On Password Guessing with GPUs and FPGAs

Markus Dürmuth and Thorsten Kranz

Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany

Abstract. Passwords are still by far the most widely used form of user authentication, for applications ranging from online banking or corporate network access to storage encryption. Password guessing thus poses a serious threat for a multitude of applications. Modern password hashes are specifically designed to slow down guessing attacks. However, having exact measures for the rate of password guessing against determined attackers is non-trivial but important for evaluating the security for many systems. Moreover, such information may be valuable for designing new password hashes, such as in the ongoing *password hashing competition* (PHC).
In this work, we investigate two popular password hashes, bcrypt and scrypt, with respect to implementations on non-standard computing platforms. Both functions were specifically designed to only allow slow-rate password derivation and, thus, guessing rates. We develop a methodology for fairly comparing different implementations of password hashes, and apply this methodology to our own implementation of scrypt on GPUs, as well as existing implementations of bcrypt and scrypt on GPUs and FPGAs.

Keywords: password hashing, password cracking, efficient implementations, FPGAs, GPUs, bcrypt, scrypt

1 Introduction

Passwords are still the most widely used form of user authentication on the Internet (and beyond), despite substantial effort to replace them. Thus, research to improve their security is necessary. One potential risk with authentication in general is that authentication data has to be stored on the login server, in a form that enables the login server to test for correctness of the provided credentials. The database of stored credentials is a high-profile target for an attacker, which was illustrated in recent years by a substantial number of databases leaked by attacks. Even worse, for storage encryption the encryption key, protected by the password using a key derivation function (KDF), is stored on the same machine as the encrypted data, and thus an even easier target. A leak of the password database is a major concern not only because the credentials for that particular site leak, and resetting all passwords for all users of a site in a short time span requires a significant effort. In addition, password re-use, i. e., using one password for more than one site, which is a frequent phenomenon to reduce the cognitive load of a user, causes a single leaked password to compromise a larger number of accounts.

In order to mitigate the adverse effects of password leaks, passwords are typically not stored in plain, but in hashed (and possibly salted) form, i. e., one stores

$$(s, h) = (salt, \mathsf{Hash}(pwd, salt))$$

for a randomly chosen value *salt*. Such a hashed password can easily be checked by recomputing the hash and comparing it to the stored value h. While a secure hash function cannot be inverted, i. e., directly computing the password pwd from (s, h) is infeasible in general, the mere fact that the server can verify the password gives rise to a so-called *offline guessing attack*. Here, an attacker produces a large number of password candidates $pwd_1, pwd_2, pwd_3, \ldots$, and verifies each candidate as described before. User-chosen passwords are well-known to be predictable on average [19, 37], so such an attack is likely to reveal a large fraction of the stored passwords, unless special precautions are taken.

A widely used method to defend against offline guessing attacks is using hash functions that are slow to evaluate. While cryptographic hash functions are designed to be fast to compute, password hashes are deliberately slow, often using iterated constructions to slow down an attacker. This, of course, also slows down the legitimate server, but the attacker is typically more substantially affected by the slow-down as he needs to evaluate the hash functions millions or billions times. Some well-known examples for password hashes are the classical descrypt [24], which dates back to the 1970s, md5crypt, sha256crypt/sha512crypt, PBKDF2 [18], bcrypt [31], and scrypt [30]. There is ongoing effort to design stronger password hashes, e. g., the password hashing competition [29].

Currently lacking is a thorough understanding of the resistance of those password hashes against attacks using non-standard computing devices, in particular FPGAs and GPUs. Understanding these issues is, however, crucial to decide which password hash should be used, and at what hardness settings.

In this work, we make several contributions towards this goal: first, we provide an implementation of scrypt on GPUs that supports arbitrary parameters, which is substantially faster than existing implementations; second, we determine "equivalent" parameter sets for password hashes to allow for a fair comparison; third, based on the equivalent parameter sets, existing implementations, and our implementation of scrypt, we draw a fair comparison between bcrypt and scrypt. In summary, we find that for fast parameters both bcrypt and scrypt offer about the same level of security, while for slow parameters scrypt offers more security, at the cost of increased memory consumption.

1.1 Related Work

Password security. Guessing attacks against passwords have a long history [2, 39, 22]. More recently, probabilistic context-free grammars [37] as well as Markov models [25, 5] have been used with great success for password guessing. Most password cracking tools implement some form of mangling rules, some also support some form of Markov models, e. g., *John the Ripper* (JtR) and hashcat. An empirical study on the effectiveness of different attacks including those based on Markov models can be found in [7]. If no salt is used in the password hash, *rainbow-tables* can be used to speed up the guessing step [15, 28] using precomputation. An implementation of rainbow-tables in hardware is studied in [23].

Closely related to the problem of password guessing is that of *estimating the strength of a password.* In early systems, password cracking was used to find weak passwords [24]. Since then, so called pro-active password checkers are used to exclude weak passwords [2, 4]. However, most pro-active password checkers use relatively simple rule-sets to determine password strength, which have been shown to be a rather bad indicator of real-world password strength [36, 20, 6]. More recently, Schechter et al. [32] classified password strength by counting the number of times a certain password is present in the password database, and Markov models have been shown to be a very good predictor of password strength and can be implemented in a secure way [6].

Processing platforms for password cracking. Password cracking is widely used on general-purpose CPUs, and cleverly optimized implementations can achieve substantial speed-up compared to straight-forward implementations. Well-known examples for such "general purpose tools" are *John the Ripper* [17], as well as specialized tools such as TrueCrack [35] for TrueCrypt encrypted volumes. However, due to the versatility of their architecture, CPUs usually do not achieve an optimal *cost-performance ratio* for a *specific* application.

Modern graphics cards (GPUs) have evolved into computation platforms for universal computations. GPUs combine a large number of parallel processor cores which allow highly parallel applications using programming models such as OpenCL or CUDA. GPUs have proven very effective for password cracking, demonstrated by tools such as the Lightning Hash Cracker by ElcomSoft [9] or hashcat [33].

