Evaluation of Standardized Password-based Key Derivation against Parallel Processing Platforms

Markus Dürmuth, Tim Güneysu, Markus Kasper,
Christof Paar, Tolga Yalcin, and Ralf Zimmermann

Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract. Passwords are still the preferred method of user authentication for a large number of applications. In order to derive cryptographic keys from (human-entered) passwords, key-derivation functions are used. One of the most well-known key-derivation functions is the standardized PBKDF2 (RFC2898), which is used in TrueCrypt, CCMP of WPA2, and many more. In this work, we evaluate the security of PBKDF2 against password guessing attacks using state-of-the-art parallel computing architectures, with the goal to find parameters for the PBKDF2 that protect against today's attacks. In particular we developed fast implementations of the PBKDF2 on FPGA-clusters and GPU-clusters. These two families of platforms both have a better price-performance ratio than PC-clusters and pose, thus, a great threat when running large scale guessing attacks. To the best of our knowledge, we demonstrate the fastest attacks against PBKDF2, and show that we can guess more than 65% of typical passwords in about one week.

1 Introduction

Password-based user authentication is the most widely used form of user authentication, and it will be in the foreseeable future. Alternative technologies such as security-tokens and biometric identification exist but have a number of drawbacks that prevent their wide-spread use outside of specific realms: Security tokens, for example, need to be managed, which is a complicated task for Internet-wide services with millions of users, they can be lost, and there needs to be some standardized interface to connect them to every possible computing device (including desktop computers, mobile phones, tablet PCs, and others). Biometric identification systems require extra hardware to read the biometrics, false-rejects cause user annoyance, and many biometrics are no secret (e.g., we leave fingerprints on many surfaces we touch). Passwords, on the other hand, are highly portable, easy to understand by users, and relatively easy to manage for the administrators. Still, there are a number of problems with passwords. Arguably the central theme is the trade-off between choosing a strong password versus one that is human-rememberable. Various studies and recommendations have been published presenting the imminent threat of insufficiently strong passwords chosen for security systems by humans (see, e.g., [4, 22, 37]).

Passwords are usually not stored in clear in computer systems but the hash of the password is stored instead. Consequently, *guessing attacks* are the most efficient method of attacking passwords, and studies indicate that a substantial number of passwords can be guessed with moderately fast hardware [38]. One measure to mitigate guessing attacks on passwords is to increase the time required to compute the key derivation function from the human-entered password. The most common approach nowadays is to run the password through a large number of hash function evaluations.

With the release of PKCS #5 v2.0 and RFC 2898 [14], a standard for password key derivation schemes based on a pseudo-random function (PRF) with variable output key size has been established. The specified Password-Based Key Derivation Function #2 (PBKDF2) has been widely employed in many security-related systems, such as TrueCrypt [35], OpenDocument Encryption of OpenOffice [25], and CCMP of WPA2 [12], to name only a few. The PRF typically involves an HMAC construction based on a cryptographic hash function that can be freely chosen by the designer. Besides the password, the PBKDF2 requires a salt S, a parameter for the desired output key length k_{Len}, and an iteration counter value c that specifies the number of repeated invocations of the PRF. While security aspects of salt and key length are quite well understood [19], it remains an open question how large c should be for practical use – especially with respect to adversaries who have access to very powerful computing resources, which have become more widely available in recent years. In particular, an impressive number of parallel computations, and thus password guessing attacks, can be performed with (clusters of) the latest many-core CPUs, highly thread-optimized graphics cards (GPUs), or modern Field-Programmable Gate Arrays (FPGAs). These latest platforms need to be considered when fixing c in practical systems. Note that recent security applications specify c typically to be in the range of 10^3 to 10^4 iterations (e. g., TrueCrypt performs between 1000 and 4000 iterations depending on the hash function applied). Referring to Paragraph 4.2 of RFC 2898, a minimum iteration count of 1000 is recommended in the original release of the standard. We argue that this number should be regularly updated to reflect the performance gains of the most recent high-performance computing platforms. In this work, more than 10 years after the initial release of RFC 2898, we will re-evaluate the security margin provided by PBKDF2 with respect to the password cracking performance of modern computing hardware.

Contribution: In this work we analyze the choice of security parameters for PBKDF2 for real-world systems against state-of-the-art attacks. More precisely, we consider different attack implementations on PBKDF2 using a range of different cluster systems employing recent CPU, GPU, and FPGA devices. As a practical case study, we take the recent security parameters used by TrueCrypt to implement attacks on PBKDF2. We compare the performance of our implementations to identify the most promising computing platform for the attack. To the best of our knowledge, we demonstrate the fastest known attack against PBKDF2. We combine these results with password guessing attacks based on Markov models [22, 7] to show that we can guess more than 65% of typical passwords in about one week. Finally, we derive recommendations how parameters for PBKDF2 should be chosen adequately.

Outline: In Section 2 we introduce some background on password-based key derivation, the PBKDF2 standard, and the state-of-the-art platforms for cracking passwords, followed by an introduction to password security and efficient password guessing in Section 3. In Section 4 we describe the relevant programming techniques of modern GPUs and our GPU implementation of PBKDF2. Likewise, in Section 5 we describe the FPGA cluster RIVYERA, and our implementation on this cluster. We compare the performance of the two implementations in Section 6, and discuss the implications of these results in Section 7.

2 Background and Related Work

With many keyboard-enabled computing systems, passwords are still state-of-the-art for user authentication. The standardized PBKDF2 maps passwords to secret keys that can be used for cryptographic operations. We review the basic operation of PBKDF2 and relevant previous work in the following.

