are called *behavioural lines*. Interaction with the IUT takes place at *points of control* *and observation* (PCOs) [8, p. 55]. There are several types of behavioural lines, the most important ones are described in the following list [8, pp. 164, 194].

- *Send event*, a data object is sent to the IUT. A send is assumed to be always successful.
- *Receive event*, a data object is received from the IUT. The range of data objects accepted is specified by *constraints*. The next behavioural line is evaluated if no matching objects are available.
- *Timeout*, which can be used to avoid endless waits for data objects.
- *Pseudo events*, which do not cause interaction with the IUT:
  - *Qualifier* evaluates a condition and acts differently depending on the outcome of the condition.
  - *Assignment* performs some calculations and assigns results to a variable.
  - *Timer operation* is used to set and reset a timer for timeout lines.

In addition to the listed behavioural lines there are default lines and constructs for a more compact description of behaviour trees. TTCN behaviour lines can be looped and branches can be defined recursively, to achieve similar description power as recursion in context-free grammars.

TTCN test suites are not directly executable, they are *abstract test suites* [8, p. 77]. They must be converted to *executable test suites* by selecting applicable test cases and providing the needed parameters, such as used network addresses and ports.

4.2.5 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) is a platform and language independent notation for specifying data structures [36; 23, p. 219; 47, p. 16]. Many protocol specifications use ASN.1 to specify the format of exchanged messages or *protocol data units* (PDUs). ASN.1 structures can be mapped into program language data-structures using automated tools [47, p. 16].

ASN.1 separates an *abstract syntax* and a *transfer syntax* [47, p. 30]. The abstract syntax describes the information content of data structures. The transfer syntax defines the octets and bits actually transmitted. The process of converting data from the abstract syntax into the transfer syntax is called *encoding*, the reverse is *decoding*. There are multiple encoding and decoding rules with different properties for ASN.1, e.g. *Basic Encoding Rules* (BER) [37; 47, p. 236]. Tools and protocol stack layers are readily available for ASN.1 encoding and decoding [47, p. 16].

4.2.6 Extensible Markup Language

The *Extensible Markup Language* (XML) has recently gained attention as a standard method for describing structure and relationships of data elements in documents [12]. A *document type declaration* (DTD) specifies how the XML documents of that particular type must be constructed. Using a DTD the correctness of a XML document can be checked. A *document object model* (DOM) defines the programming interface for accessing XML documents and tools conforming to the DOM are available.

A DTD can be considered as the grammar of the language made up from the XML documents conforming to the type. XML documents are machine readable text-only documents. XML document syntax is described in a platform-independent manner which makes XML suitable as a data exchange method. There is ongoing activity to use XML for the message syntax description of communication protocols [12].

4.3 Modelling Summary

The presented approaches provide tools for system modelling. Depending on the approach, different properties of the system are emphasised while others are abstracted away. The three main aspects of behavioural modelling can be identified:

1. *Interaction modelling* to describe joint behaviour of two or more entities.

2. *Entity modelling* to describe behaviour of an entity.

3. *Syntax modelling* to describe syntax of data exchanged by entities.

Interaction models describe messages exchanged between two or more entities, e.g. TTCN or MSC. They capture a single sequence of messages rather than all possible interaction sequences. The interplay of the entities and the causal relationship between the messages is emphasised. The syntax of the exchanged messages and behaviour not relevant for the interaction is omitted. An interaction model is not enough for the specification of the participating entities, since it describes only partial functionality. A test case is an executable interaction model.

An entity model captures the (complete) behaviour of an entity or a system, e.g. SDL. The model emphasis the order and types of messages sent and received by the entity. Message syntaxes and the behaviour of other entities are suppressed. The behaviour of a real-world entity tends to be complex, which makes behaviour models complex and difficult to understand. A formally defined entity model can be executed to provide a simulation of the modelled entity.

A syntax model describes the syntax of data exchanged by entities, e.g. ASN.1 or XML. It does not take into account the order of messages or the semantics (meaning) of the messages. An abstract syntax describes the information contained in data structures platform-independently. A transfer syntax describes the actual bits exchanged.

A software or hardware implementation is the ultimate entity and syntax model. There are tools to actually create implementations from entity and syntax models. The presented modelling methods are summarised in Table 6.

Table 6. Summary of presented modelling methods.

Modelled aspect	Method
Interaction	MSC
	TTCN
	UML interaction diagrams
Entity	SDL
	UML statechart diagrams
Syntax	ASN.1
	XML

4.4 Conclusions

Functional testing of a component requires interaction models of the peers of the tested component. In general, interaction models are derived from entity models. For example, TTCN test cases are derived from an SDL specification.

The derivation process from entity models into interaction models reduces uncertainty to the point that actions are described in sufficient detail: message contents are specified, loops are assigned with repetition counts and optional elements are included or excluded, etc. The output messages must be completely defined, but the input messages may contain uncertainty, which is resolved when the receive operation takes place. The derivation process can be automated by a tool or conducted manually. With proper tool support the resulting interaction model can be executed, but only after all required parameters, like network addresses and ports, are specified.

The derivation of interaction models from the entity model is not addressed by the presented methods. The gap has to be filled by additional methods. Also the means of communication between the interaction models and real-world programs, e.g. using network sockets, is left to be specified somewhere else. The creation and execution of interaction models seems to be a complex process, which requires the developer to master the following skills.

- Entity modelling (e.g. SDL or UML).

- Interaction modelling (e.g. UML, MSC or TTCN).

- The derivation of the entity models from interaction models.

- The use of the execution platform.

There are methods for behavioural modelling not discussed in this study, e.g. *Extended State Transition Language* (Estelle) and *Language for Temporal Ordering Specification* (LOTOS) [60, pp. 24, 92, 173]. However, they do not change the nature of behavioural modelling. It can be justifiably argued that the rigid approaches may be well suitable for conformance testing, but are not optimal for the creation of a large number of versatile test cases with exceptional or even illegally formatted data, e.g. for functional robustness testing.

5. Mini-Simulation Method

This chapter proposes a new method, the *mini-simulation method*, for the functional modelling of protocol exchanges. It is based on the techniques presented in earlier chapters. The method covers interaction modelling and syntax modelling with capabilities to create exceptional data. The idea is to take advantage of multiple miniature simulations, not a single complete simulation bound to be complex.

5.1 Mini-Simulation Overview

The mini-simulation method was originally developed for functional robustness testing. The main requirement was the ability to generate a large amount of messages with one or few exceptional elements but otherwise legal content. Checksums and other such fields must be calculated correctly, and messages had to be delivered into tested implementations.

However, from the early stages the broader potential of mini-simulations was recognised. A set of requirements started to form, deriving from the experience gained during robustness testing.

5.1.1 Requirements

A mini-simulation method provides the means for creating models from protocol specifications. The method aims to fulfil the following set of requirements.

> R1: A mini-simulation specification should be simple and human readable.

A single notation is sufficient for all behaviour modelling. A developer[5] should be able to create a simulation of the basic behaviour just by looking at the mini-

[5] The term *developer* is used herein for the user of the mini-simulation method: software designer, programmer, tester, etc.

simulation specification, even if she or he is not an expert on this particular protocol.

> R2: A specification should provide a basis for multiple models for different purposes.

Multiple models are created for a single protocol and a single protocol entity. Each model emphasises a different point of view or a different scenario. All models use the same specification as the starting point, for easier maintenance and comprehension. The maintenance is easier because corrections and additions to the protocol specification propagate to all models. Comprehension of different models should be easier if they all use the common definitions from this single specification. The derivation of the models from the specification must be covered by the method and be fully visible.

> R3: The modelling of exceptional conditions and optional elements should be possible without cluttering the specification with all turns and twists.

After the basic behaviour is covered, the developer can create models performing exceptional and optional behaviour. The basic models should remain for providing a learning path to newcomers.

> R4: Models should be applicable as prototype implementations.

The first use of the mini-simulation method in a software project can be the implementation of stubs. The developed software component is debugged against the stubs.

> R5: Models should be applicable as executable test cases.

As the project continues, test cases can be constructed using the mini-simulation method.

> R6: Execution should produce a result, which is itself executable.

A direct way to create test cases is to execute a mini-simulation model and collect the execution output. The output itself must be a mini-simulation model.

> R7: There should be a direct way to mix programming language routines into models.

A model should be able to use routines developed using a general-purpose programming language for syntactic calculations, semantic decisions, external communications, etc.

> R8: The encoding of valid and invalid behaviour should be easy.

A mini-simulation model of a message, file, or any other data structure should be flexible for modifications to produce both valid and invalid data.

> R9: The decoding and analysis of messages should be easy.

A model must be executable "both-ways", i.e. applicable both to the encoding of data structures into octets and to the decoding of octets back into structures.

5.1.2 Negative Requirements

The mini-simulation method should not attempt to provide a means for complete entity modelling or the structural decomposition of a complex system. These tasks are already addressed with "heavy-weight" methodologies such as SDL and UML. The mini-simulation method is a debugging and testing aid, not a design method.

Description of the complete entity behaviour is not required. A mini-simulation is not to be used as method for code generation, protocol standardisation or behavioural validation. The method should promote many small models, not a large monolith one. The mini-simulation method should not be bloated by conditions, expressions, etc. "programming" constructs better handled by embedded general-purpose programming language routines.

The performance of a mini-simulation model is not essential, simplicity, ease of use, flexibility, etc. are far more desirable properties. A mini-simulation is not intended for applications having real-time demands.

5.1.3 Concept

A mini-simulation model is made up of the *master specification* and *configuration*. Higher-order attribute grammars are used for both interaction modelling and syntax descriptions. BNF is the basis of the mini-simulation language notation. The master specification is usually not a complete protocol specification, but describes typical and/or error-free protocol exchanges. A partial specification is faster to write and easier to understand than a comprehensive specification. Still, the specification should contain the definitions of optional messages to support the modelling of all message exchanges.

Semantic rules for mini-simulation attribute grammar are implemented as Java objects. Communication between the model and external world is done using *communication rules*, also implemented using Java. Java is a platform-independent, high-level, versatile, and general-purpose programming language suitable for rule implementation [71].

A model is derived in the configuration. The configuration reads the master specification (grammar) and uses *operations* to modify the grammar into the required interaction model. Figure 8 visualises the mini-simulation concept. In the figure a single protocol specification is the basis for multiple models for multiple purposes: functional testing, stub implementation, traffic analysis, simulation, and conversion.

5.2 Specifications

Attribute grammar and BNF are the basis of the mini-simulation specification approach. A single notation is used for behaviour specification and syntax specification.

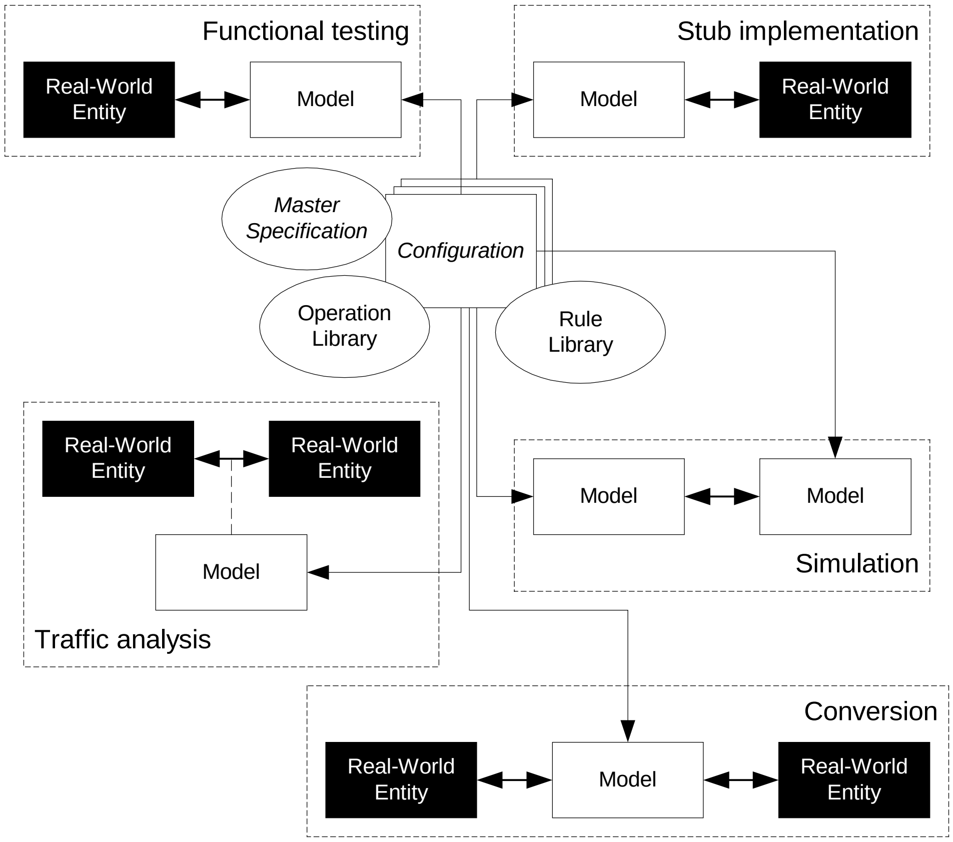


Figure 8. Visualisation of the mini-simulation concept.

5.2.1 Grammar Symbols

A mini-simulation specification is a higher-order attribute grammar. The language defined by the grammar is the set of all protocol exchanges conforming to the specification. No distinction between context-free productions and attributes is made, a production can be viewed also as an attribute whose name is the left-hand side of the production and value is the right-hand side. Productions are divided into *data productions* and *type productions*. The distinction is important mainly during *evaluation*, introduced later.

Table 7. Additions to symbols given in Table 4.

Symbol	Description	Notation
octet	8-bit octet, a one or two digit hexadecimal value prefixed with “0x”.	`0x0 0x12 0xff`
octet range	Octet range, a shorthand notation for selection between octets from a range. Octets are separated by a dash “-“.	`0x00 - 0x7f` `0xf0 - 0xff`
bit	A bit. Two special identifiers `<B0>` and `<B1>` serve as bits 0 and 1, respectively.	`<B0> <B1>`
type production	Type productions are different from data productions. They use a double-colon before the equal sign.	`<Name> ::= "xyz"`
repeat	Repeating a symbol *n* times. The repeated symbol is prefixed with *n* “`x`”.	`100 x "A"`
tag	Tagging a symbol, a tag is a string with prefix exclamation mark “`!`”. A tag bears no syntactic meaning.	`!up <PDU>`
closure with limits	A closure can be prefixed by an upper repeat limit, e.g. 3 { <a> }, or by a lower and upper limit, e.g. 1..3 { <a> }. The number of repeats is only allowed in the limit range. Letter n is used for unlimited upper limit, e.g. 0..n { <a> } equals to { <a> } closure.	`3..8 { 0x0 }` `1..n { "xyz" }`
comment	Comment in BNF, ignored during processing. Comments start with the “#” character and continue to the end of the line.	`# This is a comment`

The basic context-free grammar symbols and their notations were presented earlier in Table 4. The mini-simulation additions are given here in Table 7, the comment symbol is included to the table although it is not really a symbol. The grammar symbols and their short descriptions are presented in Appendix A. The precedence is the following:

1. Repeats and tags.

2. Sequences.

3. Selections.

The above precedence can be changed by the use of regular braces. From this point forward the term *BNF* refers to the mini-simulation notation and term *grammar* or *simulation grammar* stands for a mini-simulation protocol specification.

Strings, bytes, and bits can be mixed in the syntax modelling of protocol messages. The encoding of a string into bytes uses the Java default string encoding [71]. Mini-simulation models are byte aligned and bits must be arranged in multiples of eight bits. A run-time exception is thrown, if illegal bit arrangements are encountered.

The previously introduced TFTP server example is used also in mini-simulation examples. Figure 9 shows the grammar, using BNF, defining TFTP behaviour without error handling or timeouts.

```
<transfer> = <read-transfer> |<write-transfer>
<read-transfer> = !up<RRQ> <reads>
<write-transfer> = !up<WRQ> <writes>
<reads> = {!down<BLOCK> !up<ACK>} !down<LAST-BLOCK> !up<ACK>
<writes> = !down<ACK> {!up<BLOCK> !down<ACK>} !up<LAST-BLOCK> !down<ACK>
```

Figure 9. Mini-simulation grammar of TFTP behaviour without error handling.

The specification contains tags. A tag applies to the symbol immediately following it, e.g. in `!up<WRQ>` the tag `!up` applies to `<WRQ>`. Tags convey only semantic meaning, they have no syntactic content. Tags are used for various purposes, in the TFTP grammar they indicate the direction of a message flow:

`!up` for upstream (from a client to a server) and `!down` for downstream (from a server to a client).

A simulation grammar can be alternatively shown as a *simulation tree*. The simulation tree concept is similar to trees in TTCN [8, p. 164]. A symbol is represented in a *simulation tree* by a node. Symbols and nodes are divided into *non-terminals*, intermediate nodes, and *terminals*, leaf nodes. Sequences, selections, options, closures, repeats, and tags are represented by intermediate nodes. Strings, octets, bits, and nulls are represented by leaf nodes. An octet range can be drawn to be either a leaf node or an intermediate selection node. An identifier can be a leaf node or an intermediate node with the right-hand side of the production drawn below the identifier. Tree node syntax is given in Appendix A. Figure 10 shows the TFTP behaviour grammar in tree format.