Special-purpose hardware usually provides significant savings in terms of costs and power consumption and at the same time provides a boost in performance time. This makes special-purpose hardware very attractive for cryptanalysis [13, 14, 10, 40]. With the goal of benchmarking a power-efficient password cracking approach, Malvoni et al. [21] provide several implementations of bcrypt on low-power devices, including an FPGA implementation. Similarly, Wiemer et al. [38] provide an FPGA implementation of bcrypt. In [8], the authors provided implementations of PBKDF2 using GPUs and an FPGA cluster, targeting TrueCrypt.

1.2 Outline

We describe the scrypt algorithm and our GPU implementation in Section 2, and briefly review the bcrypt algorithm and recent work on implementing bcrypt on FPGAs in Section 3. In Section 4 we present a framework for comparing password

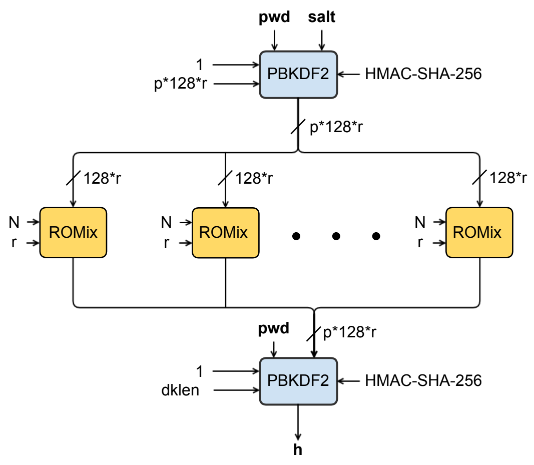

Fig. 1. Overview of scrypt. The data widths are given in bytes.

hashing functions and dedicated attacker platforms. We present the final results and a discussion in Section 5.

2 The scrypt Password Hash

In this section we describe the scrypt password hash and present a GPU implementation of scrypt for guessing passwords in parallel.

2.1 The scrypt Construction

The scrypt password hash [30] is a construction for a password hash which specifically counters attacks using custom hardware (the cost estimations specifically target ASIC designs, but the results hold, in principle, against FPGAs as well). The basic idea of the scrypt design is to force an attacker to use a large amount of memory, which results in large area for the memory cells and thus high cost of the ASICs.

Parameters. The scrypt algorithm takes as input a password *pwd* and a salt *salt*, and is parameterized with the desired output length *dklen* and three cost parameters: memory usage N, a block-size r, and a parallelism factor p. If $p > 1$ then basically p copies of the ROMix algorithm, which is described below, are executed independently of each other; the overall memory usage for ROMix is $128 \cdot r \cdot N$ bytes. The final output is a hash value h of size *dklen* bytes.

Overall structure. The overall structure of scrypt consists of three main steps (see Figure 1).

Algorithm 2.1: ROMix

```
   Data: $B$, $r$, $N$
   Result: $B'$
 1 $X \leftarrow B$;
 2 for $i \leftarrow 0$ to $N-1$ do
 3     $V[i] \leftarrow X$;
 4     $X \leftarrow \text{BlockMix}(X)$;
 5 end
 6 for $i \leftarrow 0$ to $N-1$ do
 7     $j \leftarrow \text{Integerify}(X) \bmod N$;
 8     $X \leftarrow \text{BlockMix}(X \oplus V[j])$;
 9 end
10 $B' \leftarrow X$;
```

(i) Initially, scrypt applies the PBKDF2 password hash [18] to the password and the salt, with an iteration count of 1, using HMAC-SHA-256 as MAC function, and producing $128 \cdot p \cdot r$ bytes of output. PBKDF2 is used to distribute the entropy from the password and salt and expand the input length, and presumably as a fail-safe mechanism to ensure the onewayness of the overall construction.

(ii) The output of this initial step is split into p chunks of $128 \cdot r$ bytes, and each chunk is fed into one of p parallel copies of the ROMix algorithm, which is the core part of the scrypt construction and described below.

(iii) Each invocation of the ROMix algorithm yields $128 \cdot r$ bytes of data, which are concatenated and fed into another instance of PBKDF2, together with the password, an iteration count of 1, and using HMAC-SHA-256, which finally produces the desired output of length *dklen*.

ROMix. The ROMix algorithm is the core of the construction. It operates on blocks of size $128 \cdot r$ bytes, and allocates an array V of N blocks as the main data structure. ROMix first fills the array V with pseudo-random data, and then pseudo-randomly accesses the data in the array to ensure the attacker is actually storing the data.

(i) First, ROMix fills the array V by repeatedly calling BlockMix which is basically a random permutation derived from the Salsa20/8 hash function (see below). The current state X is initialized with the input bytes (derived from the output of PBKDF2). Then, successively, BlockMix is applied to the state and the result written to successive array locations. The pseudo-code is shown in Algorithm 2.1 from line 2 to 5.

(ii) Second, the stored memory is accessed in a pseudo-random fashion in an attempt to ensure that all memory cells are stored. The initial state X is the final state of the previous step. The current state is interpreted as an index pointing to an element in the array V, that target value is XORed to the current state, and H is applied to form the next state. The pseudo-code is shown in Algorithm 2.1 from line 6 to 9

BlockMix. The BlockMix construction operates on $2 \cdot r$ blocks of size 64 bytes each. It resembles the CBC mode of operation, with a final permutation of the block order. Its main use is apparently to widen the block size from the fixed 64 bytes of Salsa20/8 to arbitrary width as required by the ROMix algorithm.

Recommended parameter values. Two sets of parameter choices are given [30] for typical use cases. For storage encryption on a local machine Percival proposes $N = 2^{20}, r = 8, p = 1$, which uses 1024 MB of memory. For remote server login he proposes $N = 2^{14}, r = 8, p = 1$, which uses 16 MB. Android since version 4.4 uses scrypt for storage encryption [11], with parameters $(N, r, p) = (2^{15}, 3, 1)$ [34].