2.1 Password-Based Key Derivation

The Password-Based Key Derivation Function #2 (PBKDF2) takes a user-defined PRF and requires four inputs to generate the output key k_{out} with

$$k_{\mathsf{out}} = \mathsf{PBKDF2}_{\mathsf{PRF}}(\mathsf{Pwd}, S, c, k_{\mathsf{Len}}),$$

where Pwd is the password, S the salt, c the iteration counter, and k_{Len} the desired key output length. By variation of the number of performed iterations c, it is possible to adjust the time needed for computation and thus, by selecting an adequately high number, key strengthening can be achieved rendering password related brute-force attacks less effective. In practice, common values for the applications mentioned above range between the recommended minimum of 1000 [14, 4.2] and 4000 iterations.

Figure 1 shows a simplified block diagram of the PBKDF2 scheme (specifically when using the SHA-512). An HMAC algorithm is repeatedly chained such that the outputs of all HMAC runs are added to the derived key. If the desired output key length is larger than the output of the hash function, the scheme is iterated multiple times, each time with a different counter value CNT. Depending on the input and output length two cases need to be distinguished: If the input length of the hash function is smaller than a padded hash-value, then the HMAC requires at least 6 executions of the compression function. Otherwise, an HMAC value can be computed by means of four executions of the compression function (e. g., RipeMD-160 and SHA-512).

As the password in each chain of the HMAC computations is the same, the outputs of the leftmost compression functions corresponding to the hashing

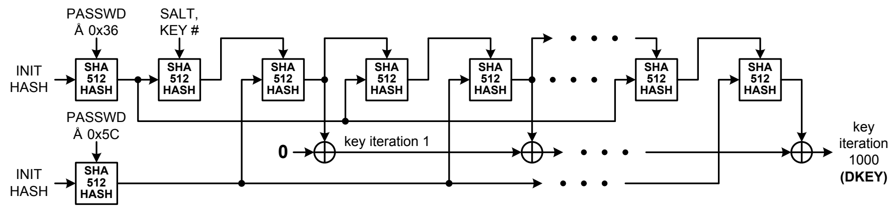

Fig. 1. SHA-512 based PBKDF2 scheme.

of the password xor `0x36..36` or `0x5C..5C`, will not change. Thus they can be computed exactly once per password and then be reused for all subsequent HMAC computations using the same password. Furthermore, the salt value will never change during our brute-force attack, so the hash value corresponding to the hashed salt can be reused when performing the HMAC chain for different counter values. These two measures reduce the required number of computations for a password evaluation to one half and one third for an HMAC with 4 and 6 invocations of the compression function, respectively.

In our evaluation, we have targeted TrueCrypt [35], a free open-source disc encryption software, where the password and salt sizes are fixed to 512 bits. For consistency, we consider TrueCrypt starting with Version 5.0 (released February 5, 2008). Since then, TrueCrypt uses AES-256, Serpent, and Twofish in XTS mode as block ciphers and generates the keys using either RIPEMD-160, SHA-512, or Whirlpool as supported hash functions. The number of HMAC iterations they require are 2000, 1000, and 1000, respectively and the corresponding number of hash runs are 4003, 2002, and 4002. The variation in the number of hash executions is due to the input block sizes of each hash function. TrueCrypt supports combinations of the block cipher algorithms. In the best case, when only one encryption algorithm is used, 512 key bits are required, and 1536 key bits in the worst case.

2.2 Processing Platforms for Password Cracking

Implementing password cracking on general purpose CPUs is straightforward, however, due to the versatility of their architecture, CPUs usually do not achieve an optimal *cost-performance ratio* for a *specific* application. As an example, there exist a number of cracking tools for TrueCrypt compiled for x86 CPUs, but few tools are available that go beyond re-using TrueCrypt-code, most notably True-Crack [34], which reports 15 passwords/sec on an Intel Core-i7 920, 2.67GHz. In the last years, other processing platforms have shown to exceed the performance (and cost-performance ratio) of conventional CPUs, for specific applications.

Modern graphics cards (GPUs) have recently evolved into computation platforms for universal computations. GPUs combine a large number of parallel processor cores (as of today up to 512 atomic cores and more) which allow highly parallel applications using programming models such as OpenCL or CUDA. Their usefulness for password cracking was demonstrated in particular by the Lightning

Hash Cracker developed by ElcomSoft, which achieves, for simple MD5-hashed password lists, a throughput rate of up to 680 million passwords per second using an NVIDIA 9800GTX2 [9]. Further work [10, 31] reports similarly impressive numbers with about 230 million SHA-1 (pure) hash operations per second on an NVIDIA 260GTX GPU. TrueCrack reports 330 passwords/sec on an NVIDIA GeForce GTX470, a press release [27] reports 2500 passwords/sec for Passware Kit 10.1, and a presentation [8] states that ElcomSoft software cracks 52400 passwords/sec on a Tesla S1070 with 4 GPUs for WPA-PSK, which essentially is PBKDF2 using only SHA-1.

Another way to tackle the large number of computations for password cracking efficiently is the deployment of special-purpose hardware. Moving applications into hardware usually provides significant savings in terms of costs and provides a boost in performance at the same time, since operations can be specifically tailored for the target application and potentially be highly parallelized. While Application Specific Integrated Circuits (ASIC) are expensive to develop due to their high non-recurring engineering costs, reconfigurable Field-Programmable Gate Arrays, or FPGAs, have been intensively studied by the crypto engineering community over the last 15 years. With today's powerful FPGA devices providing a configurable fabric consisting of millions of gate equivalences, it has become possible to create very fast implementations for specific computational problems. Given that password guessing is amenable to special-purpose hardware architectures and highly parallelizable, FPGAs are a promising platform for password cracking.

