MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Key derivation functions and their GPU implementation

BACHELOR'S THESIS

Ondrej Mosnáček

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondrej Mosnáček

Advisor: Ing. Milan Brož

Acknowledgement

I would like to thank my supervisor for his guidance and support, and also for his extensive contributions to the Cryptsetup open-source project.

Next, I would like to thank my family for their support and patience and also to my friends who were falling behind schedule just like me and thus helped me not to panic.

Last but not least, access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is also greatly appreciated.

Abstract

TODO

Keywords

key derivation function, PBKDF2, GPU, OpenCL, CUDA, password hashing, disk encryption, password cracking, LUKS

Contents

1 Introduction

Encryption is the process of encoding information or data in such a way that only authorized parties can read it [19, 7]. The encryption uses a parameter – the key. The key is an information that is only known to the authorized parties and which is necessary to read the encrypted data. In general, any piece of information can be used as the key, but since it usually has to be memorized by a human, it often has the form of a password or passphrase.

Passwords and passphrases generally have the form of text (a variable-length sequence of characters), while most encryption algorithms expect a key in binary form (a long, usually fixed-size, sequence of bits or bytes). This means that for any password- or passphrase-based cryptosystem it is necessary to define the process of converting the password (passphrase) into binary form. Merely encoding the text using a common character encoding (e. g. ASCII or UTF-8) and padding it with zeroes is often not sufficient, because the resulting key might be susceptible to various attacks. An *attack* on a cryptographic key is an attempt by an unauthorized party to determine the key from publicly known information or from a certain partial information about the key (e. g. some knowledge about the domain from which the key was chosen, the first few bits of the key, etc.)."

For this reason, a cryptographic primitive called *key derivation function* (KDF, plural KDFs) is used to derive encryption keys from passwords. KDFs are also often used for *password hashing* (transforming the password to a hash in such a way that it is easy to verify a given password against a hash, but infeasible to determine the original password from the hash) or *key diversification* (also *key separation*; deriving multiple keys from a master key so that it is infeasible to determine the master key or any other derived key from one or more derived keys) [24, 3].

KDFs usually have various security parameters, such as the number of iterations of an internal algorithm, which control the amount of time or memory required to perform the derivation in order to thwart brute-force attacks. Another common parameter is the cryptographic salt, which is a unique or random piece of data that is used

together with the password to derive the key. Its main purpose is to protect against dictionary and rainbow table attacks and it is usually not kept secret [9].

One possible application of KDFs is key derivation from passwords in disk encryption software. Disk encryption software encrypts the contents of a storage device (such as a hard disk or a USB drive) or its part (a disk volume or *partition*) so that the data stored on the device can only be unlocked by one or more passwords or passphrases. The password/passphrase is entered when the user boots an operating system from the encrypted device or when they mount the encrypted partition to the filesystem.

An example of a disk encryption program is *cryptsetup*[1] which uses the LUKS standard as its main format for on-disk data layout. In version 1 LUKS uses PBKDF2 as the only KDF for deriving encryption keys from passwords [6]. However, PBKDF2 has a range of weaknesses, one of them being high susceptibility to brute-force and dictionary attacks using GPUs[2], as this thesis aims to demonstrate.

1.1 Goals

The goal of this work is to compare the speed of a brute-force attack on a specific key derivation function (PBKDF2) performed on standard computer processors against an attack using GPUs.

Modern GPUs can be programmed using various high-level APIs (such as OpenCL[3], CUDA[4], DirectCompute or C++ AMP) and can be used not only for graphics processing but also for general purpose computation. Due to their specific architecture GPUs are suitable for parallel processing of massive amounts of data. Tasks that can be split into many small subtasks which can be run in parallel can be processed by a single GPU several times faster than by a single CPU. As was shown by Harrison and Waldron [8], using GPUs it is possible to accelerate also various algorithms of symmetric cryptography.

This work also includes analysis of susceptibility of PBKDF2 to

1. https://gitlab.com/cryptsetup/cryptsetup/wikis/home
2. GPU = Graphics Processing Unit
3. https://www.khronos.org/opencl/
4. http://www.nvidia.com/object/cuda_home_new.html

attacks using GPUs and the implementation of a demonstration program performing a brute-force attack on the password of a LUKS[5] encrypted partition.

1.2 Summary of results

TODO

1.3 Chapter contents

TODO

5. LUKS = Linux Unified Key Setup

2 Key derivation functions

Key derivation functions are cryptographic primitives that are used to derive encryption keys from a secret value. Depending on the application, the secret value can be another key or a password or passphrase [24]. A KDF that is designed for deriving cryptographic key from another key is called a *key-based key derivation function* (KBKDF); a KDF that is designed to take a password or passphrase as input is called a *password-based key derivation function* (PBKDF).

2.1 Key-based key derivation functions

Key-based key derivation functions are most often used to derive additional keys from a key that already has the properties of a cryptographic key – that is, it is a truly random or pseudorandom binary string that is computationally indistinguishable from one selected uniformly at random from the set of all binary strings of the same length [3].

Since the input to a KBKDF is already a cryptographic key, KBKDFs usually do not try to make brute-forcing more difficult by making the algorithm more computationally complex. A good cryptographic key has entropy of at least 128 bits, which means there are at least 2^{128} possible keys. Testing so many keys would be infeasible even with a very fast algorithm and an enormous computer cluster.

An example of a simple KBKDF is HKDF (HMAC-based extract-and-expand Key Derivation Function), which proceeds in two stages. The optional *extract* stage first extracts a suitable pseudorandom key from the (possibly low-entropy) input key material and an optional salt. Then the *expand* stage expands the extracted pseudorandom key, along with an optional context and application specific information (this can be used for key diversification), to the output key of the desired length [10, 12].

2.2 Password-based key derivation functions

As opposed to key-based key derivation functions, password-based key derivation functions are designed specifically to take low-entropy input such as a password or passphrase and to resist brute-force and dictionary attacks.

A *brute-force attack* is a kind of cryptanalytic attack in which the attacker attempts to determine a cryptographic key or password by systematically verifying (testing) all possible keys (this is also referred to as *exhaustive key search*) [19] or a large portion of all possible keys. In order to be able to perform the attack, the attacker must have a means to verify an unlimited number of different keys.

A *dictionary attack* is a kind of cryptanalytic attack in which the attacker attempts to determine a password by going through all passwords from a list of common passwords and/or (variations of) words from a dictionary [19]. Dictionary attacks tend to be very successful in practice thanks to the users' tendency to pick very simple and predictable passwords.

When the attacker tests the keys against a live system, the attack is called an *online attack*. When the attacker holds an information sufficient to verify the keys on their own (for example one or more encryption plaintext and cryptotext pairs or password hashes), they can perform an *offline attack* [19]. An offline attack is usually significantly faster than an online attack because the attacker can optimize the verification algorithm and/or use specific hardware to accelerate the verification.

PBKDFs aim to reduce the feasibility of these attacks by increasing the amount of time and/or memory required to test a single key. The basic principle of this approach is that in practice, the extra time/memory requirements are only a minor inconvenience for a user (especially given that these measures provide an increased resistance against a mass attack), while for an attacker (who has to repeatedly process millions or billions of possible passwords) this means that the resources needed to perform a brute-force (or dictionary) attack increase significantly.

A well-designed PBKDF also takes into account the difference between the hardware that is used in the legitimate scenario and the hardware that the attacker might have available. A highly motivated and well-funded attacker might have access to massive amounts of computing power, might often be willing to wait even years until the password is found and might possess an expensive specialized hardware which would minimize the time and resources needed to successfully break the password. A typical user, on the other hand, will use a consumer-grade hardware (such as a personal computer or a laptop), so the PBKDF should be designed so that it performs with reasonable efficiency on the user's hardware, but at the same time it is difficult to utilize specialized hardware to gain advantage in an attack [16].