2.2 GPU Programming

Over the years, GPUs have changed from mere graphic processors to general purpose processing units, offering programming interfaces such as CUDA [27] for cards manufactured by NVIDIA.

GPUs execute code in so called kernels, which are functions that are executed by many threads in parallel. Each thread is member of a block of threads. All threads within a block have access to the same shared memory, which allows communication and synchronization between threads. During execution, blocks are assigned to Streaming Multiprocessors (SMs). An SM then schedules its pending blocks in chunks of 32 threads, called a warp, to its hardware, where each thread within a warp executes the same instruction. When threads in the same warp execute different instructions they are scheduled one after another (thread divergence). When threads are scheduled for high-latency memory instructions, the scheduler will execute additional warps while waiting for the memory access to finish, thus to a certain extent hiding the slow memory access.

Each thread has private *registers* and *local memory* which is, for example, used for *register spilling*. Threads from the same block can access the fast per-block *shared memory*, which can be used for inter-thread communication. All threads can access *global memory*, which is by far the largest memory, but also the slowest. There are some specialized memory regions, *constant memory* and *texture memory*, which are fast for specific access patterns.

NVIDIA's GTX 480 GPU [26] is a consumer-grade GPU which offers reasonable performance at an affordable price. It entered market in 2010 at the price of 499 dollars. A GTX 480 consists of 15 SMs with 32 computing cores each, i. e., the architecture provides 480 cores within a single GPU. Memory bandwidth is 177.4 GB/s. The cores are running at 1401 MHz and can reach a single-precision floating point performance (Peak) of up to 1345 GFLOPS. (For comparison: Intel's recent Core i7 980 CPUs running at 3.6 GHz are listed at 86 GFLOPS [16].) The GTX 480 offers 1536 MB of global memory.

2.3 Implementing scrypt on CUDA

Our implementation performs a brute-force password search over a configurable character set. The implementation is fully on the GPU, the CPU is only responsible for enumerating the passwords, calling the GPU kernels, and comparing the final results.(Parts of the implementation are inspired by the cudaMiner [3], a miner for the litecoin cryptocurrency, which uses scrypt with very low cost parameters $(N, r, p) = (1024, 1, 1)$ as proof-of-work.)

The CPU keeps track of the current progress and calls a new kernel with a starting point in the space of all passwords. It starts as many threads in parallel as are allowed by the available global memory, but always requires the number of threads to be a multiple of 32, as we are running 32 threads per warp. If the parameter p is greater than one, then those blocks will be executed one after another, which does not increase memory usage. In the remainder of the section we give some details about the GPU implementation.

PBKDF2. The implementation of PBKDF2 is rather straightforward. The iteration count of $c = 1$ is hard-coded. Overall, the operation is not time-critical.

BlockMix. The BlockMix operation operates on a state of $2 \cdot r$ words of size 64 bytes each, thus $128 \cdot r$ bytes in total, which are kept in the registers. For an efficient implementation of the mixing layer, in addition to the array holding the data, we implement an array with pointers that serve as index for the data; this way the mixing layer can be implemented by copying pointers (4 bytes) instead of blocks of data (64 bytes). The Salsa20/8 implementation follows the original proposal [1] including the optimization to eliminate the transpositions by alternatingly processing rows and columns.

ROMix. The implementation of ROMix has to take special care of the memory hierarchy in order to utilize the GPUs potential. The main concern is maximizing *memory throughput.* Global memory can be accessed in chunks of 32, 64, or 128 bytes, which must be aligned to a multiple of their size (naturally aligned). However, one thread can access a word of at most 16 bytes, so memory throughput is maximized when several threads access contiguous and aligned words; then memory access is called *coalesced.* Therefore, reading one block (64 bytes) is distributed across four threads reading words of 16 bytes, and, as each of the four threads needs to access a full block after all, they will cooperate four times to load all four blocks. Data is first read to shared memory by the cooperating threads, then copied to the registers by each thread individually.

Writing data to global memory follows the same rules. The data is first copied by the individual threads from registers to shared memory and then written to global memory by cooperating threads in an aligned and coalesced fashion.

Time-memory trade-off. Our implementation also provides the possibility to use a time-memory trade-off. By just storing every t-th data segment generated by the initial BlockMix iterations, only $1/t$ of the original amount of memory is needed. In return, every time a segment that was not stored is needed, it must be recomputed from the nearest previous segment. If t is increased, the probability of such a recomputation rises. So does the time needed for a recomputation since there are on average more iterations to recompute.

Algorithm 3.1: bcrypt

```
  Input: cost, salt, pwd
  Output: hash
1 $state \leftarrow$ EksBlowfishSetup($cost, salt, pwd$);
2 $ctext \leftarrow$ “OrpheanBeholderScryDoubt”;
3 Repeat (64) begin
4     $ctext \leftarrow$ EncryptECB($state, ctext$);
5 end
6 return $Concatenate(cost, salt, ctext)$;
```

Algorithm 3.2: EksBlowfishSetup

```
  Input: cost, salt, pwd
  Output: state
1 $state \leftarrow$ InitState();
2 $state \leftarrow$ ExpandKey($state, salt, pwd$);
3 Repeat ($2^{cost}$) begin
4     $state \leftarrow$ ExpandKey($state, 0, salt$);
5     $state \leftarrow$ ExpandKey($state, 0, pwd$);
6 end
7 return $state$;
```

3 The bcrypt Password Hash

The second password hash we consider is the bcrypt hash function.

3.1 The bcrypt Construction

Provos and Mazières published the bcrypt hash function [31] in 1999, which, at its core, is a cost-parameterized, modified version of the blowfish algorithm. The key concepts are a tunable cost parameter and a constantly modified moderately large (4 KB) block of memory. The bcrypt password hash is used as the default password hash in OpenBSD since version 2.1 [31]. Additionally, it is the default password hash in current versions of Ruby on Rails and PHP.