A third cost-effective platform for processing parallel applications is Sony's PlayStation 3 (PS3). Bevand [3], for example, presented a Unix crypt password cracker based on the IBM Cell Broadband Engine. However, the Cell processor is slightly outdated when comparing it to recent GPU and FPGA devices. Therefore, we do not expect the Cell processors to achieve a competitive cost-performance ratio, and we don't expect the PowerXCell 8i to become available at comparable prices in subsidized commodity game consoles. Thus, we did not include the Cell processor in our comparison.

3 Password Security

Accepted best practice mandates not to store the password pwd on the server in plain, but store the hash $h := H(pwd)$ of the password instead. In an *offline attack* on passwords, an attacker is given access to the value h and tries to recover the password pwd. (As opposed to *online guessing attacks*, where the attacker is only given access to a login prompt or similar.)

User-generated passwords usually have a rich structure, e.g., many are simple compositions of words from (English) language and numbers or special characters. Consequently, *guessing attacks*, where the attacker guesses a possible password, hashes it, and compares the hash to the stored value, are usually quite efficient. This has been realized early, and password guessing has been deployed for a long time (see, e.g, [4, 39, 40, 15]).

In a *dictionary attack*, the attacker has a list of words that are likely to appear in passwords. He computes the hashes of all these words and compares them with the stored hash. He can use additional *mangling rules*, e.g., appending special characters and numbers. Tools such as John the Ripper implement dictionary attacks and come with large dictionaries of common passwords, often grouped for different languages to better meet a specific site's needs. More recent work by Weir et al. [37] can be seen as generalization of this idea. Here, patterns that constitute extended mangling rules are extracted from real-world passwords using *probabilistic grammars* (context-free grammars with probabilities associated to production rules). These structures are then used to generate passwords, based on these structures and a dictionary as before.

3.1 Attacks Based on Markov Models

Another efficient way to guess passwords, first proposed in [22], is based on *Markov models*. These base on the observation that in human-generated passwords (as well as natural language), adjacent letters are not independently chosen, but follow certain regularities (e. g., the 2-gram `th` is much more likely than `tm`, in other words, the letter following a `t` is more likely an `h` than an `m`). In an n-gram Markov model, one models the probability of the next character in a string based on a prefix of length $n-1$. Hence, for a given string $c_1, \ldots, c_m$, we can write $P(c_1, \ldots, c_m) = P(c_1, \ldots, c_{n-1}) \cdot \prod_{i=n}^{m} P(c_i | c_{i-n+1}, \ldots, c_{i-1})$.

In the *training phase*, the attacker learns the conditional probabilities from lists of leaked plaintext passwords (e. g., the RockYou password list), from available password dictionaries, or from plain English text. In the *attack phase*, the attacker generates passwords that are likely according to the Markov model. Additionally, one filters for certain patterns that typically occur for passwords; one defines finite automata for these patterns, and the algorithm ensures that only passwords that are accepted by one of the automata are tested. (An example for such a pattern is that in alpha-numeric passwords, the numerals are very likely at the end of the password (e. g., `password1`).

We use an implementation of Markov-based password guessers from [7] to feed our implementation with passwords. This algorithm additionally enumerates passwords in (approximately) decreasing order of likelihood, which substantially speeds up the guessing of frequent passwords, and does not use the hand-crafted patterns from [22]. We train the algorithm with the RockYou dataset, a dataset of 32 Million passwords that was leaked in an SQL injection attack in 2009 in clear. This dataset is publicly available and regularly used for password research. In this work we publish no information about specific passwords from the list, so we do not see ethical problems in using this list.

3.2 Further Related Work

Using precomputations, *rainbow-tables* can be used to speed up the guessing step [11, 26]. An implementation of rainbow-tables in hardware is studied in [20]. A problem closely related to password guessing is that of *estimating the strength* *of a password*, which is of central importance for the operator of a site to ensure a certain level of security. In the beginning, password cracking was used to find weak passwords [21]. Since then, much more refined methods have been developed. Later, one used so-called pro-active password checkers to exclude weak passwords [33, 17, 4, 28, 5]. 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, 18, 6]. More recently, Schechter et al. [30] 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].

4 GPU-based Attack

Next, we describe our implementation on GPUs as well as the required technical background on GPU programming.

4.1 Introduction to GPU Programming

Within the last decade, the roles of GPUs changed from mere graphic processors to general purpose processing units. Today, there are programming interfaces from all major graphic processor manufacturers, providing easy access to the processors of the graphic hardware, e.g., CUDA [24] developed by NVIDIA or Stream [1] for AMD GPUs. For heterogeneous processor platforms, supporting both CPUs and GPUs, OpenCL [16] has established combining the computational power of recent computer systems. In this section we will focus on NVIDIA GPU devices using the CUDA programming interface.