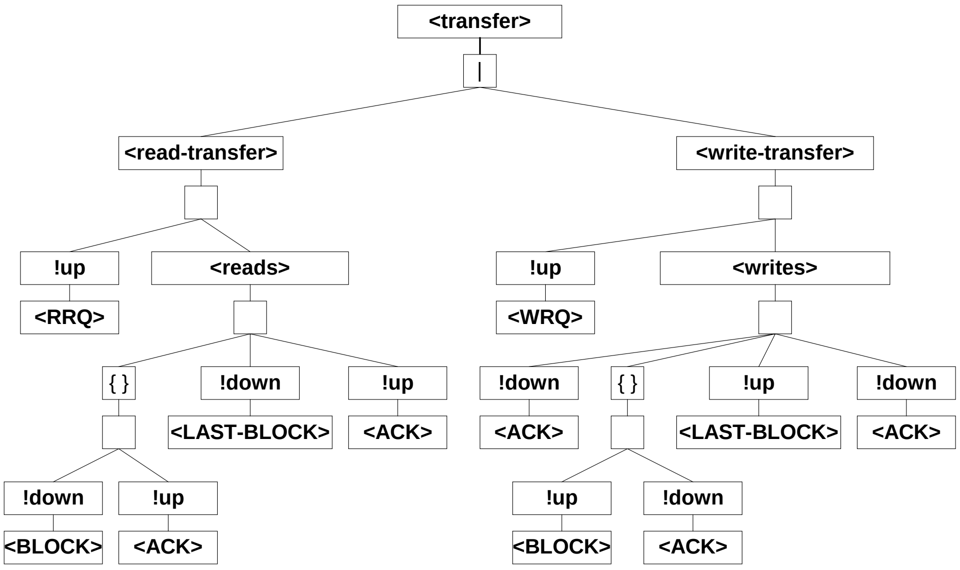


Figure 10. Simulation tree of TFTP behaviour without error handling.

Message syntax definition uses the same set of symbols as the behavioural definition. Figure 11 specification defines the message formats of the TFTP protocol using BNF.

```
# Request PDUs
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

# Subsequent PDUs
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512 x <OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 { <OCTET> }
<ACK>  ::= (0x00 0x04) <BLOCK-NUMBER>
<ERROR> ::= (0x00 0x05) <ERROR-CODE> <ERROR-MESSAGE>

# Miscellaneous productions
<MODE> ::= "octet" 0x00 |"netascii" 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<BLOCK-NUMBER> ::= <OCTET> <OCTET>
<ERROR-CODE> ::= <OCTET> <OCTET>
<ERROR-MESSAGE> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
<OCTET> ::= 0x00 - 0xff
```

Figure 11. Mini-simulation grammar of TFTP messages.

The size constrains of the last block and the intermediate blocks are modelled without a condition expression. The intermediate data block (`<BLOCK>`) is specified to contain exactly 512 octets of data and the last data block (`<LAST-BLOCK>`) is modelled to have from zero to 511 octets of data.

The grammar also includes an error message `<ERROR>`, although only error-free behaviour is specified in the behavioural part. It is a good idea to specify the error messages as well, since the grammar can be always mutated to test error handling.

5.2.2 Rules

Rules are implemented by the Java programming language. Rules are divided into *communication rules* and *semantic rules*. Communication rules provide the means for exchanging data with external entities. Semantic rules are used to implement constructs which are difficult or impossible to describe using only pure grammar, such as lengths and checksums. A rule registers *callbacks* which are associated to *triggers*. Callbacks are routines called to achieve the functionality provided by the rule. A trigger can be a tag, an identifier, or a special trigger `root`. The `root` trigger implies that the callback must be called before the grammar is evaluated. Apart from the `root` trigger multiple rules cannot share triggers.

Communication rules divide a grammar into a *behaviour part* and a *syntax part*. The behaviour part describes the behaviour of the modelled entity in a way similar to SDL or TTCN. The syntax part describes the syntax of used messages as in ASN.1 transfer syntax[6]. A communication rule *input trigger* indicates an input data stream and an *output trigger* an output data stream.

The TFTP grammar (see Figure 10) uses the tags `!up` and `!down` to indicate UDP messages. The grammar is applicable to both the modelling of a server or a client since the grammar models the traffic between them rather than the behaviour of one or the another. Whether a client or a server is simulated depends on the communication rules assigned to the tags. A server has `!up` as an input trigger `!down` as an output trigger, a client has `!down` as the input trigger !up as the output trigger.

5.2.3 Evaluation

Evaluation transforms an *input grammar* (or *input tree*) into an *output grammar* (or *output tree*). The simulation functionality, e.g. sending and receiving of messages, takes place as a side-effect of an evaluation. An output grammar could be used again as input grammar to playback the earlier evaluation.

The mini-simulation *evaluation engine* traverses an input tree in top-down, left-to-right order. When a trigger, a tag or an identifier, is encountered, the

[6] Here the transfer syntax is specified directly, but generally semantic rules can be used to achieve a separate encoding phase and use of an abstract syntax.

evaluation is transferred to the registered rule by calling the callback. The enclosed branch of the input tree is handed over to the rule as a *rule input* branch. If the trigger is a tag, then the rule input is the child symbol of the tag. If the trigger is an identifier, then the rule input is the right-hand side of the corresponding production. During evaluation the desired functions are performed by the rule. After evaluation the rule returns a *rule output* branch. The evaluation engine attaches the rule output to the output tree on the location of the trigger and continues the evaluation.

A semantic rule takes the rule input branch, does semantic calculations, and returns the output branch. On the other hand, a communication rule either *sends* the input[7] branch or *receives* data according to the input branch. During a send callback, the input branch is evaluated to calculate inner semantic rules, if any, and then *encoded* into raw data, which is sent into the communication channel. During a receive callback, the raw data is *decoded*[8] from the communication channel using the input branch. The rule output from a send callback is the evaluated branch, the output branch from a receive callback is the decoded branch.

In addition to the input branch and output branch, rules also have read and write access to all productions available at the point of the evaluation. This breaks the strict top-down, left-to-right order of the evaluation. A rule may choose productions "further down" to be evaluated immediately. Also, a rule may overwrite an already evaluated production. This is a powerful mechanism, but causes tricky problems if used without caution.

[7] The terminology is somewhat confusing: from an evaluation point of view the *rule input branch* is really sent, although generally the branch sent by a communication rule is called an output branch and the received branch is called an input branch.

[8] Term "decoding" is here synonymous to the widely used term "parsing", e.g. in [2] and [31]. The term "decoding" is used to provide symmetry to the term "encoding" although the actions are not fully symmetrical since encoding is preceded by evaluation while decoding is not.

All this is best explained by examples. Figure 12 shows a simulation tree evaluation with a single communication rule. The sample protocol consist of a request message “Ok?” and response messages “Ok” or “NotOk”.

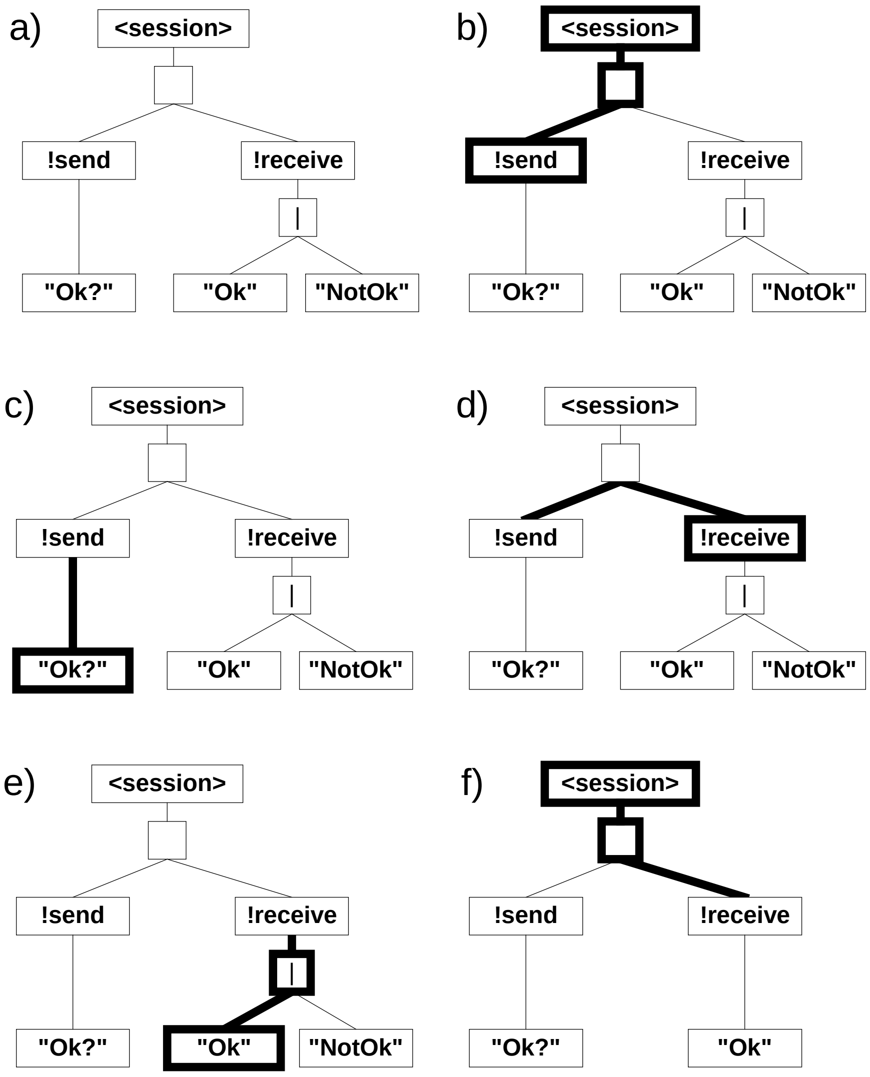


Figure 12. Evaluation example with a communication rule.

```
<session> = !send "Ok?" !receive ("Ok" |"NotOk")
```

The communication rule output trigger is the tag `!send` and the input trigger is the tag `!receive`. Figure 12 a) shows the simulation tree before evaluation. In b) the evaluation starts from the root identifier `<session>` and advances down until the output trigger `!send` is encountered. The evaluation is handled by the registered communication rule callback. In c) callback evaluates the input branch, encodes it, and sends the resulting octets into the communication channel. In d) the communication rule callback returns and the evaluation continues right until another trigger, `!receive`, is encountered. There the communication rule callback waits for data from the channel and when data is received in e) it is decoded. The received reply is the string “Ok” and the output tree is modified to reflect the response. In f) the evaluation proceeds back to the root and is terminated, the final output tree remains as a result of the evaluation.

The evaluation (or decoding) of a selection symbol from an input grammar results in the choosing of exactly one child to the output grammar. Traversal of the selection starts from the leftmost child symbol. The traversal may be unsuccessful, in which case the next child to the right is tried. The first successful child causes the skipping of the remaining children on the left. For example, in Figure 12 e) if the response would have been “NotOk”, then the decoding of the left selection child “Ok”, would have been unsuccessful, but the decoding of the right child “NotOk” would have succeeded.

A branch is unsuccessful if it is *backtracked*. Backtracking may be caused by the evaluation engine, e.g. an identifier without a corresponding production is encountered. A communication rule may backtrack for a variety of reasons e.g., if the received data does not match the syntax specification. A semantic rule may also backtrack. Generally, the backtracking of a symbol causes the backtracking of all the parent symbols until a selection is encountered. As explained before, backtracking causes the next child of the selection to be traversed. The selection itself becomes backtracked, if all children backtrack.

A child symbol of a closure is evaluated (or decoded) greedily as many times as the child symbol succeeds or the upper repeat limit imposed on the closure is achieved. In the output grammar the closure is replaced by a sequence of zero or more of the resulting child symbols. For example, in the TFTP grammar the `<reads>` production is defined as:

```
<reads> = {!down<BLOCK> !up<ACK>} !down<LAST-BLOCK> !up<ACK>
```

In a client simulation the above closure models the receiving of full 512-octet data blocks and the sending of acknowledgements. The closure is evaluated as many times as `!down<BLOCK>` is successful (sending of an acknowledgement `!up<ACK>` virtually always succeeds), i.e. as many times as a 512-octet block is received. The last received data block will be shorter than 512 octets, which causes the backtracking of the closure, subsequent reception[9] of the last data block, and sending of the last acknowledgement.

During evaluation, a data production (defined using "=" in BNF) is treated as a "write-once attribute". When the left-hand side of the production is encountered from the input tree, the traversal proceeds to the input branch defined on the right-hand side of the production. After the branch is evaluated the identifier is added to the output grammar with the resulting output branch as the right-hand side. The new data production may be different or the same as in the input grammar, depending on whether selections were encountered on the right-hand side. After the first evaluation of the right-hand side of a data production, the branch is no longer re-evaluated even if the identifier is encountered again in the simulation tree. Any subsequent evaluations are skipped and the production value in the output grammar is left untouched. This feature can be used e.g. to decode a session identifier from a received message and use it in subsequent send messages, like in the following grammar.

[9] A received UDP datagram is temporarily stored by the UDP communication rule, if the rule backtracks. The same datagram is then "received" again next time the rule input callback is called.

```
<session> ::= !receive ("id " <id> 0x00) !send ("received id " <id> 0x00)
<id> = { 0x01 - 0x7f }
```

In contrast to a data production, a type production (using "`::=`" in BNF) is not modified, but is always copied unmodified from the input grammar to the output grammar. If the evaluation of the right-hand side of a type production encounters selections, the identifier itself is replaced by the output branch in the output grammar. For example, execution of the TFTP read transfer specified in the MSC of Figure 6 results in the output grammar below[10].

```
<transfer> = <read-transfer>
<read-transfer> =
    !up((0x00 0x01) ("sample.txt" 0x00) ("octet" 0x00)) <reads>
<reads> =
    ((!down((0x00 0x03) (0x00 0x01) ...) !up((0x00 0x04) (0x00 0x01)))
     (!down((0x00 0x03) (0x00 0x02) ...) !up((0x00 0x04) (0x00 0x02))))
    !down((0x00 0x03) (0x00 0x03) ...) !up((0x00 0x04) (0x00 0x03))
```

In the grammar the type productions, e.g. `<RRQ>` or `<BLOCK>`, left-hand side identifiers are replaced by the modified right-hand sides. In contrast to this, the data productions `<transfer>` and `<reads>` are present, but modified.

5.2.4 Evaluation Problems

The top-down, left-to-right traversal method used by the evaluation engine is intuitive and easy to follow. In principle the evaluation can backtrack from infinitely deep branches when evaluation is not successful. On the negative side, backtracking is an inefficient method [2, p. 181]. More effective parsing methods are difficult to be used because of the rules, which allow unlimited features to be added in the evaluation process. The behaviour of a rule cannot be predicted beforehand.

[10] Ellipsis "**...**" indicates omitting of symbols for fitting a sample conveniently onto a printed page.

Another problem related to rules is that a rule cannot always be backtracked. For example, a communication rule which has already sent a message into the network cannot take the message back if backtracking is required.

The evaluation process can get stuck in an infinite loop, if care is not taken in specification construction. *Left recursive* productions lead to infinite loops and should not be present in mini-simulation grammars [2, pp. 176, 181]. In left recursion the left-hand identifier is also the first symbol of the right-hand side of the production, e.g. the production `<a> ::= <a> <b>` is left recursive. Also a closure whose child symbol evaluation always succeeds will be expanded infinitely until the evaluation engine runs out of memory.

5.3 Paths and Masks

Symbols and branches from a grammar need to be accessed for different purposes, e.g. by operations to mutate the grammar. A symbol from a grammar is uniquely identified by a *path*. Since the use of full paths is cumbersome, *masks* are used as shorthand notations for paths.

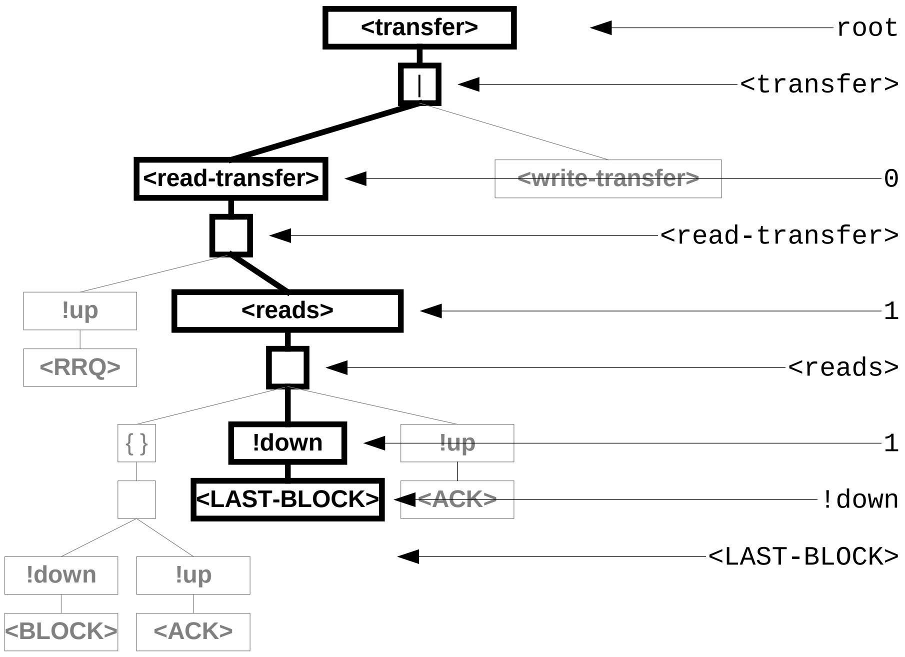
Paths and masks are made up of *segments* separated by dots "**.**". In a path each segment corresponds to an edge in the simulation tree, a mask also contains *wildcard* segments. A path specifies a single sequence of symbols starting from the root of the tree, advancing to a child of the root, to child of that child, and so on. A special path `root` points to the root of the tree, all other paths end to an intermediate symbol or to a leaf symbol. A path segment is either an index of a child symbol, an identifier, or a tag. Child indices are numbered from 0 onwards starting from the leftmost child. Figure 13 shows an example path from the TFTP grammar, the path specifies the last data block sent from a read transfer.

Full paths tend to be quite long, even for simple grammars. Masks provide the wildcard segments for selecting multiple segments at once. A wildcard may *match* several path segments, similarly a mask may match several paths: There are two wildcards.

- Question mark "?" matches any single path segment.

- Asterisk “*” matches zero or more segments until the next segment on the right side of the asterisk matches.

For example, mask `*.<LAST-BLOCK>` matches the paths to the content of all last blocks. The mask also matches the last block from the write transfer since the asterisk matches any suitable path prefix. This can be prevented by rewriting the mask into `*.<read-transfer>.*.<LAST-BLOCK>`, which explicitly matches only the read transfer. The question mark is useful for matching several symbols which are *siblings*, i.e. children of the same parent symbol. For example, all “fields” (operation code, mode, and filename) of the read request can be matched by the mask `*.<RRQ>.?` at once.



<transfer>.0.<read-transfer>.1.<reads>.1.!down.<LAST-BLOCK>

Figure 13. Sample path from the TFTP simulation tree.