Parameters. The bcrypt algorithm uses the input parameters *cost*, *salt*, and *key*. The number of executed loop iterations is exponential in the *cost* parameter, cf. Algorithm 3.2. The algorithm uses a 128-bit *salt* to derive a 192-bit password hash from a *key* of up to 56 bytes.

Design. The algorithm is structured in two phases. First, `EksBlowfishSetup` initializes the internal state. Afterwards, Algorithm 3.1 repeatedly encrypts a magic value using this state. The resulting ciphertext is then concatenated with the cost and salt and returned as the hash. While the encryption itself is as efficient as the original Blowfish encryption, most of the time is spent in the `EksBlowfishSetup` algorithm.

The `EncryptECB` encryption is effectively a blowfish encryption. Within its standard 16-round Feistel network, the S-boxes and subkeys are determined by the current *state* and the plaintext is encrypted in 64-bit blocks.

The `EksBlowfishSetup` algorithm is a modified version of the blowfish key schedule. It computes a state, which consists of 18 32-bit subkeys and four S-boxes – each 256×32-bit in size – which are later used in the encryption process. The state is initially filled with the digits of π and a modified version of the blowfish keyschedule is performed. After xoring the key to the subkeys, it successively uses the current state as S-boxes and subkeys to encrypt blocks of the current state and update the state. In this process, the function `ExpandKey` computes 521 blowfish encryptions. If the salt is fixed to zero, one call to `ExpandKey` resembles the standard blowfish key schedule.

Recommended parameter values. Provos and Mazières originally proposed to use a cost parameter of six for normal user passwords, while using eight for administrator passwords.

3.2 Implementations of bcrypt on FPGAs

While general-purpose hardware, i. e., CPUs, offers a wide variety of instructions for all kinds of programs and algorithms, usually, only a few are important for a specific task. More importantly, the generic structure and design might impose restrictions and become cumbersome, i. e., when registers are too small or memory access times becomes a bottleneck. Reconfigurable hardware like Field-Programmable Gate Arrays (FPGAs) and special-purpose hardware like Application-Specific Integrated Circuits (ASICs) are more specialized and dedicated to a single task. An FPGA consists of a large area of programmable logic resources (the fabric), e. g., lookup tables, shift registers, multiplexers and storage elements, and a fixed amount of dedicated hardware modules, e. g., memory cores (BRAM), digital signal processing units, or even PowerPCs, and can be specialized for a given task.

Recently, two groups presented implementations of bcrypt on FPGAs. The latest work is by Wiemer et al. [38], who present an implementation of bcrypt on Xilinx FPGAs from the low-power consumption and low cost segment. Their platform is the zedboard, more precisely the Zynq-7000 XC7Z020 FPGA. The Zynq-7000 persists mainly of a dual-core ARM Cortex A9 CPU and an Artix-7. The zedboard allows easy access to the logic inside the FPGA fabric via direct memory access and provides several interfaces, e. g., AXI4, AXI4-Stream, AXI4-Lite or Xillybus. These cores come with drivers for embedded Linux kernels and thus offer an easy way of accessing custom logic from a higher abstraction layer. Their design has a LUT consumption of $2,777$ LUTs per (quad-)core and uses 13 BRAMs. Including a simple logic for generating password candidates for a brute-force guessing attack, they were able to fit 10 quad-core designs on a single FPGA, which runs at a maximum clock frequency of 100 MHz. They reported $6,511$ hashes per second for a cost parameter of 5.

The other work by Malvoni et al. [21] reported a rate of 4571 passwords per second for a cost parameter of 5 on the zedboard. Due to unstable behavior, they could not fully implement there design idea of 56 bcrypt instance and had to reduce this number to 28. Therefore, the simulated their design on the larger Zynq-7045 and reported 7044 passwords per second as the expected result for a stable behavior. Additionally, they reported a theoretical rate of 8112 passwords per second which they derived from the performance for a cost parameter of 12.

4 Methodology

Next, we present the methodology that we use to compare different algorithms on different platforms.

4.1 Basic Idea

In this work we consider offline guessing attacks, and consequently the hashrate, i. e., how fast an attacker can verify its password guesses, is the critical factor. The effect of a password hash is to slow down the attacker's verification of a password guess. Slower password hashes will usually slow down both the (honest) verification of a password, as well as the attacker. However, an attacker is not bound to use the same implementation as the (honest) verification server, and so he may utilize optimizations that the legitimate verifier is not able to implement; in particular the adversary can use different hardware platforms much more easily than the verification server.

Thus, it is important to consider the ratio between the following two runtimes: first, the runtime of the normal (optimized for server use) implementation on typical server CPUs, and second, the runtime for a password on an attacker's implementation on comparable hardware of his choice. Here, the defender chooses the algorithm and parameters to be used, while the attacker can choose a hardware platform and has certain optimization techniques that the defender cannot use. As we want to compare different password hashing algorithms attacked on different platforms, we need to derive *reasonably equivalent parameters* for the different password hashes. Thus, we start by measuring the runtime of the algorithms on different PCs – which differ in the amount of processors as well as architecture and available memory – and derive comparable algorithm-parameter pairs.

4.2 Derivation of Equivalent Parameters

To determine the "equivalent" parameter sets for the different schemes, we run a series of tests on different CPUs and compare runtimes. We use implementations that target password checking by legitimate servers (i. e., that check one password at a time). Thus, we call two parameter sets of two algorithms "equivalent" if the legitimate server that checks the passwords needs the same runtime to do so in both cases.

We used the following implementations: for *PBKDF2*, we used the implementation in the OpenSSL library calling `PKCS5_PBKDF2_HMAC()` with `SHA512`. For *bcrypt*, we used a version available from the Openwall website (`http://www.openwall.com/crypt/`), which was compiled on the target system gcc and compiler flags `-O3 -fomit-frame-pointer -funroll-loops`. For *scrypt*, we use our own implementation in C, as the original implementation is packaged into a larger project. The runtimes were comparable to those published by Percival [30].