CUDA Terminology and Code Execution Basics: 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 is a kind of user-managed cache area, and can thus interact with each other. Furthermore, threads within a block can be synchronized with each other. Blocks define up to 3 dimensions to index individual threads by x, y, and z coordinates within the kernel code. The dimension of the blocks are provided as a parameter when calling a kernel from host (i.e. CPU) code. The blocks themselves are organized within a grid. During execution, blocks are assigned to *Streaming Multiprocessors* (SMs). An SM then schedules its pending blocks in chunks of 32 threads (a warp) to its hardware, where each thread within a warp executes the same instruction. When threads are scheduled for high-latency memory instructions, the scheduler will execute additional warps while waiting for the memory access to finish. This mechanism of latency hiding is one of the main reasons for the superior performance of GPUs: Whenever there are enough independent instructions on an SM that do not depend on previous results the hardware can completely hide the latency of memory accesses, by meanwhile using the idle computing cores to process the instructions of other warps.

NVIDIA'S Tesla C2070 GPU: For our experiments, we use a machine equipped with four Tesla C2070 GPUs by NVIDIA [23]. A single Tesla C2070 GPU consists of 14 SMs. Each SM has its own set of 32 computing cores, i. e., the architecture provides 448 cores within a single GPU. It provides a high memory bandwidth of 144 GB/s and a low computational overhead to initiate and manage parallel computations. The cores are running at 1.15GHz and can reach a single-precision floating point performance (Peak) of up to 1.03 TFLOPS (NVIDIA [23]). (For comparison: Intel's recent Core i7 980 CPUs running at 3.6GHz are listed at 86 GFLOPS (Intel [13]). We refer to NVIDIA'S website [24] for more detailed information about CUDA and the Tesla GPUs.

4.2 Implementing the KDF

In the following we describe the implementation aspects of our GPU implementation of the PBKDF2 scheme, following the specification of the PBKDF2 as employed by TrueCrypt. To implement the PBKDF2, we decided to aim at an implementation that avoids high-latency accesses to the main memory of the GPU by using only fast registers and shared memory. The other major strategy was to avoid redundant computation as detailed in Section 2.1. In the following we provide an overview of the algorithm specific aspects of the three hash functions RipeMD-160, SHA-512, and Whirlpool.

RipeMD-160: The state of the RipeMD hash function has a size of 320 bit, which is divided into a left and a right part, each consisting of five 32 bit values. Both parts can be processed independently. For this reason, we decided to let two threads team up to process the hashing of one key candidate. Here one thread processes the left part of the RipeMD algorithm and the other one the right part. The state, the intermediate keys, and the two hashes of the passwords are kept in registers. Shared memory is used to synchronize each thread pair and to provide input values (i. e., previous hash and message) to the compression function. The algorithm has been manually unrolled replacing all known inputs by constants residing within the kernel code. The kernel uses an overall of 40 registers and 5376 bytes of shared memory (64 passwords * (16 registers for inputs + 5 registers for outputs) * 4 bytes per 32 bit value) and runs with 128 threads per block. This allows 6 blocks in parallel per SM and an equivalent of 5376 passwords that can be processed in parallel on each GPU.

SHA-512: The state of SHA-512 consists of eight 64 bit values. Compared to the RipeMD-160 state, this complicates the computation of the compression function in two ways: On the one hand, the GPU hardware is a native 32 bit architecture (with some 64 bit extensions), slowing down most computations. On the other hand, many registers and a lot of shared memory is needed to store the state, the two hash values of the password, and the intermediate keys.

For this reason our SHA-512 implementation uses only 64 threads per block and compiles to 63 registers per thread and 4096 bytes of shared memory per block. Here 63 registers per thread are the upper bound the hardware can handle. This results in a spill of used variables into the slow device memory. Nevertheless, as the number of spilled variables is small, the device memory should be able to permanently keep them within the still reasonably fast devices memory cache. This kernel again allows 5376 passwords to be processed in parallel.

Whirlpool: The state of Whirlpool has the same size as for SHA-512, which again leads to high register pressure. We implemented the Whirlpool hash function with a table lookup implementation using eight 256×32 bit lookup tables stored in shared memory. We employ 128 threads per block, each using the maximum of 63 registers. The shared memory usage of each block is 16384 bytes per block and only 4 blocks will run in parallel on each SM. Each block processes 128 passwords, such that we achieve 7168 passwords that are processed in parallel.

4.3 Wrapper Implementation

We use a host system powered by two Intel Xeon X5660 six-core CPUs at 2.8GHz with enabled Hyperthreading and AES-NI instruction support. It is equipped with four Tesla C2070 GPUs connected by full PCIe 2.0 16x lanes. We use CUDA Version 4.1 and the CUDA developer driver 286.19 for Windows 7 (x64). The host system generates the passwords in a single threat, writing them to a memory buffer. We schedule passwords in chunks of 21504 passwords, i.e, $14 \cdot 6 \cdot 4 = 336$ blocks for RipeMD-160 and SHA-512 and $14 \cdot 6 \cdot 2 = 168$ blocks for Whirlpool. This number of blocks has been selected to be a small multiple of the maximum number of concurrent blocks on the GPU for all implemented kernels. This way the GPU hardware should always be fully occupied with respect to the number of scheduled blocks for maximum performance. The derived key material is copied back to the host memory to test for the correct decryption of the TrueCrypt header. As the host system is idle during the GPU computations, the password verification (which is much less computationally expensive) can be hidden within the kernel execution time of the GPU computations. For our experiments the implementation on the host system re-uses large parts of the cryptographic primitives from the original TrueCrypt implementation sources. To overlap memory copies between host and GPU with computations, we employed four streams per GPU. Furthermore each stream alternately uses four sets of password and result buffers. This way the GPU can process the next password chunk without having to wait for the host to finish checking the latest generated key material. The implementation is capable of generating both 1536 bits and 512 bits of key material for a password and an HMAC candidate function, according to the worst case in the TrueCrypt specification.