In order to protect against attacks using highly parallel architectures, modern PBKDFs use *sequential memory-hard functions*, which are designed in such a way that any time-efficient computation needs to use a certain configurable amount of memory (for a formal definition see [14, chapter 4]). Requiring a certain non-trivial amount of memory to compute a single instance of the PBKDF increases the size of each compute unit, thus making any increase in parallelism more expensive (as opposed to functions that use only a small constant amount of memory) [14].

Another desirable property of PBKDFs is the ability to upgrade an existing derived key/hash to another having different (stronger) security parameters (e. g. the iteration count) without knowledge of the original password [16]. However, there are no PBKDFs having this property currently available.

The most widely used password-based key derivation functions are currently *PBKDF2* and *bcrypt*.

PBKDF2 was standardized under PKCS[1] #5, Version 2.0 in 1999 (also published as RFC 2898 in 2000 [9]) and later specified in NIST Special Publication 800-132 [21] as the only password-based key derivation function approved by the U.S. National Institute of Stan-

1. PKCS = Public-Key Cryptography Standards; a group of standards published by RSA Security, Inc. (see https://www.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm)

dards and Technology. PBKDF2 is widely employed in many practical applications, such as Wi-Fi Protected Access (security protocols used to secure wireless networks), disk encryption software (Cryptsetup, TrueCrypt/VeraCrypt[2], ...) and password managers (LastPass[3], 1Password[4], ...).

Bcrypt was introduced in 1999 [17] and although it incorporates several improvements over PBKDF2, it is not as widely used.

In 2009, a new password-based key derivation function *scrypt* was introduced [14], which uses the aforementioned sequential memory-hard functions.

In 2013, an open competition called *Password Hashing Competition* was announced. The competition aims to "identify new password hashing schemes in order to improve on the state-of-the-art" [2]. The competition is organized by a group of cryptography experts, not by a standardization body. It is expected, however, that it will lead to a new standard for password-based key derivation and password hashing.

2.3 PBKDF2

PBKDF2 is a generic password-based key derivation function – its definition depends on the choice of an underlying pseudorandom function (PRF). The specification [9] does not impose any additional constraints on the PRF, other than that it takes two arbitrarily long octet strings as input and outputs an octet string of a certain fixed length. In appendix B.1, the specification presents the HMAC message authentication code (HMAC; specified in RFC 2104 [11]) using SHA-1 as the underlying hash function as an example for the PRF. The practical applications of PBKDF2 use HMAC (with various underlying hash algorithms) almost solely as the underlying PRF.

The instantiations of PBKDF2 using specific PRFs are often denoted as PBKDF2-*PRF*, where *PRF* is the name of the PRF used (for example PBKDF2 using HMAC-SHA1 would be denoted as PBKDF2-HMAC-SHA1).

2. https://veracrypt.codeplex.com/
3. https://lastpass.com/
4. https://agilebits.com/onepassword

PBKDF2 takes two important security parameters – *salt* and *iteration count*.

As noted in [9, section 4.1], salt has two main purposes in password-based cryptography:

1. To make it infeasible for an attacker to precompute the results of all possible keys (or even the most likely ones), that is to prevent rainbow-table attacks (described in 2.2). For example, if the salt is 64 bits long, a single input key (password) has 2^{64} possible derived keys, depending on the choice of salt.

2. To make it unlikely that the same key will be derived twice. This is important for some encryption and authentication techniques.

The iteration count represents the number of successive computations of the underlying PRF that are required to compute each block of the derived key. The purpose of the iteration count is to increase the time cost of the function, in order to mitigate brute-force and dictionary attacks, as discussed in 2.2. The original specification recommends a minimum of 1000 iterations to be used [9, section 4.2], however there are concerns that this number may not be sufficient and that it should be determined dynamically according to current technology [4, chapter 7] [6, footnotes 7, 8].

2.3.1 Description of the algorithm

Let *hLen* be the length in octets of the output of the pseudorandom function PRF. PBKDF2 with PRF as the underlying function takes the following input parameters [9]:

- P – the password (an octet string)
- S – the salt (an octet string)
- c – the iteration count (a positive integer)
- *dkLen* – the intended length in octets of the derived key (a positive integer, at most $(2^{32} - 1) \cdot hLen$)

The following pseudocode illustrates the process of computation of PBKDF2 with underlying function PRF, as per RFC 2898 [9]:

Algorithm 1 PBKDF2

function PBKDF2($P, S, c, dkLen$)
  if $dkLen \leq (2^{32} - 1) \cdot hLen$ **then**
    output "derived key too long" and stop
  end if
  $l \leftarrow \lceil dkLen / hLen \rceil$ ▷ l is the number of $hLen$-octet blocks in the derived key, rounding up
  $r \leftarrow dkLen - (l - 1) \cdot hLen$ ▷ r is the number of octets in the last block
  for $k \leftarrow 1, l$ **do**
    $T_k \leftarrow \mathrm{PRF}(P, S \mid \mathrm{int}(k))$
    for $i \leftarrow 2, c$ **do**
      $T_k \leftarrow T_k \oplus \mathrm{PRF}(P, T_k)$
    end for
  end for
  return $T_1 \mid T_2 \mid ... \mid T_l[0..r - 1]$
end function

Here, $A \mid B$ is the concatenation of octet strings A and B; $\mathrm{int}(x)$ is a four-octet encoding of the integer x with the most significant octet first (i. e. the big-endian encoding of the integer x); $A \oplus B$ is the bitwise exclusive disjunction (also called "exclusive or" or "XOR") of octet strings A and B; $A[i..k]$ denotes an octet string produced by taking the i-th through the k-th octet of the octet string A.

As follows from the algorithm, the derived key is divided into blocks of $hLen$ octets, each of which can be computed independently. Each of these output blocks is computed using the same algorithm, seeded by its one-based index, which allows for the blocks to be computed in parallel.

The computation of each block is performed by applying c iterations of the PRF to an initial seed consisting of the salt and a binary representation of the index of the block. After each iteration, the output from the PRF is combined with the result of the previous iteration using the bitwise XOR operation, in order to "reduce concerns about the recursion degenerating into a small set of values" [9, section 5.2].

2.4 Scrypt

The scrypt key derivation function takes the following input parameters [14, chapter 7]:

- P – the password (an octet string)
- S – the salt (an octet string)
- N – the CPU/memory cost parameter
- r – the block size parameter
- p – the "parallelization" parameter
- $dkLen$ – the intended length in octets of the derived key

The P, S and $dkLen$ parameters have the same meaning as in PBKDF2 described in the previous section (2.3). The remaining parameters can be tuned by the user according to the amount of computing power and memory available.

Increasing (decreasing) the N parameter linearly increases (decreases) both the amount of memory (space complexity) and the amount of computation power required to compute the KDF (time complexity), if the implementation makes full use of random access memory. Alternatively, the implementation may choose to use constant amount of memory, in which case the time complexity becomes $O(N^2)$ instead of $O(N)$ [14, chapter 5, proof of theorem 1]. This allows for a time-memory trade-off, which can be exploited to gain better performance on GPUs [15, 16].

Increasing (decreasing) the r parameter also linearly (increases) time and space complexity, but does not allow for a similar time-memory trade-off. However, theoriginal scrypt paper recommends using only a small, relatively fixed value for r (8) and suggests instead increasing the N parameter (in the password cracking time estimates [14, chapter 8] Percival uses values $N = 2^{14}$ or 2^{20} and $r = 8$).

The p parameter also has linear scaling effect on time and space complexity, but allows up to p parallel processes/computation units to be used for computation (with the exception of the fast initial and final steps). The value for this parameter recommended by the scrypt

specification is 1. However, the author notes that future advancement in technology might lead to higher values being more efficient [14, chapter 7].

The time-memory trade-off problem was addressed in scrypt's successor, *yescrypt*, which is one of the finalists of the Password Hashing Competition (see page 7).

As opposed to PBKDF2, the pseudorandom function and hash function used internally in scrypt are strictly defined in the specification.

3 The GPU architecture

Originally designed for acceleration of computer graphics, GPUs have recently also become a powerful platform for general-purpose parallel processing. The fast-growing computer game industry has motivated a rapid advancement of graphics hardware, which is gradually outperforming general-purpose CPUs by several orders of magnitude.