Table 8 in Appendix B lists the platforms we used for the parameter derivation. We utilized different CPUs, with an emphasis on server CPUs, and measured runtimes for each of them. Appendix B gives the full measurement results. As there is no single system we can optimize for but are interested in general statements, we take the average runtime over all CPUs we tested. Note that the runtimes were, despite the wide variance of CPUs, grouped together relatively closely, the worst-case being a factor of two between the fastest and the slowest CPU, and in general much lower.

To investigate reasonable parameters and their resulting runtimes, one must ask for the actual size of parameters used in real-world applications. First, we need to note the this strongly depends on the application scenario. In an interactive login scenario the server must be able to quickly response to the user who tries to authenticate with a password. The situation is different if we consider key derivation for storage encryption, where longer delays are acceptable. (But note that the delay time is not the only bound for a practical implementation. Also extensive memory usage may hinder a server from choosing according parameters.) In light of these differences in the security requirements for password hashing, we make a comparison across a wide range of parameters and desired runtimes.

We give four classes of parameters, for targeted runtimes of (approximately) 1ms, 10ms, 100ms, 1000ms. Percival [30] states 100 ms as an upper bound on the delay for interactive login. For storage encryption, the acceptable runtime is higher and may extend slightly higher than 1000ms. But note that the parameters used for scrypt in Android since version 4.4 [11] for storage encryption (namely $(N, r, p) = (2^{15}, 3, 1)$ [34]) yields moderate running times (around 100ms on server CPUs, but higher on typical mobile devices).

Both bcrypt and scrypt offer a relatively coarse control over the runtime (incrementing the hardness parameter by one approximately doubles the execution time), thus no parameter will match exactly the target time. Therefore, we interpolate the parameters from the measured values, to more accurately model the desired runtimes and making the comparison fair. This means we have to interpolate the runtimes for the attacking implementations in the same way.

The (interpolated) equivalent parameters are listed in Table 1, the detailed measurements are listen in Table 9, 10, and 11 in Appendix B.

Algorithm	(Parameter)	Target (CPU) runtime			
		1ms	10ms	100ms	1000ms
PBKDF2	(Iteration count)	313	3 138	31 347	313 925
bcrypt	(Cost)	3.69	7.03	10.3	13.6
scrypt	(log N (r=8, p=1))	6.93	10.3	13.7	16.9

Table 1. "Equivalent" parameters for several target runtimes for PBKDF2, bcrypt, and scrypt.

.

4.3 Comparing Different Platforms

For comparing the ratio between the runtime of the legitimate server and the attacker, we also need a method to compare attacks using different hardware platforms.

An attack scales linearly with invested resources, mainly cost of the equipment and energy consumption, and thus we have to take their influence into account. (In addition, one can consider development cost, but here we will assume that implementations are already available. While GPU programming is quite similar to CPU programming and thus generally quicker, code development for FPGAs is substantially different and usually requires more time, and thus cost.) This leads to a time-cost trade-off, which is affected by the amount of devices the attacker uses in parallel to increase the hashrate, the costs per device and the power costs.

Generally speaking, equipment cost is in favor of the graphic cards, as GPUs are a consumer product that is sold in large quantities. Also, older versions usually receive a discount, making them more cost-effective. Interestingly, FPGA vendors use a different strategy: with the release of a new product line, the price of the old family stays roughly unchanged, while the new version is offered with a small discount to make the consumers switch away from the abandoned hardware platform. In terms of power consumption, reconfigurable hardware is by far more effective than GPUs. We will consider equipment cost and energy consumption for the different devices when comparing those implementations.

5 Results

Finally, we present and compare the hash rates of different implementations.

5.1 Comparing with oclHashcat

Before giving a more detailed comparison of different platforms, we start with an evaluation of our implementation of scrypt (given in Section 2.3) with existing implementations of scrypt, in particular with the oclHashcat [33] implementation. The scrypt algorithm is supported by oclHashcat starting with version 1.30, released in August 2014. We used the latest version at the time of writing,

	$(2^{12}, 8, 1)$	$(2^{12}, 4, 1)$	$(2^{12}, 1, 1)$	$(2^{17}, 8, 1)$	$(2^{17}, 4, 1)$	$(2^{17}, 1, 1)$
oclHashcat v1.31	19.86	72.58	280.38	0.10	0.26	1.31
Our implementation	287.61	1171.82	27748.27	0.38	2.15	83.08

Table 2. Selected scrypt hashrates from oclHashcat and our GPU implementation.

oclHashcat v1.31. To the best of our knowledge, oclHashcat is the only implementation of scrypt on GPUs allowing for a full variable choice of parameters. The litecoin miner cudaMiner [3], as well as other mining software we are aware of, only implement fixed parameter values (in particular $(N, r, p) = (1024, 1, 1)$ which are the standard litecoin parameters), or partially fixed parameter values (in particular $(N, r, p) = (N, 1, 1)$ which are the parameters for some other scrypt-based altcoins). These constraints allow for deeper optimization techniques.

Both our implementation as well as oclHashcat run on a single NVIDIA Geforce GTX 480 card. Selected hashrates are listed in Table 2, more comprehensive measurements can be found in Appendix A. We can see that our implementation outperforms oclHashcat by a factor of 10 to 100 for moderate values of $N < 2^{14}$, which drops to approximately 4 for higher $N \approx 2^{17}$ and $r = 8$. We cannot investigate why oclHashcat is slower, as it is closed source software.