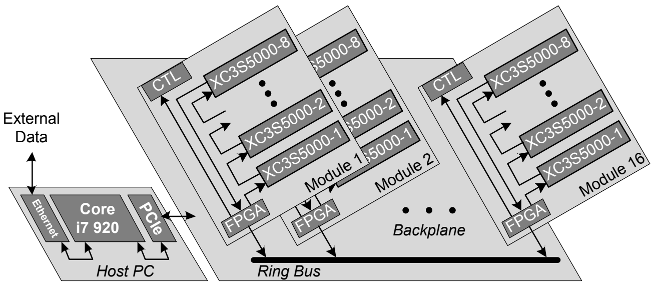

Fig. 2. The RIVYERA cluster architecture.

5 FPGA-based Attack

FPGAs combine the performance of a gate-level hardware implementation with flexibility, simple development, and reconfigurability of a software-based approach. Furthermore, FPGA implementations are truly parallel in nature. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously. This has made them an ideal choice for cryptoanalytic applications, where several instances of the algorithm under test has to be evaluated in parallel with different parameters.

5.1 RIVYERA – An FPGA-based Cluster System

The RIVYERA FPGA cluster [32], with its 128 Spartan-3 XC3S5000 FPGAs and an optional 32MB memory per FPGA, is a powerful and cost-optimized cryptanalytical machine. All FPGAs are connected with two opposite directed, systolic ring networks that directly interface with the Intel Core i7 based PC (which is integrated in the same housing) via a PCI Express communication controller, as shown in Figure 2.

In our FPGA-based attack on TrueCrypt, we implemented the PBKDF2 scheme on the RIVYERA cluster, balancing the different parts of the algorithm in terms of area and speed. In accordance with the goal of the PBKDF2 algorithm to derive a key using a hash function and perform encryption/decryption afterwards, sufficient key material has to be generated by running the hash function n times. An optimal strategy is to connect several copies of a hash function in a pipelined design in order to get the highest possible throughput. However, the high number of iterations n (1000 to 4000) makes this approach impossible.

The three hash functions used by TrueCrypt need a different amount of clock cycles to complete processing and also have different critical paths, resulting in different processing times. Partitioning parts of an FPGA between these three hash functions would result in a slower and more complex design. Therefore, we chose to implement individual systems for each hash function used and distribute them among multiple FPGAs. This also adds flexibility to implement higher percentage of a favored algorithm, e. g., in case the used algorithm is known or has a higher probability.

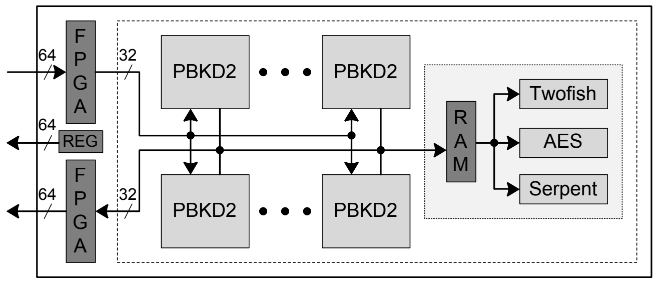

Fig. 3. Top-Level view of the FPGA Design

5.2 Implementing the KDF

Password-based Key Derivation Function #2 relies on repeated executions of a hash function in HMAC construction, where the result of each HMAC is accumulated starting with an initial all-zero key, until the final key is derived at the end of all HMAC runs.

We designed three independent single iteration cores, one for each of the three target hash functions, optimized for time-area product. The other important parameter is the number of key bits that can be generated by each PBKDF module. It is equal to the predefined message digest size of the incorporated hash function, which is 512 bits for both SHA-512 and Whirlpool, but only 160-bits for RipeMD-160. This means that while three instances of either SHA-512 or Whirlpool cores are sufficient to supply the worst case of 1536-bits key (required for Twofish, AES, and Serpent combination), the same can be accomplish with ten instances of the RipeMD-160-based PBKDF core, making it the most critical part of the whole design.

Implementing for FPGAs, the predefined topology of resources is the most limiting and hence the most important factor. It is imperative to come up with a balanced design that uses both registers and block RAMs to the highest possible ratio while losing minimum cycles for additional RAM access. For this purpose, the initial values, constants and hash results are stored in the block RAMs, while registers are utilized for storage of internal iteration variables within each hash function in all our hash cores. As mentioned above, we have developed three different FPGA designs – each targeting one hash function as shown in Figure 3 – and distributed them among the 128 FPGAs on the RIVYERA cluster.

The design uses a 64-to-32 bit input FIFO to split the data from the RIVYERA bus to the local bus architecture and switch between the system clock domain and the computation clock domain. All PBKDF2 units are initialized using the salt from the TrueCrypt header and the passwords are distributed among free units. After receiving a password, each unit immediately starts processing. As soon as a unit finishes its execution, its result is written into a dedicated memory, where the optional cipher blocks can access it and perform the on-chip test phase. An additional 64-bit register stores all information on the current FPGA operations, which the host application can access at any time. Since the additional area taken by the on-chip test is not suitable for all hash functions, the option to read the derived keys read back to the host PC for offline key tests is also supported in order to save resources for more on-chip key derivation units.