A *definition path* can be used to access the right-hand sides of productions directly. A definition path does not start from the root symbol, but from any identifier. For example, the last data block can be specified simply as `<LAST-BLOCK>`. The mask includes again the last block from both the read transfer and write transfer. Mask `<read-transfer>.*.<LAST-BLOCK>` specifies the last block only from the read transfer.

A closure is equal to a selection of two or more sequences of the closure child symbol. This is significant when closure children are accessed using paths and masks. This is best understood by considering production `<C> ::= 0..3 { <A> }`, having a limited closure as the right-hand side. The production is equal to another production `<C> ::= () |<A> |<A> <A> |<A> <A> <A>` having a selection instead of the closure. A path can be used to access the sequences inside the closure: mask `<C>.0` selects null `()`, mask `<C>.1` selects (<A>) and mask `<C>.2` selects (<A> <A>), etc.

5.4 The Mini-Simulation Toolkit

A prototype *mini-simulation toolkit* has been implemented using the *Java* programming language [71]. The toolkit is a collection of Java components rather than an application or applications. Each mini-simulation is a Java application constructed from a set of parameterised components from the toolkit. A *configuration script* specifies the required components and sets their parameters. Configuration scripts are written using the *Tool Command Language* (*Tcl)* [1; 79].

5.4.1 Two-Language Solution

The mini-simulation model uses two programming languages: a *system language* and a *scripting language*. The reasoning is to use different languages for different tasks [52]. Complex routines are done with the system language, while integration and user customisation utilises the scripting language. System programming languages (e.g. C, C++, Java) are suitable for building complex algorithms and data structures, but a programmer must write a lot of code for each task because the system programming languages tend to give little functionality per code line. Scripting languages (e.g. Perl, Tcl, shell scripts) give more functionality per code line, but they are less scalable for larger tasks than system programming languages. A script is also interpreted in run-time, which brings a negative performance impact.

The system programming language used is Java [71]. Java was chosen as the system language because it is easily portable to many different systems and provides programming aids like garbage collection, reflection, and a rich set of application programming interfaces for different purposes. The supported scripting language is Tcl with *TclBlend*, but any scripting language with Java support can be used. Tcl and TclBlend were chosen because Tcl and Java complement each other conveniently [69]. *Jacl* is a version of Tcl, with in-build TclBlend functionality, written using Java [1]. Jacl can be used instead of the regular Tcl interpreter.

5.4.2 Configuration Scripts

A simple mini-simulation configuration script in given in Figure 14. The script can be run by a Java-enabled Tcl interpreter, i.e. Tcl with TclBlend or Jacl. The script has three parts repeating in all configuration scripts: *script header*, *script body*, and *script trailer*. Tcl uses "#" as a comment indicator similarly to the mini-simulation BNF.

5.4.3 Script Header

The script header imports the Tcl packages *java* and *configurer*, using Tcl procedure `package require`. The java-package is provided with TclBlend and Jacl. The configurer-package is specific to the mini-simulation toolkit, the package contains Tcl procedures and variables for configuration scripts. Before the configurer-package can be imported, the location of the toolkit must be appended to the package search paths stored in the variable `auto_path`. The location is found by the static method `tclLib` of Java class `FI.protos.Root`, if the toolkit is properly installed. All Java classes of the toolkit have the `FI.protos`[11] prefix. The method is called from Tcl using the procedure

[11] "`FI`" stands for Finland and "`protos`" for the PROTOS-project.

`java::call`. The final procedure of the script header, `namespace import`, enables use of the procedures from the configurer-package without the “`configurer::`” package prefix.

All Tcl configuration script headers contain at least the presented initialisation procedures. Additionally they may define more procedures to be used in the script body. All Tcl features, such as command-line parameters, are available for a script writer in all parts of the configuration scripts.

```
# Script Header
package require java
lappend auto_path [java::call FI.protos.Root tclLib]
package require configurer
namespace import configurer::*

# Script Body
section simulator {
    section operation {
        parseBNFFile "tftp.bnf"
        show
    }
}

# Script Trailer
call [it] run
```

Figure 14. A simple mini-simulation configuration script.

5.4.4 Script Body

The script body defines the software architecture of the mini-simulation. It specifies the used components, their connections, and sets the required parameters of the components. The script body is made up of *configuration sections*, *configuration procedures*, and *configuration properties*. A section configures a software component for the mini-simulation. Two sections are nested, when the component configured by the inner section is part of the component configured by the outer section. A configuration property belongs to the enclosing section. A configuration procedure is a convenience mechanism, which may set multiple properties at once or configure a whole component at once. A procedure takes zero or more *configuration parameters*.

The example configuration script in Figure 14 defines a root configuration section **simulator**, inner section **operation**, and two configuration procedures: **readFile** and **show**. The former procedure has a single parameter, a file name. The example configuration does not set any configuration properties. A block diagram of the configuration is given in Figure 15.

The configuration procedures inside the **operation** section are the operations for manipulation of the mini-simulation grammar. The example script has two operations. The first operation, configured by procedure **parseBNFFile**, reads the BNF specification from the file “`tftp.bnf`”. The second operation, configured by procedure **show**, outputs the specification into the log of the simulator. All available operators are listed in Appendix B.

Script body is the variant part of a configuration script, while header and trailer tend to be quite static. In the following chapters only the script body or parts of the script body are shown, rather than the whole script.

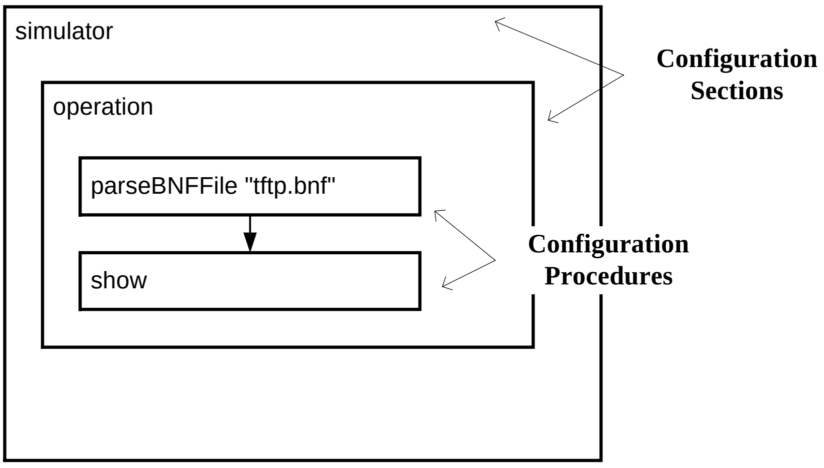


Figure 15. Block diagram of the sample mini-simulation.

5.4.5 Script Trailer

The script trailer specifies the actions taken after the application is build. A minimal script trailer typically invokes the start-up method of the constructed application, as done in the example script in Figure 14. The application may also be serialised into a file using the Java *serialization* feature [71]. A serialised configuration can be started without the Tcl interpreter as a regular Java application.

The trailer of the example script uses the Tcl procedure `call` (actually `configurer::call`) to invoke the method `run` in the Java object specified by `[it]`. The Tcl construct `[it]` invokes the Tcl procedure `it` (`configurer::it`). The procedure returns either the component configured by the enclosing section or the root object of the Java application if no enclosing section is present. In the example script the situation is the latter and the method `run` of the configured Java application is invoked.

The invoking of the `run` method causes the execution of the *mini-simulation application*. Building the mini-simulation application and executing it are two completely separate tasks. Conceptually this is equivalent to the compilation of a C-program and execution of the resulting executable.

5.4.6 Running a Configuration Script

A configuration script is run from an UNIX shell or from a Windows command prompt using TclBlend in the following fashion, assuming “`sample.config.tcl`” is the name of the configuration script file.

```
% jtclsh sample.config.tcl
```

Alternatively the script can be run using Jacl, the Java-based Tcl interpreter:

```
% java tcl.lang.Shell sample.config.tcl
```

The TclBlend or Jacl must be property configured and the root directory of the mini-simulation toolkit must be in the *classpath* of the Java installation. Readers should refer to the documentation of TclBlend and Java for instructions on how to do this [1; 71].

5.5 Modelling Tasks

This subchapter shows how some typical modelling tasks are done using the mini-simulation toolkit. Corresponding configuration script clips are presented.

5.5.1 Communication Rules

The adopted TFTP specification is jointly specified by the BNF clips in Figure 9 and Figure 11, and also given in Appendix D. The following configuration script body specifies a simulation of the TFTP server. The TFTP specification is assumed to be in BNF-format in the file “`tftp.bnf`”.

```
section simulator {
    section operation {
        parseBNFFile "tftp.bnf"
        # Specification modifications here...
        section insertRule {
            section rule [new FI.protos.rule.UDPSocket] {
                property open root
                property input !up
                property output !down
                property localPort 69
            }
        }
        evaluateRules
        show
    }
}
```

The block diagram of the mini-simulation application is shown in Figure 16. Two additions to the earlier example are introduced. Firstly, a communication rule for the TFTP upstream and downstream traffic is added. Secondly, the procedure **evaluateRules** is used for evaluation of the simulation grammar.

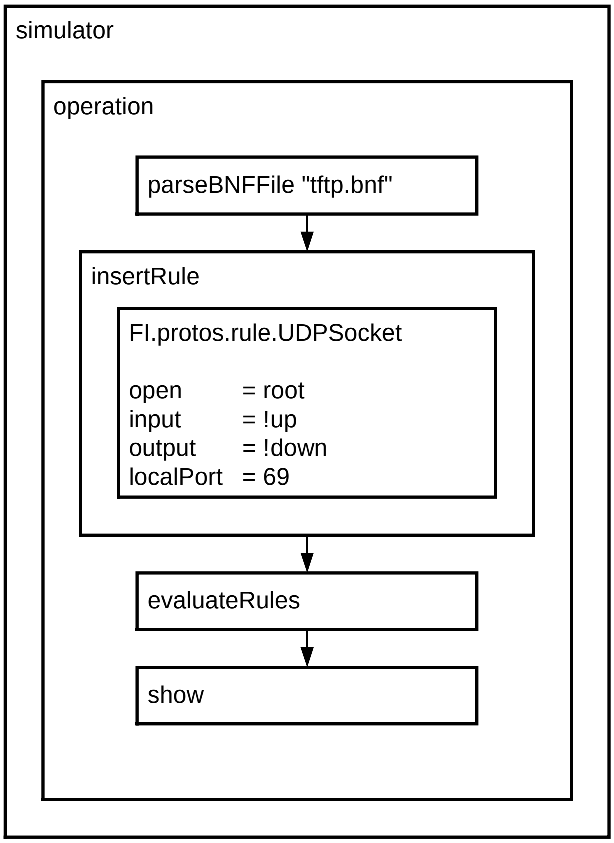


Figure 16. Block diagram of TFTP server mini-simulation.

The simulation acts as a server, so upstream traffic is input for the simulation and downstream traffic is output. The required communication rule is inserted in the **insertRule** section. The section configures the rule class in an inner section **rule**. The UDP socket communication rule is implemented using Java as the `FI.protos.rule.UDPSocket` class. An instance of the class is created using the Tcl procedure `new`. Configuration properties **open**, **input**, **output**, and **localPort** of the instance are parameterised. The **input** property determines the input trigger, and property **output** the output trigger for the communication. The property **open** indicates the *context* (branch) in which the rule operates. A context is usually the whole simulation tree, as in the example. A communication rule requires the specification of the context for correct handling of backtracked input triggers. Some semantic rules use contexts as well. Finally, the **localPort** property specifies the UDP port which the simulation should be listening for input from the client.

A TFTP client simulator would be almost similar: The rule labels of input and output would be swapped. No **localPort** property would be set, but **remoteHost** and **remotePort** properties would be set to specify the host and the port of the server.

However, the simulation produced by the configuration script will not work as a TFTP server or as a TFTP client. The problem is that no content for the TFTP transfer messages is specified. The content has to be added to the simulation grammar originally read from the master specification in the file “`tftp.bnf`”.

5.5.2 Simple Message Exchanges

The first working example is a mini-simulation of the TFTP server. The server accepts a read request and provides a 1050-octet file, if no network errors takes place. The read transfer is described in the MSC in Figure 6. The simulation reads the BNF specification (Appendix D), adds the content into the grammar, inserts a communication rule and evaluates the resulting grammar. For brevity, only the required additions to the configuration body of the previous mini-simulation are shown.

Specification is modified by the operation **replace**. It takes two input parameters, a mask and replacement branch. The replacement branch is specified using BNF. The following four replacements modify the grammar for the server-side simulation of the read transfer by adding the transferred data blocks. *Note that the curly braces around replacement branches are required by the Tcl interpreter, they are not part of the BNF notation.*

```
replace <reads>.0 {<_OLD>.2}
replace <reads>.0.0.*.<BLOCK> {<BLOCK>.0 (0x00 0x01) 512x 0x61}
replace <reads>.0.1.*.<BLOCK> {<BLOCK>.0 (0x00 0x02) 512x 0x62}
replace <reads>.*.<LAST-BLOCK> {<LAST-BLOCK>.0 (0x00 0x03) 26x 0x63}
```

The replacements are illustrated in Figure 17. The first replacement uses the *substitution identifier* `<_OLD>`. The substitution identifier is used to substitute the original branch, or parts of it, back into the new branch. In the first replacement operation the substitution identifier stands for the `{!down<BLOCK> !up<ACK>}` closure. The value `<_OLD>.2` selects a sequence of two data block and acknowledgement pairs. The next two replacements set the content of the 512-octet data blocks. The operation codes for the data blocks are not specified directly, but they are snipped using path `<BLOCK>.0` from the right-hand side of the `<BLOCK>` production. The final fourth replacement builds up the last data block with 26-octets of data.

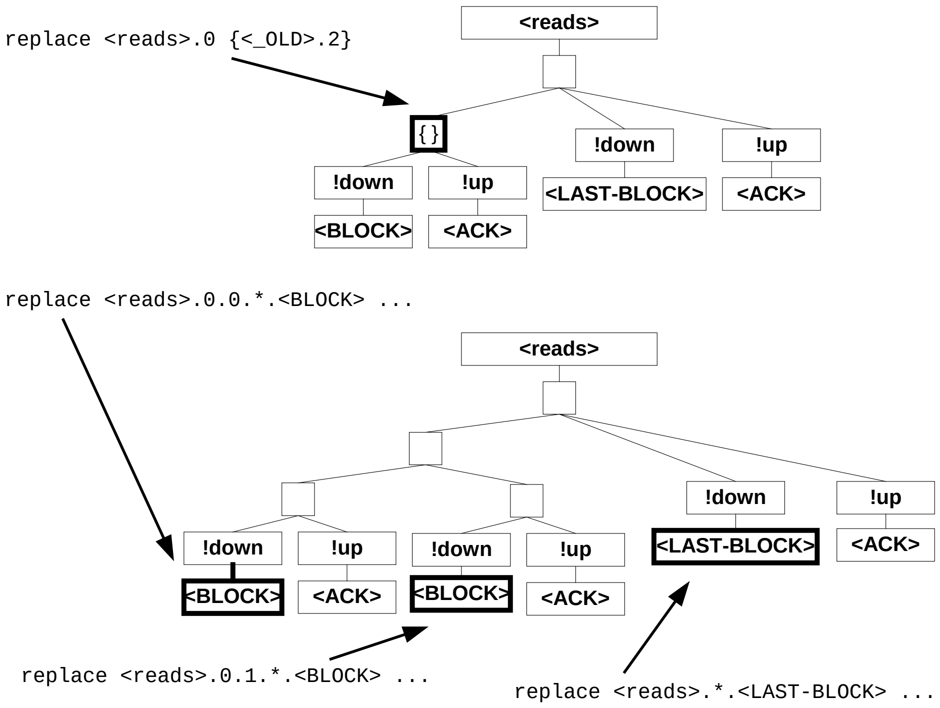

replace <reads>.0 {<_OLD>.2}
<reads>
{}
!down
!up
<BLOCK>
<ACK>
!down
!up
<LAST-BLOCK>
<ACK>
replace <reads>.0.0.*.<BLOCK> ...
<reads>
!down
!up
<BLOCK>
<ACK>
!down
!up
<BLOCK>
<ACK>
!down
!up
<LAST-BLOCK>
<ACK>
replace <reads>.0.1.*.<BLOCK> ...
replace <reads>.*.<LAST-BLOCK> ...

Figure 17. Illustration of some replacements to the TFTP grammar.

The read request message and acknowledgements are not filled with content, since they are received from the TFTP client. The production `<reads>` has the following value after the replacements.

```
<reads> = ((!down ((0x00 0x03) (0x00 0x01) 512x 0x61) !up <ACK>)
           (!down ((0x00 0x03) (0x00 0x02) 512x 0x62) !up <ACK>))
          !down ((0x00 0x03) (0x00 0x03) 26x 0x63) !up <ACK>
```

The right-hand sides of the type productions `<BLOCK>` and `<LAST-BLOCK>` are not modified, but the identifiers are replaced by the actual contents. This behavioural is analogous to the treatment of data productions and type productions by the evaluation engine. A definition path (path starting with the left-hand side identifier of a production) would have to be used in the replacements to mutate the data productions <`BLOCK`> and `<LAST-BLOCK>` themselves.

After the replacements the simulation can be started. The running simulation can be contacted by a TFTP client program to request for a read transfer. The transfer is similar to the MSC of Figure 6.

5.5.3 Exceptional Message Exchanges

The previous simulation provided an error-free read transfer. In a similar fashion one can prepare a simulation with error behaviour. The following operations build a simulation ending in an error message.

```
replace <reads>.0 {<_OLD>.2}
replace <reads>.0.0.*.<BLOCK> {<BLOCK>.0 (0x00 0x01) 512 x 0x61}
replace <reads>.0.1.*.<BLOCK> {<BLOCK>.0 (0x00 0x02) 512 x 0x62}
# Simulate error, should terminate transfer
replace <reads>.1 {!down (<ERROR>.0 (0x00 0x00) ("test error" 0x00))}
replace <reads>.2 ()
```

5.5.4 Semantic Rules

Thus far only communication rules have been used. Semantic rules are used to augment the grammar description. For example, in the previous TFTP protocol example a semantic rule could have been used to automatically number the sent data blocks.