5.2 Measuring Hashrates

We use the following platforms in the comparison:

– Our GPU-based scrypt implementation is run on an NVIDIA Geforce GTX 480. Prices for GPUs have a substantial variability over time, being influenced by competing products, customer demand and releases of newer cards. The GTX 480 was released at a price of \$499 in the first quarter of 2010 and dropped to around \$310 in the third quarter of 2011. For the subsequent discussion, we estimate a reasonable price at \$350. We assume an attacker mounts three graphics cards on a single motherboard, a setup which was empirically found to offer a good balance [12]. A suitable motherboard is estimated at \$200, and a suitable 1600W power supply we estimate at \$300. Overall, a machine with 3 GPUs will cost approx. \$1550, and the average price per GPU (including peripherals) is approx. \$517. The GTX 480 has been benchmarked with up to 430W under full load, even though the TDP is given as 250W only.
– The same setup was used to measure the speed of oclHashcat's implementation of bcrypt on GPUs.
– The bcrypt implementation from [38] runs on the zedboard, cf. Section 3. The zedboard is currently available for approx. \$319, and has a power consumption of 4.2W.

Table 3 shows the results of the implementations for the derived parameter sets. We can see a couple of interesting points already from this basic data.

	Target (CPU) runtime			
	1ms	10ms	100ms	1000ms
bcrypt				
– zedboard	9 230 H/s	916.25 H/s	98.77 H/s	9.93 H/s
– GTX 480	2 868 H/s	319.37 H/s	33.73 H/s	2.71 H/s
scrypt				
– GTX 480	42 650 H/s	2 333 H/s	49.06 H/s	0.37 H/s
	(t=1)	(t=2)	(t=8)	(t=4)

Table 3. Hashrates of attacking implementations.

		Target (CPU) runtime			
	HW cost	1ms	10ms	100ms	1000ms
bcrypt					
– zedboard	$319	28.93 H/$s	2.87 H/$s	0.31 H/$s	0.03 H/$s
– GTX 480	$517	5.55 H/$s	0.62 H/$s	0.07 H/$s	0.01 H/$s
scrypt					
– GTX 480	$517	82.50 H/$s	4.51 H/$s	0.09 H/$s	0.00 H/$s
		(t=1)	(t=2)	(t=8)	(t=4)

Table 4. Hashes per dollar-second for attacking implementations.

		Target (CPU) runtime			
	Energy	1ms	10ms	100ms	1000ms
bcrypt					
– zedboard	4.2 W	2 198 H/Ws	218.15 H/Ws	23.52 H/Ws	2.36 H/Ws
– GTX 480	430 W	6.67 H/Ws	0.74 H/Ws	0.08 H/Ws	0.01 H/Ws
scrypt					
– GTX 480	430 W	99.19 H/Ws	5.43 H/Ws	0.11 H/Ws	0.00 H/Ws
		(t=1)	(t=2)	(t=8)	(t=4)

Table 5. Hashes per watt-second for attacking implementations.

	Cost		Target (CPU) runtime			
	HW	Energy	1ms	10ms	100ms	1000ms
bcrypt						
– zedboard	$319	$7.41	28.3 H/$s	2.81 H/$s	0.303 H/$s	0.0304 H/$s
– GTX 480	$517	$759	2.25 H/$s	0.250 H/$s	0.0264 H/$s	0.00212 H/$s
scrypt						
– GTX 480	$517	$759	33.4 H/$s	1.83 H/$s	0.0384 H/$s	0.000287 H/$s
			(t=1)	(t=2)	(t=8)	(t=4)

Table 6. Hashes per dollar-second taking into account total cost for two years.

First, we see that bcrypt is faster on the zedboard than it is on a GTX 480, despite the latter being more expensive and more energy consuming. This is a common observation, that specialized hardware implementations are faster and more efficient. Second, our scrypt implementation scales reasonably well between the parameter classes for 1ms and 100ms, but then the pressure from memory consumption becomes too large and speed drops substantially. Also, as expected, with increasing runtime (and thus memory consumption) higher trade-off parameters become optimal, as they trade memory consumption for computational load (except for the highest class of 1000ms). Third, even though the hashrates for different platforms are not directly comparable, we can already see that, while the hashrates for bcrypt scale almost linearly with the CPU runtime, scrypt scales much worse, which is caused by the increased memory consumption.

5.3 Comparison Taking Cost into Account

The comparison in the previous section has the advantage that, by using equivalent parameters for the different password hashes, we obtain a fair comparison between the different password hashes. However, results from different hardware platforms, most notably GPUs and FPGAs, are still hardly comparable, as both have fundamentally different characteristics.

There are two main parameters that are different for the two different platforms: first, hardware cost is different. On the one hand, most GPUs are consumer products and are sold in huge quantities, while FPGA boards that are easily usable by consumers are a niche product. On the other hand, GPUs are equipped with additional units that are irrelevant for our specific application, and FPGAs can fully be configured to the task at hand. Second, the energy cost is fundamentally different. FPGA designs only implement the logic required to compute the desired functionality, which means that most overhead can be avoided. This results in a decreased number of switching logic gates and thus reduced power consumption. There are other factors one could consider, e.g, development cost, but in this discussion we will concentrate on energy cost and hardware cost.

Table 4 shows the results taking into account the hardware cost of the individual devices, as estimated in the previous section. Data is given in hashes per second and dollar (H/$s). What we observe is that the influence of the price is smaller than we expected, as the prices for a GPU including (shared) host PC and an FPGA that sits on a development board are quite similar. Note here that the devices used are one example only, and by using other GPUs or FPGAs the prices are somewhat variable. Also, the price of a development board such as the zedboard is substantially higher than a single FPGA only, but note that an FPGA always will need some additional hardware to facilitate its operation, e. g., to ensure network connectivity. While all these prices come with some uncertainty, the overall picture of the comparison should be quite reliable.

Table 5 shows the results taking into account the energy costs of the different architectures. We listed the approximate power consumption of the GTX 480 and the zedboard as 430 Watt and 4.2 Watt, respectively, which already illustrates the fundamental difference between the two. Note that, again, these estimates are somewhat variable and depend on the specific FPGA and GPU used, as well as the exact load put on the device. What we see is that the zedboard is clearly superior to the GTX 480 in this metric due to the significantly lower power consumption.