The password list, generated by a password derivation program, is transmitted by a host program (running on the Core i7 in the RIVYERA) to the FPGAs using the PCI Express architecture. Each of the three PBKDF units implements the scheme in Figure 1 with minor differences. The basic idea is to first hash the password XORed with IPAD and then with OPAD and store the two results as they will be repeatedly used during further iterations as initial values of the hash function. The next step is to hash the combination of SALT and key number (which is between 1-3 for SHA-512 and Whirlpool, and between 1-10 for RipeMD-160) in order to obtain the input value for the next run of the hash core. In all the following runs, the output of the previous run is the input data, and one of the two stored password hash results (in alternating order) is the initial value. The output of every second hash run (chaining variable) is accumulated (starting with all zero value) to get the final derived key. In the following paragraphs, we present the specific details for each different algorithm.

RipeMD-160: The RipeMD-160 based PBKDF core uses a 512-bit input message and hashes it by mixing with a 160-bit chaining variable which is updated in 80 rounds. At the end of all rounds, the chaining variable is added to the previous hash value. The internal round function is similar to that of SHA-1. However, the RipeMD round function has two parallel paths, whose results are stored in two 160-bit parallel registers, while the final hash result is stored in block RAMs. At the end of each round, the previous hash result, read from the RAM in 32-bit words, is added to the corresponding word of the update value from the current hash run, and then written back to the RAM. While this causes additional cycles, it saves more than 160-bit of registers and 128-bit of adders, resulting in further time-area product optimization. The total cycle count for each hash run is 95 cycles, in comparison to the ideal case of 80 cycles.

The RipeMD-160 core is run twice for the SALT and key number due to its 512-bits input block size. Since the total number of key iterations is defined as 2000 for RipeMD-160, this results in a total of $(5 + 1999 \cdot 2) \cdot 95 = 380285$ cycles for key derivation per core, each of which occupies 1032 slices (461 FF, 1764 LUTs) on a Xilinx Spartan-3 FPGA.

SHA-512: Each SHA-512 PBKDF core operates on 1024-bit message blocks and generates a 512-bit message digest. The intermediate hash values and the internal chaining variables are processed on a 32-bit datapath, which is not only compatible with the existing 32-bit block RAMs, but also minimizes delay paths. The only drawback is the number of cycles per hashing, which is 200 instead of the ideal case of 80. However, this time-area product optimization is well justified with increase in frequency and reduction in area.

Each SHA-512 based key derivation requires 1000 PBKDF iterations, which correspond to a total number of $(4 + 999 \cdot 2) \cdot 200 = 400400$ cycles for key

Table 1. Implementation Results of PBKDF2 on 4 Tesla C2070 GPUs

Hash	RIPEMD	SHA-512	Whirlpool	RIPEMD SHA-512 Whirlpool	RIPEMD	SHA-512	Whirlpool	RIPEMD SHA-512 Whirlpool
Derived Key Length	512 bits				1536 bits			
Passwords/sec	72786	105351	50686	23366	29330	35246	16980	8268
Passwords/sec (demo tool)	51661	54874	36103	19627	27591	29892	12153	6858

derivation per SHA-512 PBKDF core, each of which occupies 1001 slices (897 FFs, 1500 LUTs) on a Xilinx Spartan-3 FPGA.

Whirlpool: The structure of Whirlpool [2] significantly differs from the structures of the other two cores. It not only generates a 512-bit message digest, but also processes 512-bit message blocks. The internal structure of Whirlpool resembles a block cipher with two identical datapaths in parallel; one as key expansion module, the other as message processing module. The internal structures of each path are identical. However, the key expansion module uses hash input to generate round keys, while the message processing module uses message inputs together with round keys to generate the next state of the hash.

Whirlpool hashing needs to be executed four times during each iteration due to the equal input and output sizes. However, only 10 iterations allow a word-serial implementation, where the message and the hash (key) are processed in 64-bit chunks, considerably reducing the overall area. The total number of cycles per round becomes 9 and the total number of rounds becomes 11 (including the initial whitening), which results in 99 cycles per round. With a total number of $(6+999\cdot 4)\cdot 99 = 396198$ cycles for key derivation, each Whirlpool PBKDF core occupies 6013 slices (1131 FFs, 10878 LUTs) on a Xilinx Spartan-3 FPGA.

6 Results

In the sequel we present performance numbers for the above experiments.

6.1 Performance Numbers

GPU Implementation: Table 1 gives the performance results for each hash algorithm for the worst case (i. e., 1526 bit of key material) and the fastest case (i. e., 512 bit of key material) of TrueCrypt's password derivation. The latter case corresponds, e. g., to AES-256 in XTS mode, while the first one corresponds to a cascade of all three TrueCrypt ciphers. These numbers clearly show that the implementations scale linearly: The performance boost for the smaller key sizes

Table 2. Implementation results and performance numbers of PBKDF2 on the RIVYERA cluster (Place & Route) without on-chip verification. Please note that the current version is not optimized for speed and uses the lowest clock frequency valid for all designs, as the place & route results for the optimized version are still pending.

Hash Clock cycles per PBKDF2	RIPE-MD 380,285		SHA-512 400,400		Whirlpool 396,198	
Derived Key Length	1536 bit	512 bit	1536 bit	512 bit	1536 bit	512 bit
PBKDF2 Units	4	9	11	32	3	15
Hash Cores per PBKDF2	10	4	3	1	3	1
FPGA Resources (Slices)	29753	28227	31773	31943	18370	29528
FPGA Resources (%)	89%	84%	95%	95%	55%	88%
Passwords per sec per FPGA	368	828	957	2784	265	1325
Passwords per sec	**47 104**	**105 984**	**122 496**	**356 352**	**33 920**	**169 600**

corresponds to the difference in the number of blocks that need to be hashed to derive the desired output lengths, i. e., 4 vs. 10 rounds for RipeMD and 1 vs. 3 rounds for SHA-512 and Whirlpool.