To be able to optimally use the computational potential of GPUs, the task being computed must have certain properties. The task must be divisible into multiple small tasks that all perform the same computation over different pieces of data. Each of these tasks must only require a very small amount of memory.

A standard CPU consists of a single and relatively complex *control unit* (CU)[1] and one or more *arithmetic logic units* (ALU)[2]. Between the CPU and main memory there are several layers of cache – a small and fast memory that stores recently accessed contents of the main memory for faster access.

Note: the GPU architecture described in this chapter mostly follows the Compute Unified Device Architecture[3] (CUDA). The architecture of GPUs that do not support CUDA might differ.

A GPU consists of the *global memory* (also called *device memory*) and multiple *streaming multiprocessors*. Data can be transferred from the host's main memory to/from the global memory via a PCI Express (PCIe) bus or, in the case of an integrated GPU, the host's main memory is shared with the GPU, so there is no need to transfer data between the host and the GPU.

---

1. The CU is the part of the CPU that decodes a program's instructions and controls the operation of the rest of the CPU, the computer's memory and the input/output devices based on these instructions [23].
2. The ALU "performs arithmetic and bitwise logical operations on integer binary numbers" [22].
3. see http://www.nvidia.com/object/cuda_home_new.html or https://en.wikipedia.org/wiki/CUDA for more information

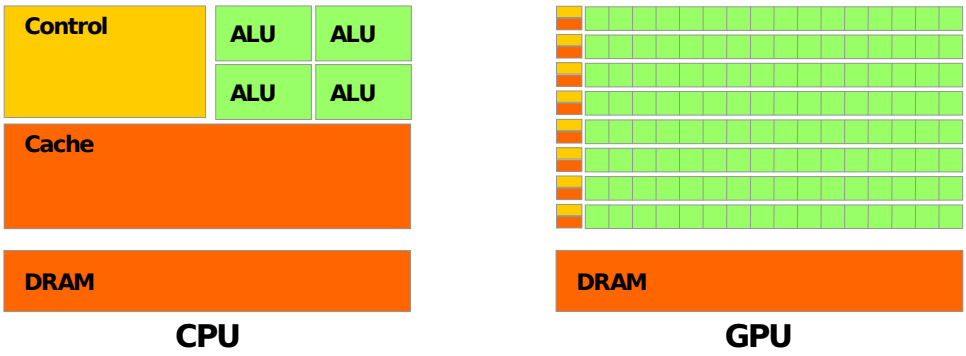

Figure 3.1: CPU vs GPU architecture[4]

Streaming multiprocessors are designed for *data parallelism* – each thread should ideally execute the same sequence of instructions over different data. The workload given to a GPU consists of a certain number of executions (*threads*) of a single program (*kernel*). Threads are divided into *blocks*, which are distributed among SMs. Blocks are further divided into smaller *warps* of threads – all threads in a warp always execute the same instruction. SM can switch between currently running warps, for example to hide memory latency while a warp waits for a memory transaction. Warp scheduling is handled by hardware automatically, which minimizes scheduling overhead [18].

Kernels may contain branches and loops – in this case both paths of each branch are executed in each thread. Branching should, however, be used carefully – if the threads within a warp diverge (that is, they end up executing different parts of the kernel) the computation becomes inefficient. As opposed to CPUs, SMs in GPUs always execute instructions in-order instead of out-of-order. Memory latency is avoided by temporarily switching to another warp [18].

Each streaming multiprocessor contains a single *instruction unit* (similar to the CPU's control unit) with instruction cache, *constant cache* (a cache for the part of global memory which contains con-

4. Copyright © 2010, NVIDIA Corporation, published under a Creative Commons Attribution 3.0 Unported License; source: https://commons.wikimedia.org/w/index.php?title=File:Cpu-gpu.svg&oldid=156649300

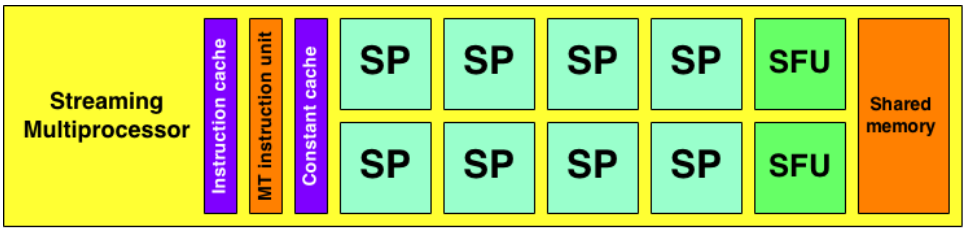

Figure 3.2: Streaming multiprocessor

stant data), *shared memory*, several *thread processors* (denoted SP) and several *special function units* (SFU) [18]. SPs execute arithmetic instructions, SFUs are used for special mathematical functions, such as sine, cosine or logarithm [18].

3.1 Memory hierarchy

3.1.1 Global memory

The main memory of a GPU is the *global memory*. Global memory is a dynamic random-access memory (DRAM), which is the type of memory that is also typically used for a computer's main memory. Accessing global memory from a thread has a relatively high latency (as opposed to registers), because it is not cached (except for the part that is reserved for constant data). The size of global memory ranges from hundreds of megabytes to several gigabytes [25].

3.1.2 Constant memory

Part of global memory is used as the so-called *constant memory*. To the threads running on the GPU the constant memory is read-only. Since constant memory is cached, accessing it from a thread has relatively low latency (unless there is a cache miss, it can be as fast as the registers). Usage of constant memory is optimal when all threads in a warp are reading the same memory location at once [1, section 5.3.2., Constant memory]. The size of constant memory in CUDA is fixed to 64 KB with 8-10 KB of cache for each streaming multiprocessor [1, appendix G.1].

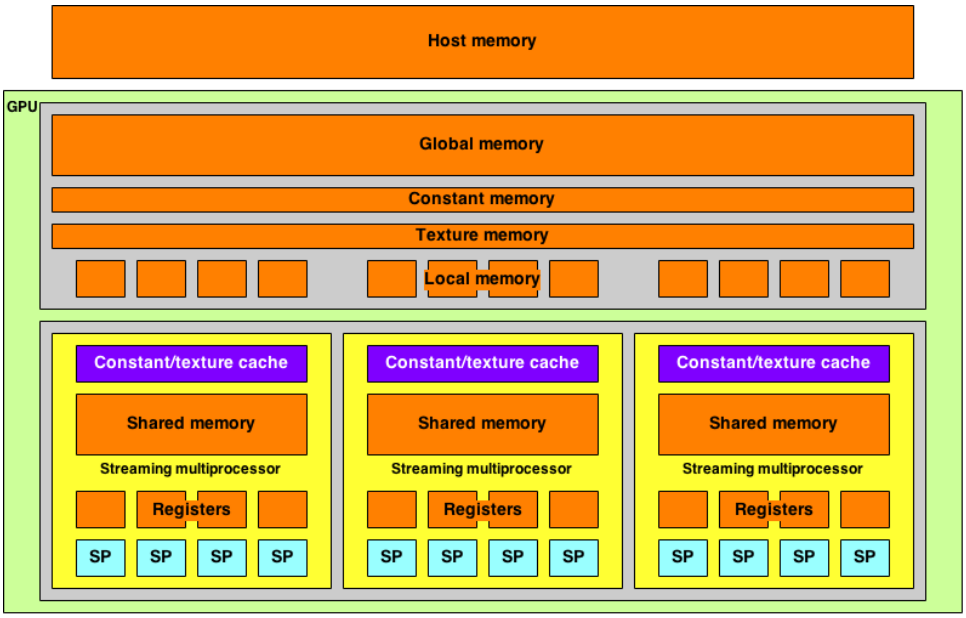

Figure 3.3: GPU memory hierarchy

3.1.3 Texture memory

Another part of global memory is used as *texture memory*. Like constant memory, texture memory is also cached. Texture memory is, as opposed to constant memory, optimized for two-dimensional spacial locality. That is, the data in the memory is interpreted as a two-dimensional array of values and accesses to the memory are optimal when threads in the same warp access memory locations that correspond to positions in the array that are close to each other [1, section 5.3.2., Texture memory].