The semantic rule **SequenceNumber** is used to create a sequence of numbers. The rule has the property **number** for the location of the sequence number and property **step** to increment the sequence number counter. Also, the format of the number and the counter value have to be set. The following script clip configures the **SequenceNumber** rule and adds the required tag to the TFTP grammar for the counter increment.

```
section insertRule {
    section rule [new FI.protos.rule.SequenceNumber] {
        property number <BLOCK-NUMBER>
        property step <ACK>
        property start 1
        property byteLength 2
    }
}
replace <reads>.0 {<_OLD>.2}
replace <BLOCK>.2 {512 x 0x61}
replace <LAST-BLOCK>.2 {26 x 0x63}
```

The **insertRule** section configures the rule to use a two-byte sequence number starting from 1 (property **start** is 1 and property **byteLength** is 2). The sequence number will replace `<BLOCK-NUMBER>` during evaluation and the counter is incremented when an acknowledgement is received. After these modifications, the server simulation can be constructed without explicitly replacing each `<BLOCK-NUMBER>` with an appropriate value. In the example, the transferred octets are directly put inside the `<BLOCK>` and `<LAST-BLOCK>` right-hand sides.

Checksums and length fields are frequently present in protocol messages, although they are not needed in the TFTP example. The following artificial example defines a PDU having an 8-bit length field, 16-bit checksum, and variable length data block. The checksum is calculated over the whole PDU and the length is calculated over the data block and checksum.

```
<pdu> = <length> <payload>
<length> = <OCTET>
<payload> = <checksum> <data>
<checksum> = <OCTET> <OCTET>
<data> = { <OCTET> }
<OCTET> ::= 0x00 - 0xff
```

Two semantic rules, **Length** and **IPChecksum**, the first for length calculation and the second for checksum calculation, are needed to encode and decode the defined PDU. Both of the inserted rules require a context trigger to be specified by the property **context**. The rules are evaluated after the evaluation of their contexts, the length is calculated after `<pdu>` is evaluated and the checksum after everything else. The length field is calculated before the checksum and hence the checksum calculation uses the correct length value. Length calculation does not need correct checksum, only the length of the checksum is relevant, which is always two octets.

```
section insertRule {
    section rule [new FI.protos.rule.Length] {
        property context <pdu>
        property length <length>
        property payload <payload>
        property byteLength 1
    }
}
section insertRule {
    section rule [new FI.protos.rule.IPChecksum] {
        property context root
        property sum <checksum>
        property payload <pdu>
    }
}
```

5.5.5 Mixing Communication Rules

Multiple communication rules can be mixed into a single simulation. This is illustrated in the final TFTP server simulation example. The simulation reads a file and sends it to the client. The data block content is read from the file using the communication rule **FileIO** and delivered to the client using the **UDPSocket** rule. The semantic rule **SequenceNumber** is used for numbering the data blocks. The configuration of the rules **UDPSocket** and **SequenceNumber** is not shown, since no modifications to their earlier configuration sections are required. The configuration section of the rule **FileIO** and a few required replacements are shown below.

```
section insertRule {
    section rule [new FI.protos.rule.FileIO] {
        property open !open-file
        property input !read-file
        property file "sample.txt"
    }
}
replace <reads> {!open-file<_OLD>}
replace <BLOCK>.2 {!read-file<_OLD>}
replace <LAST-BLOCK>.2 {!read-file<_OLD>}
```

The rule **FileIO** reads data from the file `"sample.txt"`. The file is opened when the `!open-file` tag is encountered. The file content is read into the last field of the data blocks, where the `!read-file` tags are located. When the file runs out of data, sending of the full 512 octet block `<BLOCK>` backtracks, since there is no longer 512 octets available. The evaluation engine then proceeds to the sending of the last data block `<LAST-BLOCK>`.

Default semantic rules and communication rules included in the mini-simulation toolkit are given in Appendix C. New rules can be implemented when no default rule provides the required functionality.

6. Mini-Simulation Testing

The mini-simulation method presented in the previous chapter was originally designed for functional robustness testing. This chapter provides a description of this *mini-simulation testing* method. The testing method includes both test design and test execution. The prototype mini-simulation toolkit contains the extensions required for the method.

6.1 Extensions for Testing

The mini-simulation toolkit must consider the format of test cases during test design and execution. The test cases must be injected into the IUT. The IUT must be instrumented in order to control and monitor its behaviour. The test verdicts have to be assigned.

6.1.1 Test Strategy

Test cases are designed using syntax testing principles from Beizer [9, p. 284]. The starting point of test case design is the protocol specification. A test design is done in two main phases:

1. Mutate the specification to contain both normal elements required for the test cases and also *anomalous* elements (or *anomalies*), which are unexpected or illegal protocol elements targeted for finding flaws from the IUT.

2. Design the test cases by specifying combinations of normal protocol elements and anomalies.

Each test case forms a separate simulation grammar. During the test case execution the grammar is evaluated. The injected PDUs are formatted according the protocol specification, apart from the added anomalies. The test verdicts are assigned based on the observed behaviour, the correctness of the IUT responses is not checked for.

Since all functionality is modelled in a uniform fashion using simulation grammar, robustness testing can cover all levels of the behaviour and message syntaxes. Anomalies can be syntax elements, messages or even higher-level interactions.

6.1.2 Injection

Test input must be injected into the IUT through an interface. The most used interface is probably a network socket interface, but also dial-in (serial) interfaces, command-line access, application programming interfaces, files, e-mail content etc. should be considered. In the mini-simulation testing method communication rules are used for the injection.

6.1.3 Instrumentation

Test verdicts are based on the acceptability of the behaviour of the IUT during test execution. The verdict assignment requires monitoring of the IUT. In the simplest case the IUT user interface or the entry in the process list can be monitored for visible failure modes like crashes and hangs. Unsupervised test execution and detection of more complex vulnerabilities require subtle instrumentation:

- *Test log analysis*: The log of the mini-simulation application is analysed and conclusions are drawn about the behaviour of the IUT.

- *Audit trail analysis*: The operating system may provide audit trails containing information about actions of the IUT [25]. Sometimes the operating system can be modified to support the test instrumentation.

- *Wrappers*: Messages transferred through the interface between the tested executable and the operating system can be intercepted [45]. Test verdict can be based on analysis of the intercepted calls.

- *Code modifications.* The source code of the IUT can be augmented with statements that log the actions relevant to verdict assignment. In *binary patching* an executable is modified directly.

Instrumentation is also required to control the IUT. It must be started, and possibly terminated and restarted when it hangs. Such extensive control of the IUT may be tedious and dependent on the used operating system. The design and implementation of test instrumentation is beyond this study.

6.2 The Testing Process

As in all development activities, mini-simulation testing must also be carefully designed to succeed. Although the space considerations prevent a full-fledged description of the required testing process, this subchapter shortly discusses some of the fundamental issues of the assessment process.

6.2.1 Preparations

Before any other activity is started, the testing requirements must be formulated. The problem statement must be specified and the expected value of the test results considered. When the overall problem statement is done, the tester should conduct a survey of the available information. There might be tracks of past vulnerabilities or other information available to help define the focus of the testing. One must also seek standards, RFCs, technical documentation, academic publications, etc. required for the test case design. A test plan must be written, it should define minimally:

1. The protocol or protocol family used in the test cases. The complexity of protocols may prevent full test coverage and only a subset of the selected protocol can be used.

2. The vulnerability types searched for, and the types of anomalies used in the test design.

3. Methods of the test case injection and the IUT instrumentation.

4. List of the software products or components to be tested.

Before committing to a test plan, it may be a good idea to conduct *trial tests* to get a better view of the problems ahead. Trials should be quickly drafted, something the mini-simulation method is ideal for, and only one or two products should be evaluated. Some of the problems which might be found in trial tests are:

- The selected protocol or protocol family is too complex for comprehensive tests or some required specification is not available. The problem statement should be narrowed.

- There are problems with injecting the test cases into the IUT interface. Better communication rules must be obtained or the problem statement must be adjusted.

- Monitoring of the tested component is difficult or impossible. New tools for better instrumentation are needed.

However, not too much emphasis must be given to the trial tests and no test verdicts should be drawn from this phase. After the problems are solved, the tested products or components are selected, and the tester feels confident that the tests can be conducted, the test plan must be finalised.

6.2.2 Test Design

The test cases are designed to expose security-critical flaws from the IUT. A test designer should also add a few test cases which ensure that the connection to the IUT is working properly and the IUT does attempt to process the injected data. Otherwise tests could me marked as passed without actually exercising the IUT at all or marked as failed due to injection problems.

Test design using the mini-simulation toolkit is done by obtaining a specification for the tested protocol and writing a configuration script or multiple scripts. Specification has to be converted into mini-simulation format, if no suitably formatted specification is available.

A test case should be designed to contain a single anomaly, or a combination of a few anomalies, and be otherwise legally formatted. An IUT may have too easy of a task to reject completely misshapen PDUs. Care is also needed since it is easy to create redundant test cases. Observing the messages from a live protocol session is a good starting point for test case design. It is essential to get some feedback on the effectiveness of the tests during test design. At least one implementation should be available to try out the test cases. However, the tests may become biased towards finding flaws from the IUTs used during test design and be less effective against other IUTs.

6.2.3 Test Execution

Tests should be executed in a well-defined and documented environment. Documentation must be precise enough to make it possible to reproduce the results afterwards. The test design and the test execution phases must be clearly separated to ensure the use of the same test cases for all IUTs. Tests must be run again for all implementations if the test design is changed.

A lack of instrumentation to observe and control the IUT during test execution may cause problems. Manual intervention may be required after a crash or other failure of the IUT. Manual work prolongs test execution and leaves possibilities for human mistakes. The tester should consider aborting test execution or skipping the most problematic test groups for an IUT which performs exceptionally poorly. All such short-cuts must be carefully documented.

The linking of some test case to a particular failure may be difficult or even impossible. Often the failures are caused by a combination of several test cases or the IUT may slowly degenerate during the test process and finally fail.

6.2.4 Post-Processing

Result analysis must be done to estimate the value of the findings. In the simplest case it may be enough to double-check that test cases with a fail verdict do indeed cause problems in the IUT. Sometimes is it necessary to create exploits to demonstrate the severity of the findings. It must be understood, that there may be flaws in the IUTs, which are were not found by the tests. The comparison of test results from multiple IUTs must take into account that the tests might be biased to emphasise the problems of certain IUTs.

The underlying flaws behind the vulnerabilities should usually corrected. The root causes behind the emergence of the vulnerabilities in the first place should also be resolved (e.g. bad programming conventions or ignorance) and preventive actions taken. This should be possible, if the tester is part of the organisation developing the tested product or the tester is in a position to put pressure on the product vendor. However, if there is no direct link between the tester and the originator, the process for handling vulnerability reporting and patching may become complex. An interested reader should refer to other sources for more information, e.g. papers from Laakso et. al. [44; 46].

6.3 Testing Using the Mini-Simulation Toolkit

Test case design using the mini-simulation toolkit is a matter of creating test configuration scripts. The configuration scripts for testing are similar to the basic configuration scripts and the used Tcl configurer-package is the same. This chapter presents the steps of test design and the architecture of a test mini-simulation with the help of an example test suite: the *TFTP test suite*.

6.3.1 Example: TFTP Test Suite

The TFTP test suite assesses the robustness of TFTP server implementations by feeding them with exceptional PDUs from a simulated TFTP client. Test cases shall cover the read request PDU, the acknowledgement PDU, and error PDUs. The data block PDUs are not mutated, since they are sent from server to client. The TFTP server must contain a file named “sample.txt” readable by the simulated client.

The TFTP test suite uses the TFTP specification already presented in Figure 9 and Figure 11. The specification is also given in Appendix D. The whole test suite configuration is in Appendix E. The suite is limited in scope since it is constructed only for demonstration purposes.

6.3.2 Sections

Figure 18 shows the most important configuration sections of a test configuration script. The root section is **driver**. Driver is made up of the subsections **preSelection**, **selection**, and **postSelection**. The script header and the script trailer are equal to ones shown in Figure 14.

The subsections **preSelection** and **postSelection** are similar to the **operation** subsection in the previously shown **simulator** sections. The operations in the **preSelection** section are executed before the test cases are selected in the **selection** section. The operations in the **postSelection** section are executed separately for each case after the selection process.

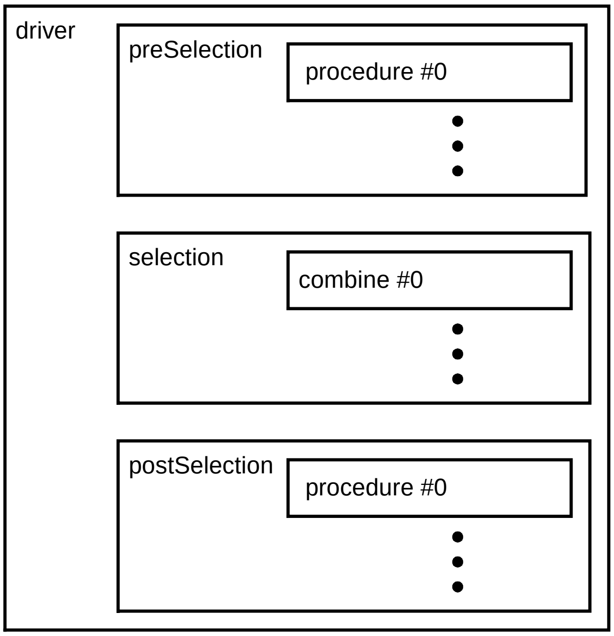


Figure 18. Main sections of a test configuration script.

6.3.3 Pre-Selection Section

The **preSelection** section contains the operations which are applied before test case selection takes place. The section usually fulfils the following tasks.

1. Read the specification file to construct the grammar.

2. Adjust the grammar for testing by adding and/or removing elements.

3. Define (legal) default content for PDUs sent to the IUT.

4. Define and insert anomalies to be used in the test cases.

5. Add the needed semantic rules and communication rules.

The **driver** section forms the body of the TFTP test suite configuration script. The first subsection to the **driver** section is the **preSelection** section. Firstly, the **preSelection** section reads the specification from a file with the **parseBNFFile** operation.

```
section driver {
    section preSelection {
        parseBNFFile "tftp.bnf"
```

The test suite concentrates on the read transfer, hence the grammar can be chopped. All test cases start with the read request PDU (but in some test cases the operation mode is mutated and the message is not a read request). The `<read-transfer>` production is augmented with some erroneous interactions, defined in the production `<read-errors>`. The test suite expects the TFTP server to supply one full data block followed by the last data block, which requires the sample file to have a size between 512 octets and 1023 octets.

```
# only read transfer
replace <transfer> <read-transfer>

# RRQ followed by successful or erroneous transfer
replace <read-transfer>.1 {<_OLD> |<read-error>}

# successful transfer, expecting file size 512...1023 byte
replace <reads>.0 <_OLD>.1

 # define transfers ending to error
```

```
data <read-error> {(!down <BLOCK> !up<ERROR>) |
         (!down <BLOCK> !up <ACK>) !down <LAST-BLOCK> !up<ERROR>}
```

The fields `<FILE-NAME>`, `<MODE>`, and `<ERROR-MESSAGE>` are all null-terminated strings. They are set with the default values "`sample.txt`", "`octet`", and "`test error`", which are accompanied with anomaly strings defined in `<A-string>`. The field `<ERROR-CODE>` is a 16-bit field , with a default value set to zero and the anomaly values defined in `<A-16>`. Operation codes are also 16-bit values with the same anomalies. The operation code is mutated only from the read request `<RRQ>` (send first) and from the error messages `<ERROR>` (sent as the second or last message). This way the operation code anomalies are tried on all of the three upstream PDUs, no anomalous operation code needs to be added to the acknowledgement PDU `<ACK>`. However, the block number of the acknowledgement PDU is mutated with anomalous values from `<A-16>`.

```
# value and anomaly to RRQ op. code, filename and mode
replace <RRQ>.0 {<_OLD> |<A-16>}
replace <FILE-NAME> {"sample.txt" 0x00 |<A-string>}
replace <MODE> {<MODE>.0 |<A-string>}

# value and anomaly to error op. code, error code and message
replace <ERROR>.0 {<_OLD> |<A-16>}
replace <ERROR-CODE> {(0x00 0x00) |<A-16>}
replace <ERROR-MESSAGE> {"test error" 0x00 |<A-string>}

# anomaly to acknowledgement block number
replace <ACK>.1 {<_OLD> |<A-16>}
```

The field <BLOCK-NUMBER> is used in the acknowledgement messages when no anomaly is present in the block number. The legal value is set by the **SequenceNumber** rule. The presented modifications and elimination of unnecessary productions result in the following TFTP grammar.

```
<transfer> = <read-transfer>
<read-transfer> = !up <RRQ> (<reads> |<read-error>)
<reads> = (!down <BLOCK> !up <ACK>) !down <LAST-BLOCK> !up <ACK>
```

```
<read-error> = !down <BLOCK> !up <ERROR> |
    (!down <BLOCK> !up <ACK>) !down <LAST-BLOCK> !up <ERROR>

<RRQ> ::= (0x00 0x01 |<A-16>) <FILE-NAME> <MODE>
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512x <OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 {<OCTET>}
<ACK> ::= (0x00 0x04) (<BLOCK-NUMBER> |<A-16>)
<ERROR> ::= (0x00 0x05 |<A-16>) <ERROR-CODE> <ERROR-MESSAGE>

<FILE-NAME> ::= "sample.txt" 0x00 |<A-string>
<MODE> ::= "octet" 0x00 |<A-string>
<OCTET> ::= 0x00 - 0xff
<ERROR-CODE> ::= 0x00 0x00 |<A-16>
<ERROR-MESSAGE> ::= "test error" 0x00 |<A-string>
```

The anomaly production <A–16> is defined to have different exceptional 16-bit integer values. The production <A-string> is defined to have various special characters, long strings with null termination, and long strings without null termination. See Appendix E for the exact values. The specified values are not intended as guideline for anomaly design, rather they provide examples of what can be tried out.

As a last step, two rules have to be added: **UDPSocket** for UDP communication and **SequenceNumber** for the creation of acknowledgement block numbers. The complete **preSelection** section is given in Appendix E. After these mutations the TFTP grammar is ready for test case selection.