When deriving 1536 bit of key material per password for each of the three hash algorithms RipeMD-160, Whirlpool, and SHA-512, our fastest implementation using a hardcoded salt was able to derive the key material at 8,268 passwords per second, i. e., about 714 million passwords per day and 21.4 billion passwords per month. Using only the TrueCrypt default settings of RipeMD-160 and AES-256 in XTS mode, i. e., 512 bit of key material are generated, the performance boosts to 72,786 passwords per second, 6.29 billion passwords per day and 188 billion passwords per month.

Our fully implemented TrueCrypt cracker tool consists of the password generator, the PBKDF2 and the decryption of the header data to verify the material. We observe a maximum speed limit of around 55,000 passwords per second, which is the speed of the used password generator. This limitation can be leveled by further optimizations. For the sake of completeness, we also provide the performance figures of the full tool. We want to mention that our numbers, as all specific implementations, can only provide a lower bound: implementations using other GPU architectures or further optimized code may improve the results.

FPGA Implementation: In case of the FPGA based key password search, we use different FPGA configurations for the best case (single block cipher) and the worst case (cascade of all three block ciphers).

Figure 2 shows the place and route results. With respect to the single instance capabilities (i. e., when targeting one specific PBKDF2 instance of TrueCrypt, the RIPE-MD design can derive 368 passwords per second for 1536 bit output and up to 828 for 512 bit output on a single FPGA, respectively. This scales to 47,104 and 105,984 passwords per second on the full RIVYERA, taking only this hash algorithm into account. The SHA-2 implementation is faster and computes 957 and 2,784 passwords per second, respectively, and using the full RIVYERA provides a throughput of 122,496 and 356,352 for the 512 and 1536 bit case,

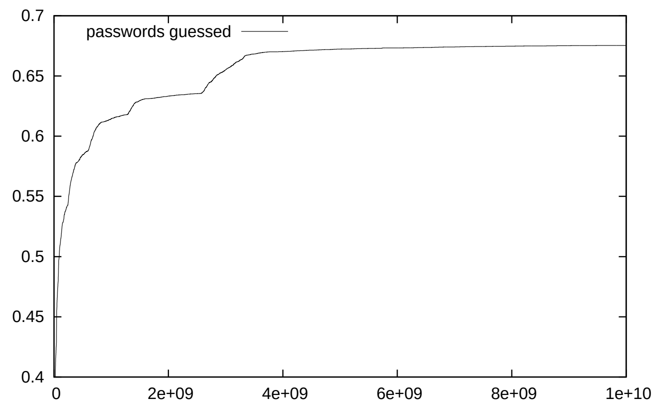

Fig. 4. Fraction of passwords guessed correctly (y-axis) vs. total number of guesses (x-axis).

correspondingly. Even though the current Whirlpool implementation does not utilize the complete FPGA area optimally due to the PBKDF2 block size, it is more than 50% faster than the RIPE-MD scheme when used for 512 bit. In order to test all three hash functions for TrueCrypt, we utilize the full RIVYERA sequentially, as the reprogramming time is negligible.

The bottleneck on FPGAs is also the host-based password generation and the throughput drops a bit due to the offline verification. But with the remaining area on the FPGA, we can build an on-chip verification. As the amount of clock cycles necessary to perform a key derivation is very large compared to the number of cycles needed to compute the block ciphers, the loss of area may be worth the speed gain. In this case, all remaining cores of the i7 host CPU could work on producing passwords in parallel, thus removing this bottleneck as well.

Comparing a single GPU and FPGA device, it turns out that GPUs are significantly better in hashing than FPGAs (e.g., 18,000 vs. 828 RIPE-MD passwords per second). We attribute this result to the high clock frequency and the underlying 32-bit micro-architecture of GPUs that finally provides the distinct advantage with 32-bit-based hash functions. It is difficult to compare the individual device costs, since both platforms cannot be used as a stand-alone device without significant overhead. However, in case we relate the overall financial system costs of our GPU system and the RIVYERA cluster, we yield a scaling factor of 3.3 in favor for the GPU cluster.

6.2 Search Space and Success Rate of an Attack

In order to determine the actual influence of the number of guessed passwords from the last section, we determine the percentage of passwords one can break (on average) with that number of guesses. To this end, we use an implementation of a Markov model based password guesser from [7] (see Section 3 for more details). As *training set* used to derive the Markov model we used a random selection of 90% of the RockYou password list, the test set consists of the remaining 10% of the RockYou list (still more than 3 million passwords).

Figure 4 shows the fraction of passwords guessed correctly (y-axis) for a certain number of guesses made (x-axis). These results were obtained by running the password generator independently of the hashing engine. The reason is that, in order to incorporate the hashing engine, we would need to generate TrueCrypt containers for each password in the test set, which is prohibitively time-consuming. From the numbers in the previous section we can estimate that, in the absolutely worst case, we can guess more than 65% of the passwords from the RockYou list in a week and more than 67% in a month.

7 Conclusions and Recommendations

Carefully chosen passwords are essential to protect systems using passwords (for recommendations on choosing good passwords, see, e. g., Appendix A of NIST SP 800-63). But even though PBKDF2 was specifically designed to prevent simple brute-force attacks, we showed that parallel hardware platforms are capable to comb through a significant amount of passwords per second (356,352 passwords per second for SHA-2/512 bit case). Our results indicate that GPU clusters have a better cost/performance ratio than FPGAs, mainly due to the low prices of the wide-spread use of GPUs.