3.1.4 Local memory

The GPU driver also allocates a block of global memory as the *local memory* for each thread (if necessary). Local memory is used for thread-local data that does not fit into the registers in the streaming multiprocessor [1, section 5.3.2., Local memory]. Since accesses to the local memory have a high latency, the programmer should keep the

per-thread data as small as possible, so that only SM registers are used.

3.1.5 Shared memory

Each streaming multiprocessor contains an on-chip *shared memory* which is shared between threads in the same block. The shared memory is distributed among blocks that are being executed by the SM. The size of the shared memory is typically about 16 KB. Shared memory has higher throughput and lower latency than global memory [1, section 5.3.2., Shared memory].

Shared memory is divided into modules of the same size called *banks*. When threads each access a different bank, the data transfer can be performed in parallel. When two or more threads access the same bank, the access has to be serialized [1, section 5.3.2., Shared memory].

3.1.6 Registers

The fastest kind of memory available on a GPU are the registers within SPs. The registers are divided among all threads in a block – the registers for each block remain allocated until all threads in the block finish execution, which allows for fast warp switching.

4 Implementing a brute-force attack on PBKDF2 on GPUs

In this chapter we discuss the feasibility of an efficient brute-force attack on the PBKDF2 key derivation function. We focus on the PBKDF2-HMAC family of functions[1] as these are most frequently used in real-world applications (see section 2.3).

4.1 Notes on implementing PBKDF2-HMAC

Let H be a hash function that uses an internal state of s octets, operates on input blocks of b octets and produces output of l octets ($l \leq s$, $l \leq b$). We shall assume that the hash function is defined by the following parameters:

- H_I – the initial state (an octet string of length s)
- H_U – the "update" function, which takes the previous state (an octet string of length s) and an input block (an octet string of length b) and returns the new state (an octet string of length s)
- H_F – the "finalize" function, which takes the previous state (an octet string of length s), the last (possibly incomplete) input block (an octet string of length at most b) and the total length of the input string (a non-negative integer) and returns the final hash (an octet string of length l)

We further assume that computing the hash function H over an input octet string D of length n proceeds as follows:

1: **function** $H(D, n)$
2: $S \leftarrow H_I$
3: $c \leftarrow \lceil n/b \rceil - 1$
4: **for** $i \leftarrow 1, c$ **do**
5: $S \leftarrow H_U(S, D[ib : (i+1)b - 1])$
6: **end for**

1. by PBKDF2-HMAC we denote the family of functions that use HMAC-*hash* (for any *hash*) as their underlying PRF

7: **return** $H_F(S, D[cb : n-1], n)$
8: **end function**

Here, $A[i..k]$ denotes an octet string produced by taking the i-th through the k-th octet of the octet string A.

It can be shown that any hash function that uses the Merkle-Damgård construction (see [13, p. 333]; this includes all of the commonly used hash functions – SHA-1, SHA-2, SHA-3, MD4, MD5, RIPEMD, Whirlpool) conforms to this definition.

Following the above definition and the HMAC specification ([11]) the pseudocode from section 2.3 can be rewritten for PBKDF2-HMAC as follows:

1: **function** PBKDF2-HMAC($P, S, c, dkLen$)
2: **if** $dkLen \leq (2^{32} - 1) \cdot hLen$ **then**
3: output "derived key too long" and stop
4: **end if**
5: $l \leftarrow \lceil dkLen / hLen \rceil$ ▷ l is the number of $hLen$-octet blocks in the derived key, rounding up
6: $r \leftarrow dkLen - (l-1) \cdot hLen$ ▷ r is the number of octets in the last block
7: ▷ Pre-hash the password if necessary and pad it with zeroes:
8: **if** $|P| > b$ **then**
9: $K \leftarrow H(P) \,|\, \text{repeat}(\text{0x00}, b - l)$
10: **else**
11: $K \leftarrow P \,|\, \text{repeat}(\text{0x00}, b - |P|)$
12: **end if**
13: ▷ Setup "ipad" and "opad" partial hash states:
14: $S_{IPAD} \leftarrow H_U(H_I, K \oplus \text{repeat}(\text{0x36}, b))$
15: $S_{OPAD} \leftarrow H_U(H_I, K \oplus \text{repeat}(\text{0x5C}, b))$
16: **for** $k \leftarrow 1, l$ **do**
17: ▷ The first iteration:
18: $A \leftarrow S \mid \text{int}(k)$
19: $c_A \leftarrow \lfloor |A| / b \rfloor$
20: $S_{SALT} \leftarrow S_{IPAD}$
21: **for** $i \leftarrow 1, c_A$ **do**
22: $S_{SALT} \leftarrow H_U(S_{SALT}, A[ib : (i+1)b - 1])$
23: **end for**

24: $T_k \leftarrow H_F(S_{OPAD}, H_F(S_{SALT}, A[c_A b : |A| - 1]))$
25: ▷ The remaining iterations:
26: **for** $i \leftarrow 2, c$ **do**
27: $D \leftarrow H_F(S_{IPAD}, T_k)$
28: $D \leftarrow H_F(S_{OPAD}, D)$
29: $T_k \leftarrow T_k \oplus D$
30: **end for**
31: **end for**
32: **return** $T_1 \mid T_2 \mid ... \mid T_l[0..r-1]$
33: **end function**

Here, $|A|$ denotes the length of octet string A; repeat(x, n) denotes an octet string produced by repeating octet x n times; 0xXY denotes an octet in hexadecimal representation.

When trying to implement PBKDF2 efficiently, the most important part for optimization is the main loop from lines 26-30, as this is the only part time complexity of which depends on the number of iterations. When HMAC is used as the PRF in PBKDF2, this part is very simple – it iteratively performs two hashing operations over the result of the previous iteration and then combines the result with the result of the previous iteration. Therefore, assuming a reasonably high iteration count, in order to optimize PBKDF2-HMAC it is sufficient to optimize the implementation of the underlying hash function.

On parallel computing platforms such as GPUs, it is also important that the computation consumes as little memory as possible. When PBKDF2 is used with HMAC as the PRF, the memory requirements of the core of the algorithm (the main loop from the previous paragraph) do not depend on the security parameters (iteration count, salt length), but only depend on the hash function used. For example, with SHA-1 used as the hash, the core of the algorithm requires approximately 164 bytes (plus few more bytes might be required for temporary variables); with SHA-256 it requires approximately 224 bytes and with SHA-512 approximately 448 bytes.

4.2 Running PBKDF2 on a GPU

PBKDF2-HMAC, when used with common hash functions, has several properties that make it possible to implement it (for bulk processing) very efficiently on GPU hardware. This poses a security/usability problem for applications that cannot utilize the GPU for evaluating PBKDF2 (this is true for almost all practical applications, with only a few exceptions – e. g. busy user authentication servers). The great difference in performance between the hardware available to the user and the hardware available to the attacker means that in order to provide reasonable security, the PBKDF2 iteration count must be set according to the attacker's potential hardware capabilities, which in turn causes an excessive inconvenience to the user, who then has to wait a long time for the password to be processed.

As with most cryptographic algorithms, an efficient implementation PBKDF-HMAC does not require any data-dependent branching (executing different code on different inputs) which avoids divergence of different GPU threads (see chapter 3). This is, however, a desirable security property that helps prevent timing attacks, where an attacker attempts to gain information about a secret parameter by measuring and analyzing the time it takes to perform a certain cryptographic function.

Another important property of PBKDF2-HMAC is that it has constant and very low memory requirements (as discussed in the previous section). This allows the GPU implementation to run more threads while keeping most (or all) of the data in the registers, thus avoiding expensive accesses to the global memory. As discussed on page 6, new password-based key derivation functions (such as scrypt) address this problem by using so called *sequential memory-hard functions*.