6.3.4 Selection Section

In the **selection** section the elements from the grammar, prepared in the **preSelection** section, are combined to form each test case. The **selection** section contains **combine** subsections, each combine creates a test group. The **combine** subsections have the property **label**, which identifies the test group. Furthermore, the subsection **masks** contains zero or more **add** procedures, which select the elements to the test cases of the test group. The grammar is modified for each case to force the evaluation of the selected elements during the test case. In the TFTP test suite the first test group consist of only one test case, labelled *zero-case*.

```
section combine {
    property label "zero-case"
    section masks {
    }
}
```

In the **masks** section there is nothing. This means that the test case is created by making the default leftmost choice for all evaluated selections, which gives a valid and error-free TFTP read transfer. The zero case is intended for ensuring that the connection to the tested TFTP server is functioning correctly.

The second combine is labelled *error-cases*, it has a **masks** section with a single mask `*.<read-error>.?` selecting all children of the `<read-error>` production.

```
section combine {
    property label "error-cases"
    section masks {
        add *.<read-error>.?
    }
}
```

The combine selects the two read transfers from `<read-error>`, both ending with an error PDU. In the first transfer the error is sent after the first data block, in the second transfer the error is send after the last data block. There test cases are also intended more for validating the existence of a proper connection to the TFTP server than actually testing it.

The third combine, labelled *string*, produces the first robustness test group.

```
section combine {
    property label "string"
    section masks {
        add *.<A-string>.?
    }
}
```

This group contains a test case for each string anomaly from the `<A-string>` tried in the locations where the `<A-string>` has been added. There are 30 different anomalies and the anomaly is in four distinct locations (the file name, the mode, and the two error messages). This gives a total of 120 test cases for the group.

The next combine, label *integer*, creates a test group using the `<A-16>` anomaly production. There are 17 anomalies in eight locations giving a total of 136 test cases.

```
section combine {
    property label "integer"
    section masks {
        add *.<A-16>.?
    }
}
```

Now all single-anomaly test cases have been created. Multiple masks are used to create the next test group, labelled *no-mode-and-filename*.

```
section combine {
    property label "no-mode-and-filename"
    section masks {
        add *.<FILE-NAME>.*.<A-string>.?
        add *.<MODE>.*.<A-string>.0
    }
}
```

The first mask selects all `<A-string>` anomalies for `<FILE-NAME>`. As itself, the test cases generated by this mask would be already covered by test cases generated by the *string* test group. However, the second mask forces the mode to be replaced by the symbol `<A-string>.0`, which is a null symbol. Thus, the test cases generated by the combine contain an anomalous file name and are simultaneously missing the mode field.

The last three test groups combine invalid operation codes in three different locations (in the read request message and in the two error messages) with anomalous content from the production `<A-string>`. The first combine is labelled *operation-code-and-overflow-1* and it mutates the read request message. The remaining two combines are quite similar and not shown here.

```
section combine {
    property label "operation-code-and-overflow-1"
    section masks {
        add *.<RRQ>.0.*.<A-16>.?
        add *.<RRQ>.2.*.<A-string>.?
    }
}
```

Table 8 summarises the test groups from the TFTP test suite. There are eight different test groups having total of 1819 test cases. The last three test groups are large, 510 test cases, and form the majority of the test cases. This happens because they combine two masks which both match a large set of anomalies.

6.3.5 Post-Selection Section

The operations in the **postSelection** section are executed separately for the grammar of each test case. The section should contain at least the procedure **evaluateRules** to evaluate the grammar and perform the test case.

In the TFTP test suite the **postSelection** section contains a few procedures for cleaning up the grammar before evaluation. The three **cutSelections** procedures remove selections from the outgoing PDUs. This is not necessary, but eases human inspection of the test cases, because all irrelevant details are removed. The procedure **cutNames** removes all names, which are not part of the simulation tree, i.e. names which will not be encountered during evaluation.

```
section postSelection {
    cutSelections <RRQ>.*
    cutSelections <ACK>.*
    cutSelections <ERROR>.*
    cutNames
```

After the cleanup, procedure **show** is used to print the grammar in the toolkit log. Then the test case is evaluated, by the procedure **evaluateRules,** and finally the resulting grammar is also added to the log.

```
    show
    evaluateRules
    show
}
```

The full configuration script is shown in Appendix E. In addition to the explained sections, the script contains configuration sections for test controlling and logging.

Table 8. Test groups of the TFTP test suite.

Label	Test case number(s)	Description
zero-case	0	Valid and error-free read transfer
error-cases	1–2	Valid, but error terminated transfers
string	3–122	String anomalies
integer	123–258	Integer anomalies
no-mode-and-filename	259–288	The read request without mode and with filename anomaly
operation-code-and-overflow-1	289–798	Invalid operation code in the read request with content anomaly
operation-code-and-overflow-2	799–1308	Invalid operation code in the first error PDU with content anomaly
operation-code-and-overflow-3	1309–1818	Invalid operation code in the second error PDU with content anomaly

7. Results

This chapter presents five test suites and the results from tests conducted using them. The implemented prototype mini-simulation toolkit was used in the design of the test suites and in test execution. Implementation of the prototype toolkit, the creation of the test suites, and the execution of the tests had multiple goals:

1. Validation and development of the testing method. The method is not usable, if we cannot use it to find real vulnerabilities.

2. Collection of quantitative information about implementation security of contemporary software products.

3. Promote discussion about vulnerability of modern software products.

4. Offering of the test suites to the public. Public test suites can be used to improve the quality of software under development or for assessing the quality of shipped products.

The following subchapters give statistics about the prototype toolkit and an overview of the test suites. For the details of test results refer to Appendix F.

7.1 Overview

A prototype toolkit was implemented to study the developed testing method in practice. The toolkit contained both the mini-simulation method and extensions for functional robustness testing.

The toolkit itself is a relatively small set of components as seen from Table 9, which lists some metrics from the Java and Tcl code of the toolkit. The lines of source code metric is calculated with comment lines, but excluding empty lines. The java class file sizes are summed from class files compiled with the compiler supplied with JDK 1.2.2 (Solaris) without debug information or optimisations, i.e. “`javac -g:none sourcefile.java ...`”

Table 9. Size of the prototype tool.

Java	Number of source files:	192
	Lines of source code:	29 214
	Number of class files:	205
	Total size of all class files (bytes):	553 865
Tcl	Number of source files:	1
	Lines of source code:	318

The created test suites addressed five different application domains using five different protocols. The application domains and the tested protocols are shown in Figure 19. The protocols and tested software components tested were:

- *Wireless Application Protocol* (WAP) gateways. WAP is intended for the integration of telecommunication networks and the Internet. A WAP gateway meditates traffic between WAP terminals and content providers.

- WAP terminals, e.g. WAP phones. WAP users use the terminals to browse the network.

- Hypertext Transfer Protocol (HTTP) clients, e.g. HTTP browsers. The HTTP browsers are used to access the *World Wide Web* (WWW).

- *Lightweight Directory Access Protocol* (LDAP) enabled databases. LDAP is used to provide access to all kinds of information, for management and business purposes.

- The *Simple Network Management Protocol* (SNMP) is used for the management of networked systems and devices.

Each test suite is a collection of test cases designed to reveal implementation vulnerabilities. After the design of each test suite, a *test campaign* was conducted [8, p. 81]. During the campaign the implementation security of a few software or hardware implementations were assessed. Assessment of a single implementation is called a *test run*.

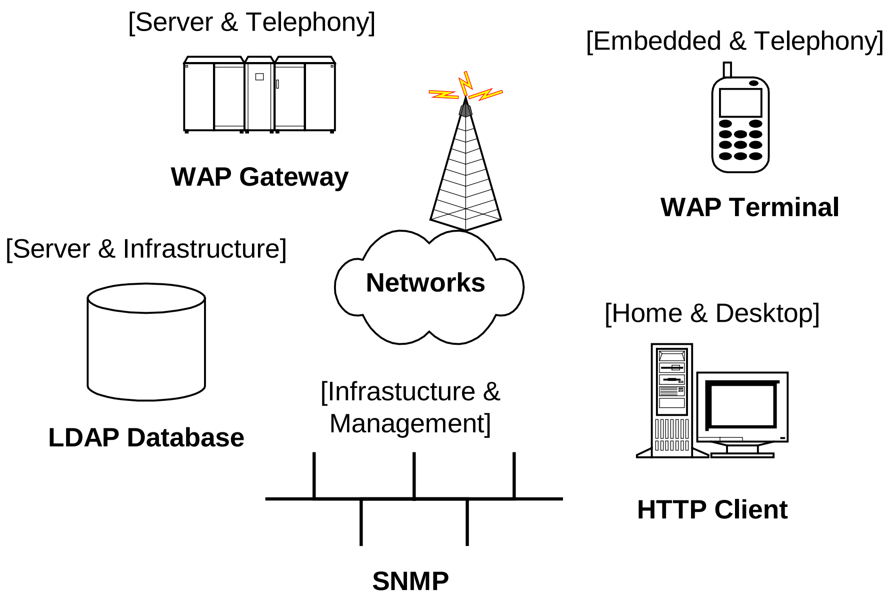


Figure 19. Tested product domains.

A summary of the test suites and test campaigns is given in Table 10, more details are provided in Appendix F. The table shows the number of test groups and test cases in a test suite. A test group corresponds to a single combine in the used mini-simulation configuration. The table also shows the total number of test runs, the number of test runs with fail verdicts, and the number of constructed exploits.

A test run received a pass verdict, if no vulnerable behaviour was observed on any of the test cases. When denial-of-service or other vulnerabilities were observed, the test run received a fail verdict. The fail verdict was assigned to 40 out of 49 test runs, i.e. 82% of the tested implementations were vulnerable to at least denial-of-service. A total of 14 exploit demonstrations were constructed for some of the implementations containing buffer overflow vulnerabilities, to prove the seriousness of the problems. Each exploit made it possible to run remotely supplied code on the host system. The number of exploits could potentially have been larger, if more resources would have been used on the demonstrations.

Table 10. Summary of test suites.

Test suite		Test groups	Test cases	Test runs	Test runs with failures	Exploits	Note
WAP-WSP-Request		39	4 236	7	7	4	
WMLC		84	1 033	10	10	2	
HTTP-Reply		115	3 966	12	6	2	
LDAPv3		93	12 649	8	6	4	a)
SNMPv1	request	118	29 516	8	7	1	b)
	trap	100	24 100	4	4	1	c)
		549	75 500	49	40	14	

a) LDAPv3 test suite is made up of application tests (77 groups) and encoding tests (16 groups).

b) SNMP request tests are divided into applications tests (61 groups) and encoding tests (57 groups).

c) SNMP trap tests are divided into applications tests (76 groups) and encoding tests (24 groups).

Appendix F gives more detailed test run statistics. For test runs in LDAPv3 and SNMPv1 test suites the exact verdicts for all test cases were impossible to determine due to difficulties in test automation. The number of test cases causing IUT failures was too large for full manual test execution. For this reason, the appendix shows only the number of test groups containing at least one test case which received the fail verdict.

The total effort spent on the preparation, design, execution, and analysis of the tests is ca. 24 man months (average 4,8 man-months per test suite). This effort also includes a considerable amount of work not presented in this publication. Due to this, the 24 man-month figure only gives a rough upper limit for the required effort.

In all test suites the focus is on the robustness of tested implementations, the overall security of system based on the tested products is not assessed. However, a single vulnerable point is sufficient to totally compromise the security of a system. Some of the presented material is also available from the PROTOS project web page [56]. Names of the products and vendors are excluded from both presentations.

7.2 WAP-WSP-Request Test Suite

WAP is a family of protocols for delivering advanced data services and Internet content to wireless terminals [78]. A WAP gateway mediates the traffic between the terminals and the actual content providers, e.g. WAP and HTTP servers. The relevant components of the WAP infrastructure are shown in Figure 20. The transportation mechanism between terminals and gateways is UDP.

WAP is an interesting subject for security analysis since it is an attempt to integrate telecommunication networks, which have very high dependability requirements, to the open Internet. The security of a WAP gateway is essential since even encrypted WAP traffic will be exposed as plain text inside the gateway.

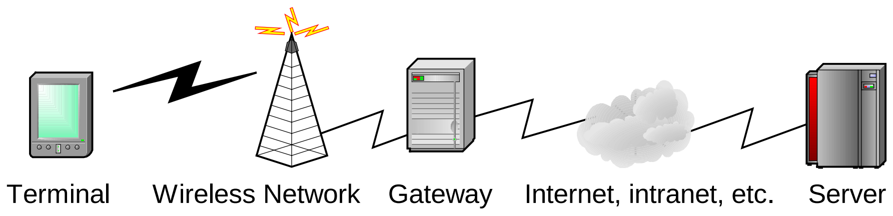


Figure 20. WAP infrastructure relevant for the WAP-WSP-Request test suite.

The purpose of the *WAP-WSP-Request test suite* is to assess the ability of a WAP gateway to handle maliciously formatted *Wireless Session Protocol* (WSP) messages [41]. Using a computer with a modem and a phone an intruder can send malicious WSP messages to a gateway. From the seven tested WAP gateways all were found to be vulnerable to denial-of-service attacks. Exploits taking advantage of the buffer overflows were constructed against four gateways.

7.3 WAP-WMLC Test Suite

The *WAP-WMLC test suite* is a complementary test suite for the WAP-WSP-Request test suite. A WAP terminal is used to access the WAP-enabled network through a WAP gateway [78]. Most WAP terminals are portable (e.g. phones and palm computers), but desktop browsers are also available. The *Wireless Markup Language* (WML) defines the content of WAP pages. WML is conceptually similar to HTML. A compact version of WML, called WML *Compiled* (WMLC), is defined to minimise bandwidth usage. The WML pages can be converted from HTML pages or created directly using WML. The compression from WML into WMLC can be done on-the-fly or beforehand. A server can submit compressed pages directly.

The purpose of the WAP-WMLC test suite is to analyse the robustness of WAP terminals in the handling of WMLC data. Malicious WMLC pages can be injected directly by a content provider since they pass through the WAP gateway without modifications. A total of six of the tested implementations were WAP phones and four were desktop browsers. All 10 implementations were found to contain denial-of-service type of vulnerabilities. Exploits beyond denial-of-service were constructed against two desktop browsers.

7.4 HTTP-Reply Test Suite

HTTP is a protocol for distributed, collaborative, hypermedia information systems [24]. It was originally designed to retrieve hypertext information but has since been used for variety of tasks such as name servers and distributed object management systems. The WWW is heavily based on the use of HTTP.

An HTTP client sends requests to HTTP servers, which return replies according to the requests. The communication protocol used is the *Transmission Control*

Protocol (TCP). End-user clients are often called WWW browsers. An HTTP *proxy* is an intermediate node which acts as both an HTTP server and a client to make requests on behalf of other HTTP clients. Proxies can be used, e.g. for caching WWW pages.

The *HTTP-Reply test suite* consists of exceptional HTTP replies to test the robustness of HTTP reply processing. Seven HTTP browsers and five HTTP proxies were tested. Six of the tested browsers failed in some of the test cases, but none of the tested proxies exhibited vulnerable behaviour. The buffer overflow vulnerabilities of two browsers were exploited to demonstrate full compromises.

7.5 LDAPv3 Test Suite

LDAP is a protocol to retrieve and manage directory information, e.g. email-addresses, phone numbers, and public key information [76]. The underlying database is exposed to intrusion attempts through the LDAP interface. An LDAP client makes requests to the LDAP server which processes the request and returns replies. The communication protocol is TCP. LDAP messages are specified by ASN.1 with BER transfer encoding. A sender must encode BER messages before sending and the correspondingly receiver must decode them before processing.

The *LDAPv3 test suite* is designed to assess the ability of an LDAP server to handle maliciously formatted requests from a rogue LDAP client. The “v3” part in the name comes from the newest version of LDAP, version 3. The test suite contains mutated LDAP *search request* messages. The test cases are divided into two parts, *encoding* test cases and *application* test cases. The former contains anomalies in the BER encoding of messages and the latter contains anomalies in the LDAP application content. The encoding test cases are aimed at revealing vulnerabilities in the decoder used by the LDAP server, while the application test cases exercise the LDAP application code. The idea is that the two different functions might be developed independently, possibly by different vendors. The ability to test BER-encoded data required ASN.1/BER support for the creation of both illegal and legal BER-encoded PDUs.

A total of eight LDAP-enabled databases were tested, out of which six were found vulnerable to at least denial-of-service. The exact verdicts for all test cases were difficult to gather since the controlling of the IUTs and observation of test case behaviour had to be performed mostly manually. The verdicts in six test runs are inconclusive for some of the test cases, since there were too many failures to run the full test suite. In two test runs the number of test groups containing failed test cases is unknown and only the lower bounds can be provided. Exploits were constructed taking advantage of buffer overflow vulnerabilities in four tested databases.

7.6 SNMPv1 Test Suite

SNMP is a protocol for the transportation of network management information [15]. The SNMP traffic takes place between network *management stations* and management *agents*. The agents are located in managed network elements. Management stations use SNMP to set or get (retrieve) values of variables located in the network elements. SNMP requests are sent from management stations to the agents which reply back to the management station. SNMP *traps* are used by agents to signal condition changes to the management stations. SNMP takes advantage of ASN.1 and BER as a protocol specification and encoding method. The transportation method is UDP.

The *SNMPv1 test suite* assesses the ability of SNMP management stations and agents to tolerate maliciously formatted SNMP version 1 messages. As in LDAP, the tests were divided into encoding tests and application tests. Furthermore, the tests were divided into SNMP *request* tests and SNMP *trap* tests, giving a total of four different test categories: request/encoding, request/application, trap/encoding, and trap/application.

Some of the tested implementations were network devices dedicated to forwarding and screening network traffic. A total of eight agents were tested using anomalous request messages, only one of them received a pass verdict. Four management stations were tested using the trap tests, all stations received a fail verdict. Buffer-overflow exploits were constructed against one agent and one management station.

8. Analysis

This chapter provides an analysis of the applicability of the presented testing and modelling approaches for robustness testing. Some thoughts about the quality of contemporary software are given in light of the presented test results. In the last subchapters some of the difficulties and dead-ends of the PROTOS-project are explained and a few open issues are collected.