Finally, we aim to bring these different results together and determine a total cost, combining the energy and hardware cost for a duration of two years. We estimate energy cost at a price of 10.08 cents per kWh (average retail price of electricity in the United States in 2013, according to the U.S. Energy Information Administration[1]). The results are shown in Table 6. Basically what this table shows is that scrypt can be attacked rather efficiently for low parameters with the GTX 480. The attack is even stronger than the bcrypt attack with the zedboard. But for higher parameters the FPGA attack on bcrypt is more efficient than the GPU attack on scrypt.

Finally, we can say that we were surprised by the fast operation of scrypt on GPUs for moderate parameters. In scenarios where FPGA-based crackers are unavailable (e. g., for the casual password cracker), or for applications where power cost is not a critical factor, bcrypt is more resistant to password cracking for parameters up to the 100ms class. When we additionally consider FPGA-based attacks, the picture changes, as bcrypt can be computed quite efficiently on FPGAs, in particular in terms of energy cost. Except for low parameters from the 1ms class (where GPUs against scrypt are more efficient than FPGAs against bcrypt in terms of hardware cost as well as total cost for two years), scrypt is harder to attack, based on the implementations we are considering. This advantage is almost exclusively caused by the energy cost (energy cost for a single GTX 480 is approximately a factor 100 higher than for a single zedboard).

6 Conclusion

In this work we have provided a methodology for comparing different password hashes on varying platforms. We have applied this methodology to bcrypt and scrypt implementations on GPUs and FPGAs, including our own implementation of scrypt on GPUs, which may be of independent interest. Taking into account all the attacking implementations we have considered, bcrypt and scrypt offer comparable strength for smaller parameters (that take about 1ms to 10ms on the defenders side), while scrypt is stronger for larger parameters.

Acknowledgements

This work was supported in part by the BMBF Project UNIKOPS (01BY1040) and the DFG Research Training Group GRK 1817/1.

[1] http://www.eia.gov/electricity/data/browser

References

1. D. J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
2. M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.
3. Bitcoin Forum. cudaMiner – a new litecoin mining application. `http://bitcointalk.org/index.php?topic=167229.0`.
4. W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic Authentication Guideline: NIST Special Publication 800-63, 2006.
5. C. Castelluccia, A. Chaabane, M. Dürmuth, and D. Perito. Omen: An improved password cracker leveraging personal information. Available as arXiv:1304.6584, 2013.
6. C. Castelluccia, M. Dürmuth, and D. Perito. Adaptive Password-Strength Meters from Markov Models. In *Proc. Network and Distributed Systems Security Symposium (NDSS).* The Internet Society, 2012.
7. M. Dell'Amico, M. Pietro, and Y. Roudier. Password Strength: An Empirical Analysis. In *INFOCOM '10: Proceedings of 29th Conference on Computer Communications.* IEEE, 2010.
8. M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalçin, and R. Zimmermann. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In *Proc. European Symposiumon Research in Computer Security (ESORICS)*, volume 7459 of *Lecture Notes in Computer Science*, pages 716–733. Springer, 2012.
9. ElcomSoft. Lightning Hash Cracker, Nov 2011. `http://www.elcomsoft.com/lhc.html`.
10. T. Gendrullis, M. Novotný, and A. Rupp. A Real-World Attack Breaking A5/1 within Hours. *IACR Cryptology ePrint Archive*, 2008:147, 2008.
11. Google Inc. – Open Handset Alliance. Android KitKat. `https://developer.android.com/about/versions/kitkat.html`.
12. J. Gosney. Password cracking hpc. Presentation at Passwords12 Conference. Online at `http://passwords12.at.ifi.uio.no/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf`, 2012.
13. T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.
14. T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler. Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis. In *Proceedings of the Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 675–678, 2008.
15. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
16. Intel. Intel Core i7-900 Desktop Processor Series, 2011. `http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf`.
17. John the Ripper. `http://www.openwall.com/john/`.
18. B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000. `http://tools.ietf.org/html/rfc2898`.
19. D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proc. USENIX UNIX Security Workshop*, 1990.

20. S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman. Of Passwords and People: Measuring the Effect of Password-Composition Policies. In *CHI 2011: Conference on Human Factors in Computing Systems*, 2011.
21. K. Malvoni, S. Designer, and J. Knezovic. Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware. In *Proc. 8th USENIX Conference on Offensive Technologies*, WOOT'14. USENIX Association, 2014.
22. S. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008.
23. N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. In *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 323–334. Springer Berlin / Heidelberg, 2006.
24. R. Morris and K. Thompson. Password Security: A Case History. *Commun. ACM*, 22(11):594–597, Nov. 1979.
25. A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.
26. Nvidia. NVIDIA GeForce GTX 400 GPU Datasheet, 2010. `http://www.nvidia.com/docs/IO/90025/GTX_480_470_Web_Datasheet_Final.pdf`.
27. Nvidia. CUDA Developer Zone (Website), 2011. `http://developer.nvidia.com/category/zone/cuda-zone`.
28. P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proc. of Advances in Cryptology (CRYPTO 2003)*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.
29. Password hashing competition. `https://password-hashing.net/`.
30. C. Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. Presentation at BSDCan'09. Available online at `http://www.tarsnap.com/scrypt/scrypt.pdf`, 2009.
31. N. Provos and D. Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
32. S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.
33. J. Steube. oclhashcat, 2014. `http://hashcat.net/oclhashcat/`.
34. P. Teufl, A. G. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer. Android encryption systems. In *International Conference on Privacy & Security in Mobile Systems*, 2014. in press.
35. TrueCrack. Online at `http://code.google.com/p/truecrack/`.
36. M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS 2010)*, pages 162–175. ACM, 2010.
37. M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password Cracking Using Probabilistic Context-Free Grammars. In *IEEE Symposium on Security and Privacy*, pages 391–405. IEEE Computer Society, 2009.
38. F. Wiemer and R. Zimmermann. High-speed implementation of bcrypt password search using special-purpose hardware. In *Proc. International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2014.

39. T. Wu. A real-world analysis of kerberos password security. In *Network and Distributed System Security Symposium*, 1999.