An interesting property of PBKDF2-HMAC is also the fact that each hash-output-sized block of the derived key can be computed independently. This allows the GPU implementation to compute each block in a separate thread, which means that when deriving longer keys, fewer PBKDF2 tasks are sufficient to saturate all cores of the

GPU. This, however, does not provide a significant advantage for brute-force/dictionary attacks, as the attacker has a lot of tasks to process and can always submit more of them at once.

5 The demonstration program

To demonstrate how GPUs can be used to accelerate a brute-force or dictionary attack on a real-world system that uses PBKDF2 for password-based key derivation, we implemented a simple demonstration program in the form of a command-line application that performs an offline attack on the access passwords of a LUKS partition.

The source code of the demonstration program can be found in the source code archive included in the thesis repository, under the `lukscrack-gpu` directory. The source code is also accessible online as a GitHub repository[1].

5.1 LUKS

LUKS (Linux Unified Key Setup) is a standard for key setup[2] in disk encryption [6]. It was originally developed for the purpose of standardizing and simplifying the process of key management for disk encryption on the Linux operating system [5, chapter 6]. Nevertheless, the standard itself is platform independent and can be implemented on any platform.

LUKS defines a binary format for storing keys and encrypted data on disk partitions, as well as basic operations on partitions which use this format (partition initialization, adding, changing and revoking passwords, etc.) [6].

A LUKS partition begins with a *partition header*, followed by eight sections of encrypted key data (called *key material*), which is then followed by the encrypted *payload* (the contents of the partition).

The payload of a LUKS partition is encrypted by a *master key* (MK), which is randomly generated when initializing the partition and does not change during the lifetime of the partition.

The partition header contains eight *key slots*, each of which can be active or disabled. Each active key slot is associated with one of the key material sections. Each key material section contains the

1. https://github.com/WOnder93/pbkdf2-gpu
2. key setup = the management of encryption keys used in a cryptosystem

master key processed by a transformation called *AFsplit* (explained below) and encrypted with a key derived from the *user password*. Each key slot contains parameters that specify how to obtain the master key from the associated key material and the corresponding user password. This means that the user needs to know the correct password of at least one key slot in order to decrypt the whole partition.

A more detailed description of the LUKS partition format can be found in the LUKS On-Disk Format Specification [6, section 2.4].

5.1.1 AFsplit

The master key is usually only 16 or 32 bytes long [6, chapter 1]. Therefore when stored on a hard disk device, it is likely to end up in a single physical disk sector. When a disk sector gets damaged or corrupted, the disk firmware may silently remap it another spare sector and the original sector becomes inaccessible [6, chapter 1]. Even though the remapped sector becomes inaccessible to the software and is likely unreadable, an advanced forensic analysis might still be able to recover all or part of the data stored in the sector.

Suppose an encrypted master key was stored in a single sector, which was later remapped to another sector. If the key used to encrypt the MK was then revoked, the disk encryption software, not knowing that the remapping occurred, would only erase the new sector, while the data might still be readable from the original one.

To counter this problem, LUKS defines a transformation called *AFsplit* (the inverse transformation is called *AFmerge*), which transforms a key (any octet string) into an octet string, size of which is an arbitrary multiple of the key size, such that the master key can only be recovered from the whole string.

A brief definition of the AFsplit/AFmerge transformation can be found in [6, section 2.4]. A more detailed description and rationale is provided by Fruhwirth in [5, section 5.2].

5.1.2 Verifying LUKS partition passwords

To implement a brute-force attack on a LUKS partition, it is essential to understand the process of verifying whether a password is valid for a given active key slot. In order to verify a key slot password, the

following fields of the partition header are needed (field names are as per [6]):

- *cipher-name*, *cipher-mode* – text identifiers of the cipher (encryption algorithm) that is used to encrypt the key material (as well as to encrypt the payload)
- *hash-spec* – a text identifier of the hash function used for key derivation, the AFsplit/AFmerge transformation and for computing the master key digest (see below)
- *key-bytes* – the length in bytes of the key used with the cipher specified by *cipher-name* and *cipher-mode*
- *mk-digest* – the master key digest, which is computed from the MK using PBKDF2-HMAC with the hash function specified by *hash-spec*
- *mk-digest-salt*, *mk-digest-iter* – the PBKDF2 salt and iteration count used for computing the MK digest

In addition to this information, the following fields of the key slot are needed:

- *iterations*, *salt* – the PBKDF2 iteration count and salt used for deriving the key material encryption key from the user password
- *key-material-offset* – the offset in sectors[3] of the key material associated with the key slot
- *stripes* – the factor by which the MK is expanded by the AFsplit transformation

Using the information above, the key slot password verification proceeds as follows:

1. The key material sectors are read from the partition. The key material starts at sector specified by *key-material-offset* and takes up as many sectors as needed to store *key-bytes* times *stripes* bytes.

3. a sector in LUKS is 512 bytes long (this fact is omitted in version 1.2.1 of the LUKS On-Disk Format Specification)

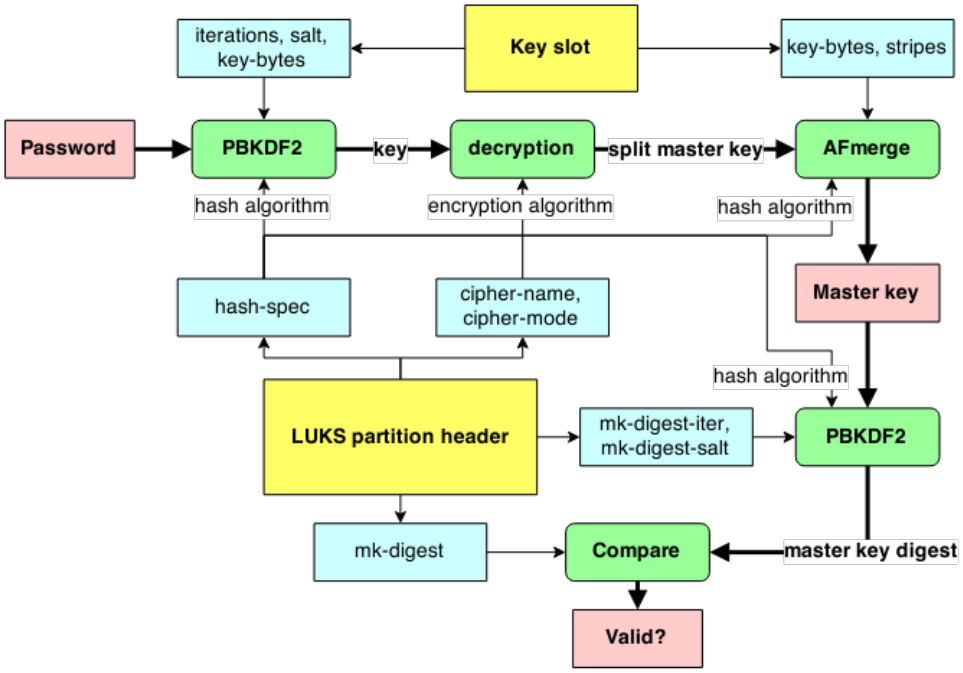

Figure 5.1: LUKS partition password verification

2. A key of *key-bytes* bytes is derived from the password using PBKDF2-HMAC with the hash function specified by *hash-spec*, *iterations* as the iteration count and *salt* as the salt.

3. The derived key from the previous step is used to decrypt the key material sectors. The first *key-bytes* times *stripes* bytes of the decrypted data are taken as the split master key. The trailing bytes (if any) are ignored.

4. The master key candidate is obtained from the split master key by applying the AFmerge transformation with the hash function specified by *hash-spec*, with *key-bytes* as the key length and with *stripes* as the expansion factor.

5. Finally, a 20-byte digest of the master key candidate is produced by applying PBKDF2-HMAC with the hash function specified by *hash-spec*, *mk-digest-iter* as the iteration count

and *mk-digest-salt* as the salt. The computed digest is then compared to *mk-digest* – if the digests match, the password is valid, otherwise it is not.

5.2 Implementation

As shown in the previous section, verifying a LUKS key slot password requires two computations of PBKDF2, between which a different computation needs to be performed (key material decryption and AFmerge).