8.1 Mini-Simulation Method

Traditional software modelling and testing methods are not by themselves suitable for flexible test data creation. Heavy tool support and the mastering of many skills seem to be required. Versatile debugging and robustness testing activities are not promoted. An alternative approach, the mini-simulation method, is presented in this study.

Higher-order attribute grammars provide a solid base for mini-simulation models applicable for debugging and testing purposes. The higher-order attribute grammars are suitable for both behaviour modelling and syntax definitions. The operations with paths and masks are used for grammar mutations. The presented method provides a powerful approach for the explicit derivation of interaction models and promotes reuse of the master specifications. The grammar can be kept simple by leaving the modelling of complex structures to the semantic rules. A direct link to the external world is provided by communication rules.

Security assessment using robustness testing can take advantage of the features of the mini-simulation method. The test design starts from the protocol specification, mutates it with operations, and combines the suitable elements into test cases. The design process is fully visible without any gaps in the path from the protocol specification into each test case. A large number of test cases can be constructed with ease. The generation of test cases with anomalous messages or invalid message exchanges is feasible.

The test campaigns conducted using the prototype tool were effective and found vulnerabilities from 84% of the tested products. However, the content of the test cases is totally determined by the test designer. The knowledge and experience of the test developer may become a limiting factor.

8.2 Scope of the Analysis

In order to interpret the test results presented in chapter 7, one must answer a fundamental question. *What does it mean if a software component fails in 0, 3, 100, or 300 test cases from a total of 1000?* A low number of failures may result if the tested product had good quality, but also if the tests were not effective. A high number of failures clearly demonstrates the presence of problems, but the comparison of numbers between two different products is dubious. Both products may be equally vulnerable, but the tests only triggered the flaws from one. This problem is emphasised if the other product is used during test case design as the trial target. The test cases may become biased for finding flaws from the trial target and be less efficient against other products.

The code coverage analysis provides information about the proportion of code actually executed. A suite which executes only a few percent of the code of a component is likely to miss most of the existing vulnerabilities. This means that the component may not become significantly more secure by fixing the exposed vulnerabilities only. Even very high coverage does not give a full guarantee of robustness, since the successful execution of a code statement does not mean that it is flawless. Indeed, some of the products which failed the tests presented in chapter 7 have probably received unit testing of near 100% code coverage. These tests did not seek for vulnerabilities, so they remained hidden.

Until more research on the topic is conducted, the final merits of robustness testing as a security assessment method are unknown. It is safe to argue that a test suite samples the robustness of the tested component and, at best, the results represent the quality of the whole component. This assumption may be realistic provided that:

1. Tests do reveal vulnerabilities, if they exist.

2. The executed code part reflects the overall quality of the product.

The confirmation of the first assumption requires that we compare test results with material from other sources, e.g. from vulnerability histories or inspection results. The second assumption is difficult to probe without a source code analysis of the tested implementation.

Publicly disclosed vulnerabilities are often found by an ad-hoc manual analysis which probes for well-known vulnerabilities. This analysis would be less successful, if products would be systematically analysed for the presence of the most obvious vulnerabilities. The major benefit from a wider use of robustness testing might be the elimination of obvious vulnerabilities from products in their early stages, which would cut down on the number of publicly disclosed vulnerabilities found by ad-hoc analysis.

All testing suffers from the *pesticide paradox*: The programmers eventually learn to avoid the flaws that testers can find, rendering testing in some sense useless [9, p. 9]. Wide use of robustness testing would probably bring another kind of problem as well: The underground community would shift focus from the vulnerabilities covered by the tests to vulnerabilities not tested for. This could lead to a circle of extending tests to the new vulnerability types followed by a counter-move from the underground community. However, in this circle *software quality would be improving*, in contrast to the current state of affairs where similar vulnerabilities reappear again and again.

It is worth to emphasise again that the presented security assessment does not cover design vulnerabilities, deliberately added backdoors, or other vulnerabilities having complex failure modes. We cannot test for vulnerabilities that we cannot both trigger and detect.

8.3 Applicability

A *metric* is a measurement which relates to a software system, process, or related documentation [68, p. 598]. Robustness testing promises to provide a repeatable and quantitative metric about the quality and security of tested components. The metric can be combined from various parts, such as the number of fail verdicts, the severity of discovered problems, the number of separate underlying flaws, etc. This kind of metric may be used for a variety of purposes. [42]

- Creation of a robustness *benchmark*. A benchmark is a standard against which measurements or comparisons are made [33, p. 12]. A product may pass or fail the robustness benchmark.

- Measuring changes in robustness between different versions of the same software component. It can also provide feedback about the effectiveness of the applied quality improvement actions.

- Quantitative comparison of several products, e.g. for purchase decisions. The product with a suitable ratio between features, quality, and price can be selected.

- Locating the weakest components from a system. The weakest components may be replaced or hardened to improve system-wide security.

The limitations of the presented approach must be weighted against the aforementioned applications. The calculated metric should not be considered as an absolute value of “goodness” or “badness” of the IUT. Passing the test is by no means an indication of definitive security or even bullet-proof robustness. The tests may be biased to find flaws from a product while missing them from an another. The pesticide paradox may fool us to think that the quality of software is improving, but in reality the tests have just become inefficient.

8.4 Implementation Quality and Security

The quality of contemporary software seems to be low in the robustness respect. This statement can be drawn, not only from the test results of chapter 7, but from other similar results as well [42; 50; 65]. Many widely deployed software components cannot withstand a systematic analysis for vulnerabilities. In many cases, the vulnerabilities can be exploited to totally compromise the system hosting these components. The results from the testing of WAP terminals did not indicate that the quality of software in the embedded systems would be better than the quality of other kinds of software. The observed quality does not meet high security requirements.

There is a lack of functional security testing standards or widely adopted conventions or tools. The efforts used for improving quality have focused on improving the process of making software rather than to the product itself [74, p. 14; 68, p. 591]. In functional testing the focus has been on rigid requirement-based conformance testing. Also, the effectiveness and reach of the functional techniques for security assessment has been, quite rightly, questioned [63].

Improvements in the software design process and in programming languages are used to push the *complexity barrier* further [9, p. 9]. The complexity barrier is the ultimate limit of complexity we can handle and still produce an operational system. New features inevitably add to complexity and require more lines of code. The more code there is, the more opportunities there are for programmer mistakes [63]. Some of the mistakes are likely to result in vulnerabilities. A security feature added to a system may also make the system less secure by introducing new implementation vulnerabilities. This means that a secure system should be designed to provide the required functionality *as simply as possible*. A simpler system is likely to contain less vulnerabilities than a complex system. Assessing the properties of the simple system is cheaper and a more throughout analysis is possible. No serious improvements in software security can be expected before the goal is changed from new features to true security.

Wide use of security assessment, e.g. robustness testing, could create a pressure to create high-quality software. Vendors would have to invest in the true security in their products, if they know that potential customers or some third parties may analyse their products independently. The overall security of software would be better and the number of publicly disclosed vulnerabilities would decrease. Smaller numbers of vulnerabilities revealed after shipment would result in less security advisories and patches. Administrators and users are more likely to apply patches if they are less frequent. There might be a possibility that all this leads to more up-to-date systems having less security problems than nowadays.

The huge number of security problems, and the fact that the number of problems is rather increasing than decreasing, has given space for pessimism. The software industry has its roots in trusted isolated systems and the state of affairs has fundamentally changed with the introduction of heterogeneous systems interconnected by inherently insecure networks. Some believe that traditional techniques simply cannot address the challenge. As said in a paper by Blakley [11]:

> "... the traditional model of computer security is no longer viable, and ... new definitions of the security problem are needed before the industry can begin to work toward effective security in the new environment."

8.5 Difficulties and Dead-Ends

We have found the monitoring and controlling of tested implementations a problematic issue. A fully automatic test environment would require the ability to start and stop the IUT and observe IUT behaviour without the presence of an operator. The observations must be linked into test cases for verdict assignment. The testing must be suspended while the IUT has stopped responding and continue after it is again up and running. All this requires a considerable amount of software which may be operating system dependent.

As an additional hurdle the programs tend to catch fatal failures, e.g. illegal memory accesses, branch a new thread, and continue "business as usual". Detecting such situations is problematic and may require kernel-level modifications to the operating system. Unfortunately, this failure hiding does not necessarily prevent the exploitation of the vulnerability, so an overly positive impression of program quality and security may be perceived.

In our earliest attempts we tried to fully automate the generation of test cases from the protocol specification. It become quickly clear, that the goal was too ambiguous, at least as a starting point. The design of effective test cases requires an understanding of the protocol semantic structure, which cannot be determined from the syntax specification alone. In the next attempt we provided *semantic clues* embedded into the specification for the test case generator. A clue pointed out symbol types, separators, file names, length fields, etc. from the specification. The resulting method was quite rigid and we found ourselves adding as many different clue types as there were potential locations for anomalies. In the current approach the anomalies are directly inserted as alternatives for the original protocol elements. Natural naming of the anomaly identifiers and verbose comments provide the required “clues” for human understanding.

8.6 Open Issues

The ultimate value of security assessment by robustness testing is unknown. Some experiments would bring more light to this issue:

1. Comparing test results from two independent test suites created by different teams who have used different trial implementations. Would both tests reveal the same set of problems?

2. Comparing vulnerabilities discovered using a test suite to results from other kinds of sources, e.g. to code reviews or to incident histories. Would the tests reveal vulnerabilities not found by other means? Would the tests fail to reveal vulnerabilities which can be found by other means?

3. Comparing the number of publicly disclosed vulnerabilities from products hardened by the help of robustness testing to products which have not received any special hardening. Does the hardening lessen the number of publicly disclosed vulnerabilities?

The optimum location of robustness analysis embedded into the software development process (e.g. see *Figure 1*) is an open question. Implementation vulnerabilities should be visible already in individual program units and the robustness analysis can be a part of unit testing. Additionally, robustness test cases might be usable for revealing interoperability problems between software modules. This would be a good reason for running robustness tests in later phases as well. The robustness analysis might also be a part of the acceptance tests.

This study has said only a little about the actual design of effective test cases. The discussion in chapter 2.1 about implementation vulnerabilities and domain testing and syntax testing in chapter 0 provide some ideas. Still, the systematic enumeration of effective anomalies and their best locations in PDUs remains to be done. Also, the applicability of formal techniques for robustness test design should be studied. Mini-simulation testing may also be used for the testing of software properties beyond robustness, such as timing problems, interface mismatches, etc.

The applicability of standard testing methods and notations for security assessment should be analysed. Surely test cases could be specified using, for example, TTCN, but test case design cannot use the approach as presented in this study, since there are no methods for mutating a TTCN specification. The other side of the coin is the applicability of the mini-simulation method to traditional requirements-based conformance testing. The strong points of the mini-simulation toolkit, one simple notation, powerful operations, and easy test case selection would probably be as useful for traditional testing as well. An input/output oracle should be able to be produced in a similar fashion as test cases are combined.

9. Summary

The increasing dependence of modern society on computer networks and the emergence of new critical applications emphasise the importance of information security. A growing number of intrusion attempts is expected as more systems become connected to open networks. Up till recently, security may not have been a good selling point, most products have been bought for their novel features, not for their high security. Even if customers have demanded secure software, there often have been no practical methods to verify the claims of the vendor. The few generally applicable methods require access to source code, which is not often delivered to the customer. There is a lack of security assessment methods that are applicable without source code.

Software vulnerabilities expose a system to malicious intrusions. Often, information security is only seen as a collection of cryptographic algorithms and protection components. In reality, security is also a quality attribute built inside the systems. The vulnerabilities are caused by decisions and mistakes made during software design and implementation. The complexity of a system is determined at the time of the design and has a great impact on the resulting security. A complex system is likely to be difficult to implement and use. Implementation vulnerabilities result from programming mistakes, they make up a large portion of the vulnerabilities reported to the public.

From a security point of view, the robustness of software components is an important quality. Robustness is the ability to tolerate unexpected and anomalous input and stressful environmental conditions. Failures in robustness are at least denial-of-service problems. Certain kinds of robustness problems, especially buffer-overflows in C and C++ programs, can be further exploited to execute malicious code in the vulnerable system. Robustness problems usually have direct failure modes like crashing or hanging. Despite this there are only a few tools available for functional robustness analysis. The available testing methods are mainly focused on requirement-based testing of software and exclude quality factors like robustness.

Software testing assesses that a software component meets the requirements or holds some other desirable properties. Functional (black-box) testing is based on the interface specification of the component and can be performed without source code access. Structural (white-box) testing requires access to internals of the component. The functional testing approach is more universal since it can be applied to any software component. Different functional testing approaches have been proposed.

- **Transaction flowgraph testing**. Test design is based on a graph defining the expected behaviour of the component using nodes and edges. A node represents a status or state of the component. An edge is a transition between nodes. Ideally the testing covers the whole graph.

- **Domain testing** and **syntax testing**. Test design is based on the range and syntax of the input elements the component accepts. Potentially problematic input values are tried in order to expose flaws in the component implementation.

- **Fault injection using interfaces**. The component interfaces are used to inject anomalous, unexpected, or even illegal input to the tested component. The purpose of fault injection is to study the impact of faults on the system.

Robustness testing can take advantage of all the above approaches. An attractive property of robustness testing is that the correctness of the responses from the tested component does not have to be checked. Test verdicts can be based solely on whether the component behaved in a secure fashion or whether there were indications of possible vulnerabilities. The freedom from response validation greatly reduces the effort required for testing.

Functional testing of a protocol entity requires modelling of the other peers participating in the protocol, e.g. testing of a server requires the model of a client. Behavioural modelling is often based on extensions of finite state machines, regular expressions, or context-free grammars. Several standard methods are available for software specification and testing activities. Different methods emphasise different aspects:

- **SDL** provides tools for both the structural and behavioural decomposition of the modelled entity. Emphasis is on specification of the entity.

- **MSC** specifies interaction sessions between two or more protocol entities. The order and the relationships of the exchanged messages is emphasised.

- **UML** provides a means for entity and interaction specification.

- **TTCN** is aimed for the description of conformance tests. It provides a rich set of behaviour description elements and is aimed for executable interactions.

- **ASN.1** is a notation for the syntax modelling of data and messages. Multiple platform independent encoding conventions are specified.

- **XML** is a method for describing data structures and relationships. It can be used as a platform independent data transfer method.

The use of these methods for testing is a complex process. The process starts from entity behaviour specification (e.g. SDL), which is translated into interaction specifications of each test case (e.g. TTCN). An interaction specification has to be made executable by specifying network addresses, etc. A separate syntax specification language (e.g. ASN.1) may be required. The developer has to master all notations, translations, and the execution environment. In all, the process is not optimal for versatile robustness testing and debugging.

As an alternative, this study described the mini-simulation method based on attribute grammars. The mini-simulation method is a simple and light-weight approach for software modelling. A mini-simulation is derived from a master specification by using a configuration script. The simulated protocol is described using an attribute grammar augmented with semantic and communication rules. The attribute grammar is applicable for both syntax modelling and behavioural modelling. The semantic rules are used to calculate structures unfeasible to be modelled using pure grammar, such as lengths and checksums. External communication is provided by communication rules. A prototype mini-simulation toolkit has been implemented using Java.

A configuration script contains operations to mutate the simulation grammar, for adding the required rules, and to set parameters. Multiple simulation grammars can be created for different purposes from the same master specification. The evaluation engine is used to execute the simulation grammar. The engine traverses the grammar and evaluates the encountered semantic and communication rules. External interaction takes place as a side-effect of the evaluation of communication rules. The result of the evaluation is an output grammar. Since it uses the same notation, it can be re-evaluated to reproduce the same simulation.

The mini-simulation method provides a relatively simple but powerful means for protocol entity simulation. The use of the master specification and operations provides a convenient means for producing different models for different debugging and testing purposes. The effective creation of messages with both legal and illegal elements is possible. The models are readily executable and can interact with real-world products and components.

A robustness testing method was further developed from the mini-simulation method. The idea is to mutate the master specification to contain anomalous, unexpected, or illegal protocol elements targeted for finding flaws from the IUT. The anomalies are combined with normal protocol elements to form test cases. The mini-simulation toolkit automatically generates the messages required in the test cases. The messages are semantically valid apart from the added anomalies.

Five test suites have been created with the mini-simulation testing method and used to test a total of 49 different available products, see Table 10. The products represent various application domains from telecommunication end-user terminals to network maintenance including embedded devices. A total of 40 tested products were found to be potentially vulnerable, at least to denial-of-service attacks. 14 tested products were proven to contain buffer-overflows exploitable to the point of total compromise of the host system. The security of software products seems to be low, at least in the robustness respect.

The dominance of rigid methods aimed for conformance has not promoted the development of robustness testing methods. The proposed mini-simulation robustness testing method is effective for finding vulnerabilities from contemporary software products, including software in embedded products. Different components implementing the same functionality can be evaluated in a comparable manner. The number of failures provide a quantitative figure, which can be used as a benchmark, as a quality feedback metric, or as criterion for purchase decisions or allocation of system hardening efforts.

However, robustness analysis has its limitations. A low number of fail verdicts may result either because of good quality software or inefficient tests. The differences between the test results of two products may be caused by their quality differences, but the tests may also be biased to find more flaws from the other product. The code coverage of tests may be low, which means that the overall security of the product may not be reflected by the test results. Additionally, the programmers will eventually learn to avoid the flaws which are found by robustness testing and it will no longer find problems. Still, there might be problems not discovered by testing, which the underground community eventually will learn to exploit.

Despite its limitations, functional security testing may have the potential to lessen the number of vulnerabilities reported in the public. A large portion of the disclosed vulnerabilities are variations of well-known vulnerability types. If testing would remove these most obvious vulnerabilities, the underground community would have a harder time discovering new vulnerabilities. Also, if vendors know their products might be analysed without their co-operation, they may invest more effort to ensure the true quality of the products. All this may result in fewer numbers of patches being released to fix serious vulnerabilities in shipped products. The fewer patches required, the more likely is that they are applied and less systems are left vulnerable. All in all, the robustness and security of contemporary software would potentially be improved if methods similar to the one presented in this study would be taken into wide use.