The main parameter of PBKDF2 specifying the level of protection is the iteration counter c. Due to the progress in technology (outlined by Moore's law), we do not consider it sufficient for a secure system to run a constant (minimum) amount of 1000 hash iterations in the lifetime of an application or a system, as defined by RFC 2898 for PBKDF2. We therefore recommend to replace this constant iteration count c with a dynamic variable that is stored in each respective application instance and which is adjusted over time according to technological scaling effects. The iteration count c should be lower-bounded by the computational resources of the least-capable target platform of the application. Note, however, that even recent "low-end" processing device (e.g., smart phones) often provide powerful ARM processors with 1GHz or more so that running 4000-10000 hash iterations is certainly feasible even on these devices.[1] Note that an update of this dynamic iteration count is simple and can take place frequently right after unlocking the application instance with the correct password.

Finally, we like to point out the structural limits of password-based key derivation. Even if we assume a much stronger key derivation function than PBKDF2 being available[2] so that much less passwords can be searched per second, we still achieve with our approach a significant coverage of the password space due to limited selection criteria of human-chosen passwords (see Fig. 4). Although certainly no real news, we need to emphasize the importance of choosing

[1] For recent performance figures of hash functions on a wide range of low-cost and high-performance CPUs, see `http://bench.cr.yp.to/primitives-hash.html`.

[2] For alternative proposals on password-based key derivation, see for example [29].

strong passwords, possibly combined with additional security credentials such as cryptographic hardware tokens or biometrics.

References

1. AMD. ATI Stream Technology (Website), 2011. `http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx`.
2. P. Barreto and V. Rijmen. The Whirlpool hashing function. In *First open NESSIE Workshop, Leuven, Belgium*, volume 13, page 14, 2000.
3. Marc Bevand. *Breaking UNIX crypt() on the PlayStation 3.* (Presentation, ToorCon 10), September 2008.
4. M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.
5. William E. Burr, Donna F. Dodson, and W. Timothy Polk. Electronic authentication guideline: NIST special publication 800-63, 2006.
6. Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from Markov models. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 2012.
7. Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Personal communication, 2012.
8. Elcomsoft. GPU assisted password cracking. Online at `http://www.slideshare.net/andrey.belenko/gpuassisted-password-cracking`.
9. ElcomSoft. Lightning Hash Cracker, Nov 2011. `http://www.elcomsoft.com/lhc.html`.
10. Ivan Golubev. IGHASHGPU, Nov 2011. `http://www.golubev.com/hashgpu.htm`.
11. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
12. IEEE Computer Society. IEEE Standard for Information technology 802.11 - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements, Jun 2007. `http://standards.ieee.org/getieee802/download/802.11-2007.pdf`.
13. Intel. Intel® Core i7-900 Desktop Processor Series, 2011. `http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf`.
14. B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000. `http://tools.ietf.org/html/rfc2898`.
15. G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.
16. Khronos Group. OpenCL - The open standard for heterogeneous systems (Website), 2011. `http://www.khronos.org/opencl/`.
17. D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proc. USENIX UNIX Security Workshop*, 1990.
18. Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *CHI 2011: Conference on Human Factors in Computing Systems*, 2011.
19. A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

20. Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid 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.
21. R. Morris and K. Thompson. Password security: a case history. *Communications. ACM*, 22(11):594 – 597, 1979.
22. Arvind Narayanan and Vitaly 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.
23. Nvidia. TESLA C2050/C2070 GPU Computing Processor, 2010. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`.
24. Nvidia. CUDA Developer Zone (Website), 2011. `http://developer.nvidia.com/category/zone/cuda-zone`.
25. OASIS. Open Document Format for Office Applications (OpenDocument) Version 1.2, April 2012. `http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2-part3.html`.
26. 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.
27. Passware Kit 10.1 – Press Release. `http://www.prnewswire.com/news-releases/passware-kit-101-cracks-rar-and-truecrypt-encryption-in-record-time-99539629.html`.
28. The password meter. Online at `http://www.passwordmeter.com/`.
29. Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan*, 2009.
30. Stuart Schechter, Cormac Herley, and Michael 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.
31. Marc Schober. Efficient password and key recovery using graphics cards. Master's thesis, Ruhr-Universität Bochum, 2010.
32. SciEngines GmbH. RIVYERA S3-5000, 2010. `http://www.sciengines.com/joomla/index.php?option=com_content&view=article&id=60&Itemid=74`.
33. E. H. Spafford. Observing reusable password choices. In *Proceedings of the 3rd Security Symposium*, pages 299–312. USENIX, 1992.
34. Truecrack. Online at `http://code.google.com/p/truecrack/`.
35. TrueCrypt - Free Open-Source On-The-Fly Encryption, Nov 2011. `http://www.truecrypt.org/`.
36. Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry 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. Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy*, pages 391–405. IEEE Computer Society, 2009.
38. Openwall Community Wiki. John the Ripper benchmarks, April 2012. `http://openwall.info/wiki/john/benchmarks`.
39. T. Wu. A real-world analysis of kerberos password security. In *Network and Distributed System Security Symposium*, 1999.
40. M. Zviran and W. J. Haga. Password security: an empirical study. *J. Mgt. Info. Sys.*, 15(4):161–185, 1999.