For a typical LUKS partition created by cryptsetup the most computationally difficult is the first PBKDF2 instance (deriving encryption key from the password). The second PBKDF2 instance (master key digest computation) is about 8 times less computationally difficult and the difficulty of the rest of the computation is usually negligible. In our demonstration program both PBKDF2 instances are computed on the GPU, while the rest of the computation is performed on the CPU.

5.2.1 Optimal utilization of hardware resources

In order to employ all cores of the GPU, our demonstration program processes the candidate passwords in *batches*. The state of processing a password batch is managed by an object which we call the *batch processing context* (BPC). The BPC can be thought of as a state machine with four states (excluding the initial and final state) where the transitions between states represent separate stages of computation. Figure 5.2 shows a state diagram of the BPC.

Since most of the computation is performed on the GPU, it is desirable that it is kept fully utilized throughout the attack. Also, in case the GPU used was so powerful that the CPU computation takes longer than the GPU computation, it would be preferable if the CPU was kept fully utilized.

For this reason, we devised a simple scheduling algorithm which coordinates execution of three concurrent BPCs in a way that ensures

Algorithm 2 Batch processing context scheduling

```
1: procedure CPUPHASE1(bpc)
2:     ENDMKDIGESTCOMPUTATION(bpc)
3:     PROCESSDIGESTS(bpc)
4:     INITPASSWORDS(bpc)
5:     BEGINKEYDERIVATION(bpc)
6: end procedure
7:
8: procedure CPUPHASE2(bpc)
9:     ENDKEYDERIVATION(bpc)
10:    DECRYPTKEYMATERIAL(bpc)
11:    BEGINMKDIGESTCOMPUTATION(bpc)
12: end procedure
13:
14: procedure PROCESSPASSWORDS(bpc1, bpc2, bpc3)
15:    (run a partial version of the body of the loop below to initialize the BPCs to the loop's invariant)
16:    while (there are passwords to process) do
17:        CPUPHASE1(bpc1)
18:        CPUPHASE2(bpc3)
19:        CPUPHASE1(bpc2)
20:        CPUPHASE2(bpc1)
21:        CPUPHASE1(bpc3)
22:        CPUPHASE2(bpc2)
23:    end while
24:    (run a partial version of the body of the loop above to finalize the BPCs from the loop's invariant)
25: end procedure
```

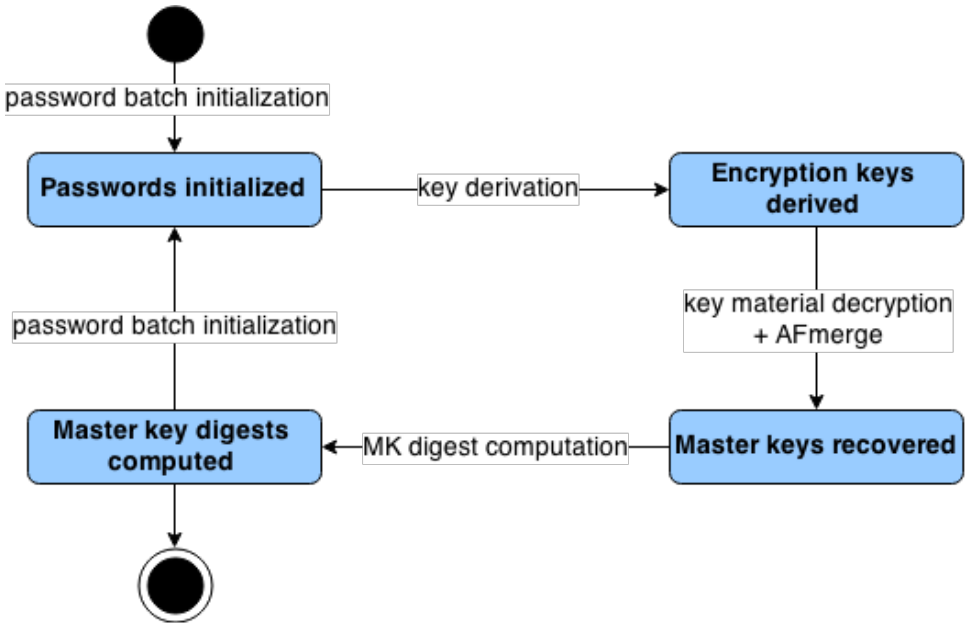

Figure 5.2: Batch processing context – a state diagram

that the computing hardware is optimally utilized (as described in the previous paragraph).

The algorithm is informally described in the pseudocode of the PROCESSPASSWORDS procedure presented in algorithm 2. An explanation of the procedures used in algorithm 2 follows:

- INITPASSWORDS(*bpc*) – initializes *bpc* with a new password batch.
- BEGINKEYDERIVATION(*bpc*) – submits the PBKDF2 task to derive the keys for key material decryption from the passwords to the GPU.
- ENDKEYDERIVATION(*bpc*) – waits for the task submitted by BEGINKEYDERIVATION(*bpc*) to end.
- DECRYPTKEYMATERIAL(*bpc*) – decrypts the key material using the derived keys, then obtains MK candidates from the de-

crypted versions of the key material using the AFmerge transformation.

- BEGINMKDIGESTCOMPUTATION(*bpc*) – submits the PBKDF2 task to compute digests of the MK candidates to the GPU (for the BPC *bpc*).
- ENDMKDIGESTCOMPUTATION(*bpc*) – waits for the task submitted by BEGINMKDIGESTCOMPUTATION(*bpc*) to end.
- PROCESSDIGESTS(*bpc*) – compares the computed MK candidate digests with the MK digest from the partition header. If a matching digest is found, reports that a valid password was found and stops the processing.

The GPU tasks are executed one at a time in the order in which they have been submitted.

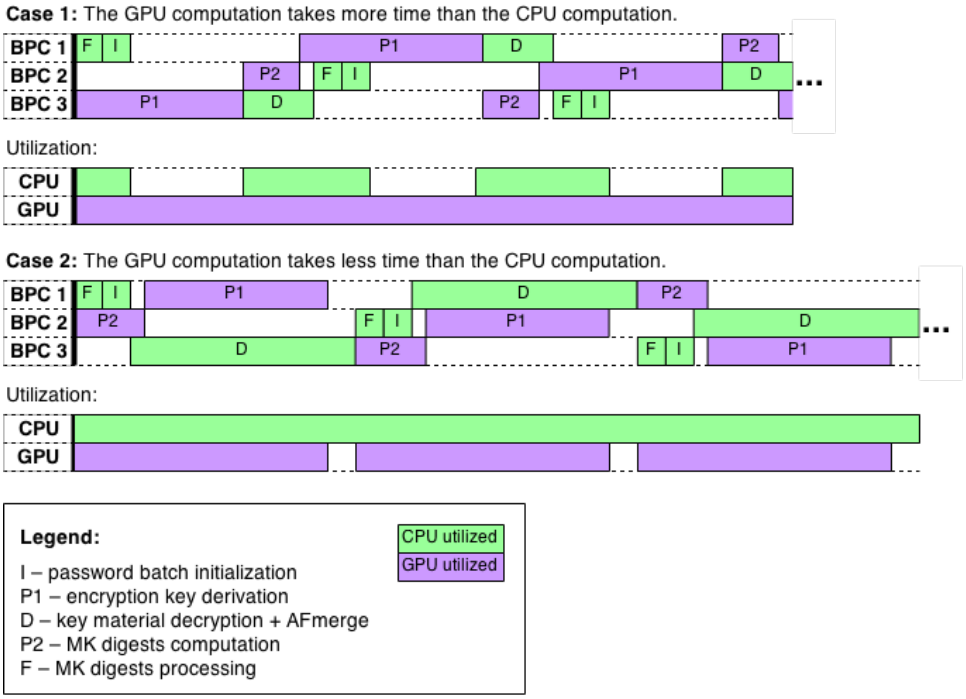

Figure 5.3: The scheduling algorithm – resource utilization

Figure 5.3 shows how our algorithm achieves optimal CPU/GPU utilization in both cases discussed earlier in this section.