40. R. Zimmermann, T. Güneysu, and C. Paar. High-Performance Integer Factoring with Reconfigurable Devices. In *FPL*, pages 83–88. IEEE, 2010.

A Full Runtime Listings for hashcat

	$(2^{11},8,1)$	$(2^{12},8,1)$	$(2^{13},8,1)$	$(2^{14},8,1)$	$(2^{15},8,1)$	$(2^{16},8,1)$	$(2^{17},8,1)$
oclHashcat	71.80	19.86	5.34	1.35	0.62	0.26	0.10
Our results	909.75	287.61	85.12	26.30	4.92	0.93	0.38
	$(2^{11},4,1)$	$(2^{12},4,1)$	$(2^{13},4,1)$	$(2^{14},4,1)$	$(2^{15},4,1)$	$(2^{16},4,1)$	$(2^{17},4,1)$
oclHashcat	146.29	72.58	20.41	5.05	1.33	0.62	0.26
Our results	3253.26	1171.82	365.22	108.52	31.86	7.94	2.15
	$(2^{11},2,1)$	$(2^{12},2,1)$	$(2^{13},2,1)$	$(2^{14},2,1)$	$(2^{15},2,1)$	$(2^{16},2,1)$	$(2^{17},2,1)$
oclHashcat	285.87	143.72	71.34	20.32	5.26	1.35	0.61
Our results	17887.84	5311.56	1967.22	660.43	191.21	54.62	15.64
	$(2^{11},1,1)$	$(2^{12},1,1)$	$(2^{13},1,1)$	$(2^{14},1,1)$	$(2^{15},1,1)$	$(2^{16},1,1)$	$(2^{17},1,1)$
oclHashcat	567.47	280.38	140.80	70.47	19.53	5.25	1.31
Our results	55694.39	27748.27	9179.61	3380.60	1075.55	313.39	83.08

Table 7. Comparison of hashrates for our implementation and oclHashcat v1.31.

B Full Runtime Listings for the Benchmark CPUs

	CPU	CPU Launch	OS	Type
CPU1	Intel Core i5-2400, 3.1 GHz	Q1'11	Win7/CygWin	Desktop
CPU2	AMD Opteron 6276, 2.3 GHz	Q1'13	CentOS 6.2	Cluster
CPU3	Intel Core i5-2520M CPU, 2.50 GHz	Q1'11	Win/CygWin	Laptop
CPU4	Intel Xeon E5540, 2.53 GHz	Q1'09	Ubuntu 12.04	Server
CPU5	Intel Xeon E3-1220 V2, 3.10 GHz	Q2'11	Fedora 19	Server

Table 8. Hardware used to measure CPU runtimes.

Iterations	250	500	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k	1024k
CPU1	0.63	1.27	2.53	5.05	10.06	20.22	40.44	80.92	162.4	323.1	646.2	1288	2604
CPU2	1.03	2.05	4.11	8.22	16.51	33.00	66.16	131.7	262.8	524.5	1051	2119	4217
CPU3	1.04	2.09	4.18	8.36	16.62	33.38	66.74	133.5	266.9	532.7	1063	2136	4264
CPU4	0.79	1.57	3.15	6.29	12.59	25.19	50.38	100.8	201.4	402.9	805.3	1612	3222
CPU5	0.50	0.99	1.98	3.97	7.93	15.87	31.72	63.47	127.0	254.0	507.8	1015	2035
Average	0.80	1.59	3.19	6.38	12.74	25.53	51.09	102.1	204.1	407.5	814.6	1634	3268

Table 9. Running times of **PBKDF2** with HMAC and SHA-512 on CPUs (in *ms*).

N	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
CPU1	1.13	2.29	4.53	8.95	17.81	35.49	71.57	143.5	287.0	578.6	1159	2324	4664
CPU2	1.02	1.90	3.62	7.09	14.07	28.24	56.75	115.5	237.3	474.3	959.3	1938	3911
CPU3	1.19	2.42	4.74	9.45	18.88	37.98	76.00	152.9	307.3	616.8	1236	2479	4958
CPU4	1.16	2.21	4.32	8.53	16.97	33.91	68.34	137.1	287.4	577.4	1157	2317	4643
CPU5	0.72	1.38	2.70	5.33	10.59	21.12	42.56	86.00	173.5	346.9	694.5	1389	2780
Average	1.04	2.04	3.98	7.87	15.66	31.35	63.04	127.0	258.5	518.8	1041	2089	4191

Table 10. Running times of **scrypt** (r=8, p=1) on CPUs (in *ms*).

Cost	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU1	1.28	2.40	4.63	9.12	18.04	35.93	72.15	144.0	288.2	576.6	1148	2307	4613
CPU2	1.67	3.11	6.00	11.79	23.38	46.63	92.81	185.3	370.4	740.5	1480	2960	5918
CPU3	1.35	2.52	4.87	9.55	18.92	37.76	75.20	150.4	301.2	600.4	1204	2410	4842
CPU4	1.48	2.76	5.34	10.47	20.75	41.32	82.44	164.8	329.1	658.3	1317	2631	5264
CPU5	1.04	1.95	3.77	7.41	14.68	29.23	58.34	116.5	232.9	465.5	931.8	1863	3728
Average	1.36	2.55	4.92	9.67	19.15	38.17	76.19	152.2	304.4	608.3	1216	2434	4873

Table 11. Running times of **bcrypt** on CPUs (in *ms*).

C Full Runtime Listings for Different Trade-Off Parameters for scrypt

log(N) (r=8,p=1)	6.93	10.35	13.66	16.94
No Tradeoff	42,650.62	2,053.98	30.97	-
Tradeoff = 2	22,153.09	2,333.11	39.84	-
Tradeoff = 4	15,870.75	1,552.01	45.02	0.37
Tradeoff = 8	9,548.42	949.73	49.06	0.23

Table 12. Obtained hashrates for scrypt. Computed as interpolation of the nearest measurements.