References

1. ActiveState. Tcl Developer Xchange [HTML] [Accessed 2001-08-23] URL: http://tcl.activestate.com/

2. Aho, A. V., Sethi, R. & Ullman, J. D. 1986. Compilers, Principles, Techniques, and Tools. Reprinted with corrections March, 1988. USA. Addison–Wesley Publishing Company. 796 p. ISBN 0-201-10088-6

3. Al-Herbish, T. 1999. Secure UNIX Programming FAQ, Version 0.5. [HTML or Text] [Accessed 2001-08-22] URL: http://www.whitefang.com/sup/

4. Anderson, D. P. & Landweber, L. H. 1985. A Grammar-Based Methodology for Protocol Specification and Implementation. In: Proceedings of the Ninth Symposium on Data Communications. September 10–12, 1985. British Columbia Canada. ACM. Pp. 63–70.

5. Anderson, R. 1993. Why Cryptosystems Fail. In: First Conference of Computer and Communication Security. November 3–5, 1993, Fairfax VA USA. ACM. Pp. 215–227.

6. Bach, J. 1999. Risk and Requirements-Based Testing. IEEE Computer, June, pp. 113–114. ISSN 0018-9162

7. Bace, R. & Mell, P. 2001. NIST Special Publication on Intrusion Detection Systems, Draft Publication, February 12, 2001. [Microsoft Word] National Institute of Standards and Technology (NIST), Computer Security Resource Center. [Accessed 2001-08-23] URL: http://csrc.nist.gov/publications/. 51 p.

8. Baumgarten, B. & Giessler, A. 1994. OSI Conformance Testing Methodology and TTCN. Amsterdam, The Netherlands. Elsevier Science B.V. 328 p. ISBN 0-444-98712-7

9. Beizer, B. 1990. Software Testing Techniques. Second Edition. New York, USA. Van Nostrand Reinhold. 550 p. ISBN 0-442-20672-0

10. Binkley, D. W. 1995. C++ in Safety Critical Systems, NIST IR 5769, November 1995. [PostScript] Gaithersburg. National Institute of Standards and Technology (NIST) [Accessed 2001-08-23] URL: http://hissa.nist.gov/. P. 31

11. Blakley, B. 1996. The Emperor's Old Armor. In: Proceedings of the UCLA Conference on New Security Paradigms Workshops, September 17–20, 1996, Lake Arrowhead, CA, USA. ACM. Pp. 2–16. ISBN 0-89791-944-0

12. Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Maler, E. 2000. Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000. World Wide Web Consortium (W3C). 59 p.

13. Bruschi, D., Rosti, E. & Banfi, R. 1998. A Tool for Pro-Active Defense Against the Buffer Overrun Attack. In: Quisquater, J.-J., Deswarte, Y., Meadows, C. & Gollmann, D. (Eds.) Computer Security – ESORICS'98 Lecture Notes in Computer Science No. 1485, Fifth European Symposium on Research in Computer Security, September 16–18, Louvain-la-Neuve, Belgium, 1998, Proceedings. Springer–Verlag. Pp. 17–31. ISBN 3-540-65004-0

14. BugTraq. 1993. [E-mailing list] SecurityFocus.com. [Accessed 2001-08-23] Subscriptions at URL: http://www.securityfocus.com/

15. Case, J., Fedol, M., Schoffstall, M. & Davin, J. 1990. Request for Comments: 1157 A Simple Network Management Protocol (SNMP). May 1990. IETF. 36 p.

16. Carreira, J. V., Costa, D. & Silva, J. G. 1999. Fault Injection Spot-checks Computer System Dependability. IEEE Spectrum, August, pp. 50–55. ISSN 0018-9235

17. Cigital [Accessed 2001-08-23] URL: http://www.cigital.com/

18. Coleman, D., et al. 1994. Object-Oriented Development The Fusion Method. Englewood Cliffs, New Jersey. Prentice-Hall. 332 p. ISBN 0-13-338823-9

19. Cowan, C. & Pu, C. 1998. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In: Proceedings of the 1998 Workshop on New Security Paradigms, September 22–26, 1998. Charlottesville, VA USA. ACM. Pp. 54–70. ISBN 1-58113-168-2

20. Cowan, C., et al. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the Seventh USENIX Security Symposium, January 26–29, 1998. San Antonio, TX USA. The USENIX Association. Pp. 63–78.

21. Crocker, D. H. 1982. Request For Comments: 822 Standard For The Format of ARPA Internet Text Messages, August 13, 1982. IETF. 47 p.

22. Devanbu, P. T. & Stubblebine, S. 2000. Software Engineering for Security: a Roadmap. In: Proceedings of the Conference on the Future of Software Engineering (ICSE 2000), June 4–5, 2000. Limerick, Ireland. ACM. Pp. 227–239. ISBN 1-58113-253-0

23. Ellsberger, J., Hogrefe, D. & Sarma, A. 1997. SDL Formal Object-Oriented Language for Communicating Systems. Great Britain. Prentice-Hall Europe. P. 312. ISBN 0-13-632886-5

24. Fielding, R., et al. 1999. Request for Comments: 2616 Hypertext Transfer Protocol – HTTP/1.1, June 1999. IETF. 176 p.

25. Fink, G. & Levitt, K. 1994. Property-Based Testing of Privileged Programs. In: Proceedings of the 10th Annual Computer Security Conference, 5–9 December, 1994. Orlando, FL USA. Pp. 154–163.

26. Ghosh, A. K., O'Connor, T. & McGraw, G. 1998. An Automated Approach for Identifying Potential Vulnerabilities in Software. In: Proceedings of the IEEE Symposium on Security and Privacy, May 3–6, 1998. Oakland, CA USA. IEEE. Pp. 104–114.

27. Ghosh, A. K. & McGraw, G. 1998. An Approach for Certifying Security in Software Components [PDF] Proceedings of the 21st National Information Systems Security Conference, October 5–9, 1998. Crystal City, VA, USA. National Institute of Standards and Technology (NIST). [Accessed 2001-08-23] URL: http://csrc.nist.gov/nissc/1998/

28. Ghosh, A. K. & O'Conner, T. 1998. Analyzing Programs for Vulnerability to Buffer Overrun Attacks [PDF] Proceedings of the 21st National Information Systems Security Conference, October 5–9, 1998. Crystal City, VA, USA. National Institute of Standards and Technology (NIST). [Accessed 2001-08-23] URL: http://csrc.nist.gov/nissc/1998/

29. Ghosh, A. K. & Voas, J. M. 1999. Inoculating Software for Survivability. Communications of the ACM, Vol. 42, Issue 7, pp. 38–44. ISSN 0001-0782

30. Gollmann, D. 1999. Computer Security. New York, USA: John Wiley & Sons. 320 p. ISBN 0-471-97844-2

31. Gough, K. J. 1988. Syntax Analysis and Software Tools. Great Britain. Addison–Wesley. 459 p. ISBN 0-201-18048-0

32. Howard, J. D. & Longstaff, T. A. 1998. A Common Language for Computer Security Incidents, Sandia Report SAND98-8667. Printed October 1998. USA. Sandia National Laboratories. 26 p.

33. IEEE Std 610.12-1990. 1991. IEEE Standard Glossary of Software Engineering Terminology, Corrected Edition, February 1991. IEEE. 83 p.

34. ITU-T Recommendation Z.100 (11/99). 1999. Specification and Description Language (SDL). International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

35. ITU-T Recommendation Z.120 (11/99). 1999. Message Sequence Chart (MSC). International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

36. ITU-T Recommendation X.208 (11/88). 1988. Specification of Abstract Syntax Notation One (ASN.1). International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

37. ITU-T Recommendation X.209 (11/88). 1988. Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

38. ITU-T Recommendation X.290 (04/95). 1995. OSI Conformance Testing Methodology and Framework for Protocol Recommendations for ITU-T Applications – General Concepts. International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

39. ITU-T Recommendation X.292 (09/98). 1998. OSI Conformance Testing Methodology and Framework for Protocol Recommendations for ITU-T Applications – The Tree and Tabular Combined Notation (TTCN). International Telecommunication Union, Telecommunication Standardization Sector (ITU-T).

40. Jones, R. W. M. & Kelly, P. H. J. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In: Kamkar, M. (Ed.) Linköping Electronic Articles in Computer and Information Science, Vol. 2, No. 009. AADEBUG’97. Proceedings of the Third International Workshop on Automatic Debugging, May 26–27, 1997. Linköping, Sweden. [Accessed 2001-08-23] URL: http://www.ep.liu.se/ea/cis/1997/009/. ISSN 1401-9841

41. Kaksonen, R., Laakso, M., & Takanen, A. 2001. Software Security Assessment through Specification Mutations and Fault Injection. In: Steinmetz, R., Dittmann, J., Steinebach, M. (Eds.). Communications and Multimedia Security Issues of the New Century. IFIP TC6/TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01) May 21–22, 2001. Darmstadt, Germany. Pp. 173–183. ISDN 0-7923-7365-0

42. Koopman, P. & DeVale, J. 2000. The Exception Handling Effectiveness of POSIX Operating Systems. IEEE Transactions on Software Engineering, Vol. 26, No. 9, September 2000, pp. 837–848. ISSN 0098-5586

43. Krsul, I., Spafford, E. & Tripunitara, M. 1998 An Analysis of Some Software Vulnerabilities [PDF] Proceedings of the 21st National Information Systems Security Conference (NISSC), October 5–9 1998. Crystal City, VA, USA. National Institute of Standards and Technology (NIST). [Accessed 2001-08-23] URL: http://csrc.nist.gov/nissc/1998/

44. Laakso, M., Takanen, A. & Röning, J. 1999. The Vulnerability Process: a tiger team approach to resolving vulnerability cases. In: Proceedings of the 11th FIRST Conference on Computer Security Incident Handling and Response, 13–18 June 1999. Brisbane, Australia. Forum of Incident Response and Security Teams (FIRST).

45. Laakso, M.,Takanen, A. & Röning, J. 1999. Runtime Symbol Interposition – Infiltrating the Black-box. In: Proceedings of the Eighth Annual Conference on System Administration, Networking and Security (SANS'99), 7–14 May, 1999. Baltimore, USA. System Administration, Networking, and Security (SANS) Institute.

46. Laakso, M., Takanen, A. & Röning, J. 2001. Introducing Constructive Vulnerability Disclosures. In Proceedings of the 13th FIRST Conference on Computer Security Incident Handling. June 17–22, 2001. Toulouse, France. Forum of Incident Response and Security Teams (FIRST).

47. Larmouth, J. 1999. ASN.1 Complete [PDF] OSS Nokalva. 387 p. [Accessed 2001-08-23] URL: http://www.oss.com/

48. McGraw, G. 1998. Testing for Security During Development: Why We Should Scrap Penetrate-and-Patch. IEEE Aerospace and Electronic Systems, 13(4), April 1998, pp. 13–15. ISSN 0885-8985

49. Maurer, P. M. 1990. Generating Test Data with Enhanced Context-Free Grammars. IEEE Software, July 1990, pp. 50–55. ISSN 0740-7459

50. Miller, B. P. et al. 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, April 11 1995. [USA] Computer Sciences Department, University of Wisconsin.

51. Object Management Group (OMG). 1999. Unified Modelling Language (UML) Specification, Version 1.3, June 1999. [Microsoft Word] [Accessed 2001-08-23] URL: http://www.omg.org/

52. Ousterhout, J. K. 1998. Scripting: Higher Level Programming for the 21th Century. IEEE Computer, March 1998, pp. 23–30. ISSN 0018-9162

53. Packetfactory [HTML] [Accessed 2001-08-23]
URL: http://www.packetfactory.net/

54. Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp. 196–255. ISSN 0360-0300

55. Perl.com [HTML] [Accessed 2001-08-23] URL: http://www.perl.com/

56. Project: Security Testing of Protocol Implementation (PROTOS). 1999–. [HTML] University of Oulu, Computer Engineering Laboratory [Accessed 2001-08-23] URL: http://www.ee.oulu.fi/research/ouspg/protos/

57. Rational. Purify: Fast Detection of Memory Leaks and Access Errors. Whitepaper [HTML] [Accessed 2001-08-23] URL: http://www.rational.com/

58. Rudolph, E., Graubmann, P. & Grabowski, J. 1996. Tutorial on Message Sequence Charts. Computer Networks and ISDN Systems, Volume 28, Issue 12, June 1996. Elsevier Science. Pp. 1629–1641. ISSN 0169-7552

59. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. 1991. Object-Oriented Modelling and Design. New Jersey, USA. Prentice-Hall. 512 p. ISBN 0-13-629841-9

60. Sarikaya, B. 1993. Principles of Protocol Engineering and Conformance Testing. Great Britain. Ellis Horwood. 502 p. ISBN 0-13-012642-X

61. Schneier, B. 1998. Cryptographic Design Vulnerabilities. IEEE Computer, September 1998, pp. 29–33. ISSN 0018-9162

62. Schneier, B. 1998. E-Mail Viruses, Worms, and Trojan Horses. Crypto-Gram, June 15, 1998 [e-mail newsletter] Counterpane Internet Security, Inc. [Accessed 2001-08-15] URL: http://www.counterpane.com/

63. Schneier, B. 2000. Software Complexity and Security. Crypto-Gram, March 15, 2000 [e-mail newsletter] Counterpane Internet Security, Inc. [Accessed 2001-08-15] URL: http://www.counterpane.com/

64. Secure Software Solutions. RATS (Rough Auditing Tool for Security). [HTML] [Accessed 2001-08-13] URL: http://www.securesw.com/

65. Shelton. C. P., Koopman, P. & DeVale, K. 2000. Robustness Testing of the Microsoft Win32 API. In: International Conference on Dependable Systems and Networks (DSN 2000), 25–28 June 2000. New York, NY USA. IEEE. Pp. 261–272. ISBN 0-7695-0707-7

66. Smith, N. P. 1997. Stack Smashing Vulnerabilities in the UNIX Operating System. [PostScript] [Accessed 2001-03-12] URL: http://destroy.net/

67. Sollins, K. 1992. Request For Comments: 1350 The TFTP Protocol (Revision 2), July 1992. IETF. 11 p.

68. Sommerville, I. 1992. Software Engineering. USA. Addison–Wesley Publishers. 649 p. ISBN 0-201-56529-3

69. Stanton, S. 1998. Blending Tcl and Java. Dr. Dobb's Journal, February 1998. pp. 50–54. ISSN 1044-789X

70. Stein, L. D. 2001. The World Wide Web Security FAQ. Version 3.1.0, July 28, 2001 [HTML] World Wide Web Consortium (W3C) [Accessed 2001-08-23] URL: http://www.w3.org/

71. Sun Microsystems. The Source for Java Technology [HTML] [Accessed 2001-08-23] URL: http://java.sun.com/

72. Takanen, A., Laakso, M., Eronen, J. & Röning, J. 2000. Running Malicious Code by Exploiting Buffer Overflows: A Survey Of Publicly Available Exploits. In: EICAR 2000 Best Paper Proceedings. First European Anti-Malware Conference, March 4–7, 2000. Brussels, Belgium. EICAR. Pp.158–180

73. Viega, J. & Bloch, J. T. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code [PDF] 16th Annual Computer Security Applications Conference, December 11–15, 2000. New Orleans, Louisiana, USA. 20 p [Accessed 2001-08-23] URL: http://www.acsac.org/

74. Voas, J. & McGraw, G. 1998. Software Fault Injection. New York, USA. John Wiley & Sons. 352 p. ISBN 0-471-18381-4

75. Wagner, D., Foster, J. S., Brewer, E. A. & Aiken, A. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. [PDF] Networking and Distributed System Security Symposium, 3–4 February, 2000. San Diego, California, USA. [Accessed 2001-08-23] URL: http://www.isoc.org/ndss2000/

76. Wahl, M., Howes, T. & Kille, S. 1997. Request for Comments: 2251 Lightweight Directory Access Protocol (v3), December 1997. Internet Engineering Task Force (IETF) 50 p.

77. Wall, L., Christiansen, T. & Orwant, J. 2000. Programming Perl, 3rd Edition. O'Reilly & Associates. 1092 p. ISBN 0-596-00027-8

78. WAP Forum. [HTML] [Accessed 2001-08-23]
URL: http://www.wapforum.com/

79. Welch, B. B. 2000. Practical Programming in Tcl and Tk. New Jersey, USA: Prentice-Hall. 772 p. ISBN 0-13-022028-0

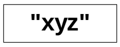
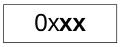
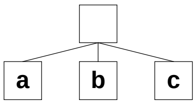
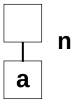
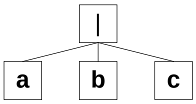
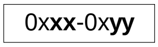
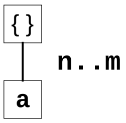
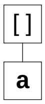
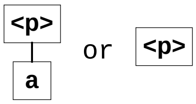
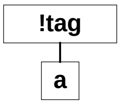
80. Wheeler, D. A. 2001. Secure Programming for Linux and Unix HOWTO, April 2001 [HTML, PDF, PostScript] Linux Documentation Project [Accessed 2001-08-13] URL: http://www.linuxdoc.org/

81. Wheeler, D. A. *Flawfinder*. [HTML] [Accessed 2001-08-23] ULR: http://www.dwheeler.com/

82. Wirth, N. 1977. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, Vol. 20, No. 11, November 1977. ACM. pp. 822–823. ISSN 0001-0782

83. Wojtczuk, R. Defeating Solar Designer's Non-Executable Stack Path. [HTML] [Accessed 2001-03-12] URL: http://area.insecure.org/

Appendix A: BNF and Tree Notations

`"xyz"`		Terminal string **xyz**.
`0xXX`		Terminal octet, hexadecimal value **XX**.
`a b c`		Sequence of child symbols **a**, **b** and **c**.
`n x a`		Repeat sequence, the child symbol **a** is repeated **n** times.
`()`		Empty sequence, i.e. empty symbol or null.
`a|b|c`		Selection of one child symbol, **a**, **b** or **c**.
`0xXX-0xYY`		A range of octets. Equal to selection from octets **xx**, **yy** and all octets between..
`n..m {a}`		Repeat selection, the child symbol **a** is repeated from **n** to **m** times. Default is from zero to infinity.
`[a]`		Optional child symbol **a** . Equals to repeat selection from zero to one, or selection between empty and the symbol.
`<p>`		Identifier **<p>** defined as "**<p> = a**" or "**<p> ::= a**". The child branch can be omitted to simplify presentation.
`!tag`		Tag symbol. Child symbol **a** is tagged.
`...`	● ● ●	Some symbols are omitted.