5.2.2 Using multiple CPU threads or GPUs

The demonstration program allows the user to specify multiple GPUs to use for computation of PBKDF2, as well as the number of CPU threads to use for the key material decryption and AFmerge transformation phase.

When the user specifies multiple GPUs to be used, the program runs processing on each GPU separately. That is, each GPU is controlled from a separate CPU thread, which executes a separate set of interleaved batch processing contexts as described in the previous subsection.

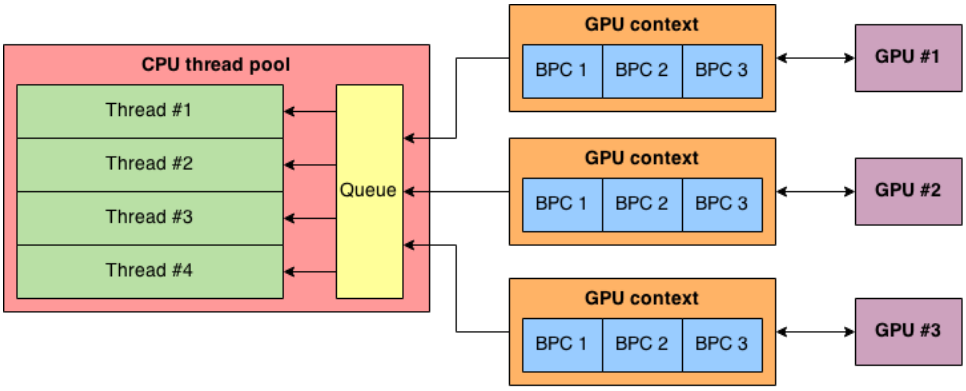

Figure 5.4: The overall architecture of the demonstration program

When the user specifies that more than one CPU thread should be used, the program allocates the requested number of threads which share a synchronized queue from which they pull the tasks to execute. All GPU controlling threads have a reference to the queue and push tasks into it. When a GPU controlling thread needs to perform key material decryption and AFmerge transformation over a batch

of inputs, the inputs are divided into as many groups of approximately equal size as there are CPU threads allocated. For each group a separate task that processes the inputs in the group is pushed to the queue. The GPU controlling thread then waits for all tasks it has pushed to the queue to complete.

Figure 5.4 shows an illustration of the overall architecture as described in this subsection.

If the user specifies that only one CPU thread should be used, then no extra threads are allocated and all CPU computation is performed directly in each GPU controlling thread.

6 Comparison of CPU and GPU attack speeds

In order to compare the potential speed of a brute-force/dictionary attack on the PBKDF2 key derivation function, we wrote a GPU implementation of PBKDF2-HMAC-SHA1 and ran benchmarks on both the GPU implementation and a CPU reference implementation on various devices.

The source code of the benchmarking program as well as the raw measurement data can be found in the source code archive included in the thesis repository, under the `benchmarking-tool` directory. The source code and data is also accessible online as a GitHub repository[1].

6.1 Implementation and methodology

In the GPU implementation, we used the OpenCL API for submitting work to the GPU.

In order to utilize all cores of the GPU and therefore maximize the performance, our implementation processes passwords in large fixed-size batches. The optimal password batch size varies between devices and can be specified by the user.

The implementation works as follows:

1. A new password batch is initialized on the host CPU (in the benchmarking program the passwords are initialized to fixed-length pseudorandom strings). Each password is pre-hashed and padded with zeroes if necessary as per the HMAC specification [11]. The resulting fixed-size blocks are transferred to the global memory on the GPU.

   The initialization and memory transfer are not included in the benchmark, since in a practical attack they can be performed while the GPU is processing another password batch, and thus have no effect on the attack speed.

2. Next, the work is submitted to the GPU. One GPU thread is assigned for each derived key block of each password (see section

1. https://github.com/W0nder93/pbkdf2-gpu

4.2).

3. The CPU thread waits for the GPU to finish processing the password batch and the difference between the time the work was submitted and the time the waiting finished is taken as the result of the benchmark.

The CPU implementation uses the API of the OpenSSL "crypto" library[2], specifically the PKCS5_PBKDF2_HMAC function. The benchmarks were run with the LibreSSL library[3] due to problems when building the OpenSSL library on the system where the benchmarks were run.

Due to the variance in running times, the benchmarks were run several times for each data point and the arithmetic mean was taken as the final value. For GPU benchmarks the number of samples was 10, for CPU benchmarks it was 20 (the variance in running times for the GPU implementation was smaller than for the CPU implementation).

6.2 Results

The results of the benchmarks show that a brute-force attack on PBKDF2-HMAC-SHA1 can be performed very efficiently on GPU hardware. The GPU implementation significantly outperforms the reference CPU implementation both in terms of attack speed per single device and in terms of power efficiency.

Figure 6.1 shows a comparison of attack speeds for various devices. The attack speed is measured in PBKDF2 block-iterations per second (PBIPS) – that is the number of PBKDF2 instances (i. e. the number of passwords processed) times the number of blocks of the derived key computed independently times the iteration count over the time it took to process these passwords. We use this unit because the amount of computation needed to perform one PBKDF2 instance is proportional to the iteration count and to the smallest number of hash-output-sized blocks such that they can fit the derived key.

The default parameters used in the benchmark were:

2. https://www.openssl.org/docs/
3. https://github.com/libressl-portable/portable

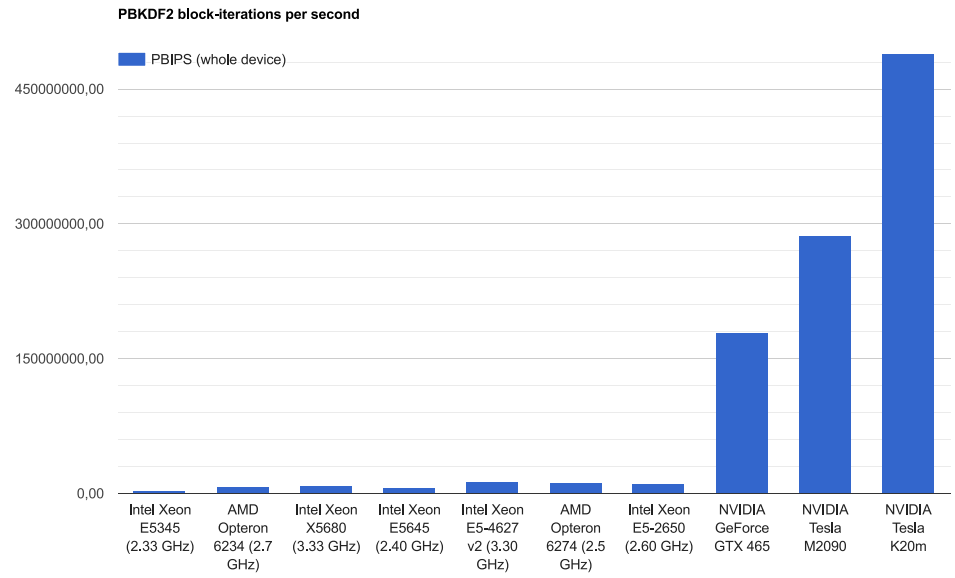

Figure 6.1: PBKDF2-HMAC-SHA1 attack speed per single device

Device type:	**Iterations:**	**Derived key length (blocks):**
CPU	4096	16 (1)
GPU	16384	16 (1)

Smaller iteration count was used for CPU benchmarks only so that they finish within a reasonable time. As the graph in figure (TODO) shows, except for extremely small values the iteration count does not influence the attack speed significantly.

The highest attack speed we measured on a CPU was 12.276 million PBIBS (on *Intel Xeon E5-4627 v2*), while the highest attack speed on a GPU was 489.437 million PBIPS (on *NVIDIA Tesla K20m*), which means the attack on a GPU was almost 40 times faster than on a (single) CPU.

For an attacker, however, an increase in attack speed per device alone is probably not going to make a significant difference. If we disregard the initial cost[4], the the total cost of the attack can be ex-

4. The initial costs can be avoided entirely by buying computing power by the hour via a service such as Amazon EC2 (https://aws.amazon.com/ec2/).

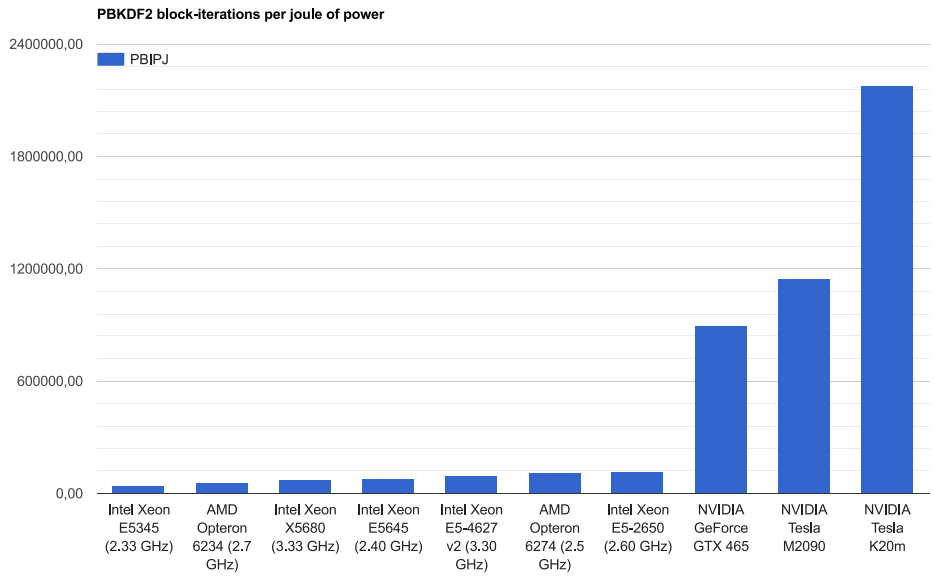

Figure 6.2: PBKDF2-HMAC-SHA1 attack speed over TDP

pressed as the product of the attack speed (in work items per second), power consumption at full speed and the price of electricity divided by the number of work items to compute. Therefore, a more meaningful metric would be the attack speed (on a given device) divided by power consumption (of the device while performing the attack).

In figure 6.2 the devices are compared by attack speed divided by their thermal design power (TDP). The TDP of a device is defined as the maximum amount of heat generated by the device in typical operation [20]. TDP is generally used when designing CPU/GPU cooling systems, however since almost all of the power drawn by a CPU/GPU eventually converts to heat, it is an acceptable approximation of the actual power consumption.

The highest attack power efficiency we measured on a CPU was 112.385 thousand PBIPJ[5] (on *Intel Xeon E5-2650*) and the highest attack power efficiency on a GPU was 2 175.276 PBIPJ (on *NVIDIA Tesla K20m*). This means that the power efficiency of PBKDF2-

5. PBIPJ = PBKDF2 block-iterations per joule – equivalent to PBIPS per watt

HMAC-SHA1 processing on a GPU was about 19 times higher than on a CPU.

It should be noted that a GPU cannot operate without at least one CPU, and therefore the actual gain from using a GPU for an attack on PBKDF2 might be lower.

7 Conclusion

A Additional benchmark graphs

Bibliography

[1] CUDA C Programming Guide, version 7.0. NVIDIA Corporation, March 2015. [Online, accessed 6-May-2015, retrieved from http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf].

[2] Password Hashing Competition, 2015. [Online, accessed 30-April-2015, retrieved from https://password-hashing.net/].

[3] CHEN, L. Recommendation for key derivation using pseudorandom functions. NIST Special Publication 800-108, The U.S. National Institute of Standards and Technology, November 2008. [Available at http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf].

[4] DÜRMUTH, M., GÜNEYSU, T., KASPER, M., PAAR, C., YALCIN, T., AND ZIMMERMANN, R. Evaluation of standardized password-based key derivation against parallel processing platforms. In *Computer Security – ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 716–733. [Available at https://hgi.rub.de/media/crypto/veroeffentlichungen/2013/01/29/esorics_pbkdf2.pdf].

[5] FRUHWIRTH, C. New methods in hard disk encryption. Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, July 2005. [Online, accessed 10-May-2015, retrieved from http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf].

[6] FRUHWIRTH, C., AND BROŽ, M. LUKS On-Disk Format Specification, Version 1.2.1, October 2011. [Online, accessed 27-April-2015, retrieved from https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf].

[7] GOLDREICH, O. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[8] HARRISON, O., AND WALDRON, J. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 195–209. [Available at https://www.usenix.org/legacy/event/sec08/tech/full_papers/harrison/harrison.pdf].

[9] KALISKI, B. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, RFC Editor, September 2000. [Available at https://tools.ietf.org/html/rfc2898].

[10] KRAWCZYK, H. Cryptographic extraction and key derivation: The HKDF scheme. Cryptology ePrint Archive, Report 2010/264, International Association for Cryptologic Research, 2010. [Available at https://eprint.iacr.org/2010/264].

[11] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997. [Available at https://tools.ietf.org/html/rfc2104].

[12] KRAWCZYK, H., AND ERONEN, P. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, May 2010. [Available at https://tools.ietf.org/html/rfc5869].

[13] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 1996. [Available at http://cacr.uwaterloo.ca/hac/].

[14] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions, May 2009. [Online, accessed 30-April-2015, retrieved from https://www.tarsnap.com/scrypt/scrypt.pdf].

[15] PERCIVAL, C. Re: scrypt time-memory tradeoff. scrypt@tarsnap.com (mailing list), June 2011. [Available at http://mail.tarsnap.com/scrypt/msg00029.html].

[16] PESLYAK, A., AND MARECHAL, S. Password security: past, present, future (presentation). Openwall, Inc., December 2012. [Online, accessed 30-April-2015, retrieved from http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf].

[17] PROVOS, N., AND MAZIÈRES, D. A future-adaptable password scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), ATEC '99, USENIX Association, pp. 32–32. [Available at https://www.usenix.org/legacy/publications/library/proceedings/usenix99/provos/provos.pdf].

[18] REGE, A. An Introduction to Modern GPU Architecture (presentation). NVIDIA Corporation, 2015. [Online, accessed 4-May-2015, retrieved from ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf].

[19] SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[20] SHVETS, G. Thermal Design Power (TDP), 2013. [Online, accessed 8-May-2015, retrieved from http://www.cpu-world.com/Glossary/T/Thermal_Design_Power_(TDP).html].

[21] TURAN, M. S., BARKER, E. B., BURR, W. E., AND CHEN, L. Recommendation for password-based key derivation, Part 1: Storage applications. NIST Special Publication 800-132, The U.S. National Institute of Standards and Technology, December 2010. [Available at http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf].

[22] WIKIPEDIA CONTRIBUTORS. Arithmetic logic unit — Wikipedia, the free encyclopedia. Wikimedia Foundation, Inc., 2015. [Online; accessed 4-May-2015, retrieved from https://en.wikipedia.org/w/index.php?title=Arithmetic_logic_unit&oldid=660450386].

[23] WIKIPEDIA CONTRIBUTORS. Central processing unit — Wikipedia, the free encyclopedia. Wikimedia Foundation, Inc., 2015. [Online; accessed 4-May-2015, retrieved from https://en.wikipedia.org/w/index.php?title=Central_processing_unit&oldid=660636564].

[24] WIKIPEDIA CONTRIBUTORS. Key derivation function — Wikipedia, the free encyclopedia. Wikimedia Foundation, Inc., 2015. [Online, accessed 11-April-2015, retrieved from https://en.wikipedia.org/w/index.php?title=Key_derivation_function&oldid=644734578].

[25] WIKIPEDIA CONTRIBUTORS. List of Nvidia graphics processing units — Wikipedia, the free encyclopedia. Wikimedia Foundation, Inc., 2015. [Online; accessed 6-May-2015, retrieved from https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=660980423].