Appendix B: Operations

The following table summarises the most important mini-simulation operations available for Tcl configuration scripts.

cutNames	Cut unused productions, whose left-hand side identifiers cannot be reached from the root symbol.
cutRules	Cut rules from the grammar.
cutSelections mask	Cut selections by leaning the leftmost choice from selections pointed to by a mask.
data *name symbol*	Define a data production with left-hand side *name* and right-hand side *symbol*.
evaluateRules	Evaluate the grammar.
insert *mask index symbol*	Insert *symbol* as a child to the position *index* of the non-terminal pointed by *mask*
parseBNF *text*	Read BNF specification directly from *text*.
parseBNFFile *filename*	Read BNF specification from file *filename*.
remove *name*	Remove production with left-hand side *name*.
replace *mask* [*symbol*]	Replace the branch pointed to by *mask* with *symbol*. Remove the branch, if *symbol* parameter is not specified.
save *filename*	Save the grammar into file *filename* using BNF.
section evaluateRule ...	Section for configuring a rule and evaluating the grammar with the configured rule.
section insertRule ...	Section for configuring a rule and inserting the configured rule into the grammar.
show	Write the current grammar into the log.
type *name symbol*	Define a type production with left-hand side *name* and right-hand side *symbol*.

Appendix C: Default Rules

The following tables summarise the semantic and communication rules implemented in the mini-simulation toolkit.

Semantic rules

Embed	Embed a symbol into an encoded binary stream.
IntegerField	Encode and decode integer field using ASCII notation, bytes or bits.
IPChecksum	Calculate IP checksum used in various IP-protocol family PDUs.
Length	Calculate a length field with explicit context.
Length2	Calculate a length field.
Offsets	Calculate offsets between symbols in an encoded binary stream.
Padding	Pad a symbol length in encoding.
PrefixPadding	Pad an ASCII numeric field by prefix zeroes.
Reverse	Reverse the order of bytes during encoding and decoding.
SequenceNumber	Add sequence numbers.
StringField	Convert binary sequence into ASCII string during decoding.
Wait	Suspend evaluation for a specified amount of time.

Communication rules

DirectIO	Decode data directly from the configuration script. Encode data to the log.
Evaluator	Decode data from the standard output or the standard error stream of an external process. Encode data to the standard input stream.
FileIO	Decode from a file. Encode into a file.
MemoryIO	Decode from an encoded symbol. Encode from a decoded symbol.
TCPClientSocket	Decode from the input stream of a TCP client socket. Encode to the output stream.
TCPServerSocket	Decode from the input stream of a TCP server socket (or listening socket). Encode to the output stream.
UDPSocket	Decode from a received UDP datagram. Encode into an outgoing datagram.

Appendix D: TFTP Specification

```
# A simple BNF specification of a TFTP transfer

# Model a single transfer
<transfer> = <read-transfer> |<write-transfer>

# Read transfer
<read-transfer> = !up<RRQ> <reads>
<reads> = {!down<BLOCK> !up<ACK>} !down<LAST-BLOCK> !up<ACK>

# Write transfer
<write-transfer> = !up<WRQ> <writes>
<writes> = !down<ACK> {!up<BLOCK> !down<ACK>} !up<LAST-BLOCK> !down<ACK>

# Request PDUs
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

# Subsequent PDUs
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512 x <OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 { <OCTET> }
<ACK>  ::= (0x00 0x04) <BLOCK-NUMBER>
<ERROR> ::= (0x00 0x05) <ERROR-CODE> <ERROR-MESSAGE>

# Miscellaneous productions
<MODE> ::= "octet" 0x00 |"netascii" 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<BLOCK-NUMBER> ::= <OCTET> <OCTET>
<ERROR-CODE> ::= <OCTET> <OCTET>
<ERROR-MESSAGE> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
<OCTET> ::= 0x00 - 0xff
```

Appendix E: TFTP Test Suite Configuration

```
# A simple TFTP test suite Tcl configuration script
# - for assessing the robustness of TFTP servers
# - error handling excluded
# - the download file "sample.txt" size must be between 512...1023 bytes

package require java
lappend auto_path [java::call FI.protos.Root tclLib]
package require configurer
namespace import configurer::*

# Script Body
section driver {
    section preSelection {
        parseBNFFile "tftp.bnf"

        # only read transfer
        replace <transfer> <read-transfer>

        # RRQ followed by successful or erroneous transfer
        replace <read-transfer>.1 {<_OLD> |<read-error>}

        # successful transfer, expecting file size 512...1023 bytes
        replace <reads>.0 <_OLD>.1

        # define transfers ending to error
        data <read-error> {(!down <BLOCK> !up<ERROR>) |
            (!down <BLOCK> !up <ACK>) !down <LAST-BLOCK> !up<ERROR>}

        # value and anomaly to RRQ op. code, filename and mode
        replace <RRQ>.0 {<_OLD> |<A-16>}
        replace <FILE-NAME> {"sample.txt" 0x00 |<A-string>}
        replace <MODE> {<MODE>.0 |<A-string>}

        # value and anomaly to error op. code, error code and message
```

```
replace <ERROR>.0 {<_OLD> |<A-16>}
replace <ERROR-CODE> {(0x00 0x00) |<A-16>}
replace <ERROR-MESSAGE> {"test error" 0x00 |<A-string>}

# anomaly to acknowledgement block number
replace <ACK>.1 {<_OLD> |<A-16>}

# remove unused productions
cutNames

# define anomalies
type <A-16> {
    (0x00 0x00) |(0x00 0x01) |(0x00 0x02) |(0x00 0x03) |
    (0x00 0x04) |(0x00 0x05) |(0x00 0x06) |(0x00 0xff) |
    (0x7f 0xff) |(0x80 0x00) |(0x80 0x01) |(0xff 0xfe) |
    (0xff 0x7f) |(0x00 0x80) |(0x01 0x08) |(0xfe 0xff) |
    (0xff 0xff)
}
type <A-string> {
    # string missing and empty string
    () |0x00 |

    # "illegal" characters
    0x01 0x00 |0x10 0x00 |0x1f 0x00 |0x7f 0x00 |0x80 0x00 |
    0x81 0x00 |0xa0 0x00 |0xfe 0x00 |0xff 0x00 |

    # overflow strings
    32x 0x61 0x00 |64x 0x61 0x00 |128x 0x61 0x00 |256x 0x61 0x00 |
    511x 0x61 0x00 |512x 0x61 0x00 |513x 0x61 0x00 |
    1024x 0x61 0x00 |2048x 0x61 0x00 |4096x 0x61 0x00

    # runaway overflow strings (no terminating null)
    32x 0x61 |64x 0x61 |128x 0x61 |256x 0x61 |
    511x 0x61 |512x 0x61 |513x 0x61 |
    1024x 0x61 |2048x 0x61 |4096x 0x61
}
```

```
    # add rules
    section insertRule {
        section rule [new FI.protos.rule.UDPSocket] {
            property timeoutPeriod 5000

            property open root
            property output !up
            property input !down

            property remotePort 69
            property followPort true

            #property remoteHost "10.10.10.205"
            property remoteHost "10.10.10.41"
        }
    }
    section insertRule {
        section rule [new FI.protos.rule.SequenceNumber] {
            property number <BLOCK-NUMBER>
            property step <BLOCK>
            property start 1
            property byteLength 2
        }
    }
    show
}

section selection {
    section combine {
        property label "zero-case"
        section masks {
        }
    }
    section combine {
        property label "error-cases"
        section masks {
            add *.<read-error>.?
    }
}
# RRQ filename, RRQ mode or error message anomaly
section combine {
    property label "string"
    section masks {
        add *.<A-string>.?
    }
}
# PDU operation code, ack number anomaly or error code
section combine {
    property label "integer"
    section masks {
        add *.<A-16>.?
    }
}
# RRQ filename overflow with mode missing
section combine {
    property label "no-mode-and-filename"
    section masks {
        add *.<FILE-NAME>.*.<A-string>.?
        add *.<MODE>.*.<A-string>.0
    }
}
# First PDU operation code anomaly and overflow
section combine {
    property label "operation-code-and-overflow-1"
    section masks {
        add *.<RRQ>.0.*.<A-16>.?
        add *.<RRQ>.2.*.<A-string>.?
    }
}
# Intermediate PDU operation code anomaly and overflow
section combine {
    property label "operation-code-and-overflow-2"
    section masks {
        add *.<read-error>.0.*.<ERROR>.0.*.<A-16>.?
```

```
                add *.<read-error>.0.*.<ERROR>.*.<A-string>.?
            }
        }
        # Last PDU operation code anomaly and overflow
        section combine {
            property label "operation-code-and-overflow-3"
            section masks {
                add *.<read-error>.1.*.<ERROR>.0.*.<A-16>.?
                add *.<read-error>.1.*.<ERROR>.*.<A-string>.?
            }
        }
    }

    section postSelection {
        # clean up the grammar a bit (not necessary)
        cutSelections <RRQ>.*
        cutSelections <ACK>.*
        cutSelections <ERROR>.*
        cutNames

        # show grammar, evaluate and show again
        show
        evaluateRules
        show
    }

    section logger {
        property debug true
    }
    section controller [new FI.protos.driver.BlindControl] {
        property timeout -1
    }
}

# Script Trailer
call [it] run
```

Appendix F: Results from Test Runs

WAP-WSP-Request Test Suite

Test groups	**39**		**Test cases**	**4 236**
Test run	Fail verdicts: test groups	test cases	Note	
001	10	569		
002	18	141	Exploited	
003	2	10	Exploited	
004	16	385	Exploited	
005	8	664		
006	14	622		
007	20	148	Exploited	

Legend:	*Exploited*	One of the found vulnerabilities was exploited by running remotely supplied code in the host system.

WAP-WMLC Test Suite

Test groups	**84**		**Test cases**	**1 033**

Test run	Fail verdicts: test groups	test cases	Note
001	4	26	
002	43	183	
003	2	8	Exploited
004	2	4	Exploited
005	4	34	Embedded
006	9	21	Embedded
007	2	25	Embedded
008	11	31	Embedded
009	1	9	Embedded
010	15	34	Embedded

Legend:	*Exploited*	One of the found vulnerabilities was exploited by running remotely supplied code in the host system.
	Embedded	The tested implementation was embedded into a hardware product.

HTTP-Reply Test Suite

Test groups	**115**	**Test cases**	**3 966**

Test run	Fail verdicts: test groups	test cases	Note
001	8	122	Exploited
002	4	57	
003	8	120	
004	1	2	Exploited
005	1	2	
006	0	0	
007	0	0	Proxy
008	0	0	Proxy
009	0	0	Proxy
010	0	0	Proxy
011	0	0	Proxy
012	1	9	

Legend:	*Exploited*	One of the found vulnerabilities was exploited by running remotely supplied code in the host system.
	Proxy	Tested implementation was HTTP proxy, not a browser.

LDAPv3 Test Suite

Application	Test groups	16	Test cases	5 964
Encoding	Test groups	77	Test cases	6 685
	Test groups	**93**	**Test cases**	**12 649**

Test run	Fail verdicts:				Note
	application test groups	application test cases	encoding test groups	encoding test cases	
001	0	0	1	N	
002	*) 1	N	*) 1	N	Exploited
003	0	0	0	0	
004	23	N	1	5	Exploited
005	46	N	1	N	Exploited
006	*) 4	N	9	79	
007	0	0	0	0	
008	1	9	12	N	Exploited

Legend:	*Exploited*	One of the found vulnerabilities was exploited by running remotely supplied code in the host system.
	*)	The exact number of test groups with failures could not be verified, but a lower bound is given.
	N	Failures were too frequent to execute all test cases.

SNMPv1 Test Suite

Request	Application	Test groups	61	Test cases	10 601
	Encoding	Test groups	57	Test cases	18 915
		Test groups	**118**	**Test cases**	**29 516**
Trap	Application	Test groups	76	Test cases	15 323
	Encoding	Test groups	24	Test cases	8 777
		Test groups	**100**	**Test cases**	**24 100**

Test run	Fail verdicts: application test groups	application test cases	encoding test groups	encoding test cases	Note
001	8	N	0	0	
002	0	0	0	0	
003	6	13	5	N	Net, Exploited
004	8	16	0	0	
005	0	0	3	4	Net
006	8	224	0	0	Net
007	4	57	5	N	Net
008	0	0	2	10	
009	9	N	1	5	Exploited
010	17	166	5	24	
011	8	12	10	N	
012	1	N	3	N	

Legend:	*Exploited*	One of the found vulnerabilities was exploited by running remotely supplied code in the host system.
	Net	The tested product was an embedded device dedicated to network traffic forwarding and screening.
	N	Failures were too frequent to execute all test cases.

Published by		Series title, number and report code of publication
VTT	Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 4561 Fax +358 9 456 4374	VTT Publications 448 VTT–PUBS–448

Author(s)

Kaksonen, Rauli

Title

A Functional Method for Assessing Protocol Implementation Security

Abstract

Serious information security vulnerabilities are discovered daily and reported from already deployed software products. Customers have no feasible means for estimating the security level of the products they purchase. The few generally applicable methods require the source code, which is often not delivered with a product. Many of the reported vulnerabilities are robustness problems. Robustness can be functionally assessed without the source code by injecting anomalies, unexpected input elements, to the tested component. The component passes the tests if it can securely handle the injected anomalies.

The methods generally applied for software testing and modelling were found to be too complex and rigid for functional robustness assessment. A new mini-simulation method using attribute grammar to model both input syntax and software behaviour was proposed. Means for the systematic creation of a large number of test cases was presented. The method was used to test the robustness of 49 software products. A total of 40 tested products were found to be vulnerable to denial-of-service problems, and 14 of them were proven to contain vulnerabilities making it possible to execute remotely supplied code on the host system.

Applications of the method include quantitative comparisons and the benchmarking of software components, but it has some limitations. The proportion of the flaws found using the method compared to the actual number of flaws is difficult to assess and the tests may favour some components over others. However, if the method can help to eliminate the most obvious vulnerabilities, it would be much more difficult to find serious flaws using unsystematic methods. This could cut down on the number of publicly disclosed vulnerabilities.

Keywords

information security, automated testing, software quality, implementation vulnerabilities, programming mistakes, mini-simulation method

Activity unit

VTT Electronics, Telecommunication Systems, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN 951–38–5873–1 (soft back ed.) 951–38–5874–X (URL: http://www.inf.vtt.fi/pdf/)			Project number
Date October 2001	Language English	Pages 128 p. + 15 p.	Price C
Name of project Security Testing of Protocol Implementations (PROTOS)		Commissioned by	
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

Julkaisija

VTT
Vuorimiehentie 5, PL 2000, 02044 VTT
Puh. (09) 4561
Faksi (09) 456 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 448
VTT–PUBS–448

Tekijä(t)

Kaksonen, Rauli

Nimeke

Menetelmä ohjelmistojen tietoturvan toiminnalliseen analysointiin

Tiivistelmä

Ohjelmistoista löytyneitä vakavia tietoturvallisuushaavoittuvaisuuksia löydetään ja raportoidaan päivittäin. Asiakkailla ei ole käyttökelpoisia menetelmiä hankkimiensa tuotteiden tietoturvallisuustason arviointiin. Harvat yleisesti käyttökelpoiset menetelmät vaativat lähdekoodin, eikä sellaista yleensä toimitetakaan tuotteen mukana. Monet raportoiduista haavoittuvaisuuksista ovat ohjelmistojen toimintavarmuus-ongelmia. Toimintavarmuutta on mahdollista arvioida funktionaalisesti ilman lähdekoodia syöttämällä anomalioita, odottamattomia dataelementtejä, testattavaan ohjelmistoon. Ohjelmisto läpäisee testit, jos se pystyy turvallisesti käsittelemään syötetyt anomaliat.

Yleisesti käytetyt ohjelmistojen testaus- ja mallinnusmenetelmät havaittiin monimutkaisiksi ja jäykiksi funktionaaliseen toimintavarmuuden analysointiin. Analysointiin ehdotettiin uutta mini-simulaatio-menetelmää, joka käyttää attribuuttikielioppia sekä syötteiden syntaksin että ohjelmiston käyttäytymisen mallintamiseen. Suurten testitapausmäärien systemaattinen luominen menetelmällä esiteltiin. Menetelmää käytettiin 49:n eri ohjelmiston toimintavarmuuden testaamiseen, näistä 40:stä löydettiin toiminta-varmuusongelmia. 14 ohjelmiston sisältämien haavoittuvaisuuksien osoitettiin mahdollistavat ulkopuolelta syötetyn ohjelmakoodin ajamisen kyseessä olevissa järjestelmissä.

Esitellyn menetelmän sovelluksia ovat ohjelmistokomponenttien kvantitatiiviset vertailut ja mittaukset, mutta sillä on joitain heikkouksia. Paljastuvien virheiden ja komponenttien sisältävien kaikkien virheiden suhdetta on vaikea arvioida ja testit voivat suosia joitakin komponentteja toisten kustannuksella. Kuitenkin, jos menetelmän avulla voitaisiin karsia ilmeisimmät haavoittuvaisuudet, olisi haavoittu-vaisuuksia paljon vaikeampi löytää ei-systemaattisilla menetelmillä. Tämä voisi vähentää julkisesti paljastettujen tietoturvahaavoittuvaisuuksien määrää.

Avainsanat

information security, automated testing, software quality, implementation vulnerabilities, programming mistakes, mini-simulation method

Toimintayksikkö

VTT Elektroniikka, Tietoliikennejärjestelmät, Kaitoväylä 1, PL 1100, 90571 OULU

ISBN 951–38–5873–1 (nid.) 951–38–5874–X (URL: http://www.inf.vtt.fi/pdf/)			Projektinumero
Julkaisuaika Lokakuu 2001	Kieli Englanti	Sivuja 128 s. + liitt. 15 s.	Hinta C
Projektin nimi Security Testing of Protocol Implementations (PROTOS)		Toimeksiantaja(t)	
Avainnimeke ja ISSN VTT Publications 1235–0621 (nid.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)		Myynti VTT Tietopalvelu PL 2000, 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	