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 [16, 5]. 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) [20, 2].

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 [7].

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 [4]. 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 [6], 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 an illustration 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 [20]. 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 [2].

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 [8, 10].

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*) [16] 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 [16]. 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* [16]. 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 [13].

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 [11, 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) [11].

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 [13]. 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 [7]) and later specified in NIST Special Publication 800-132 [17] 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 [14] 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 [11], 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" [1]. 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 [7] 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 [9]) 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 [7, 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 [7, 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 [3, chapter 7] [4, 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 [7]:

- 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 [7]:

Algorithm 1 PBKDF2

```
1:  function PBKDF2($P, S, c, dkLen$)
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:      for $k \leftarrow 1, l$ do
8:          $T_k \leftarrow \mathrm{PRF}(P, S \mid \mathrm{int}(k))$
9:          for $i \leftarrow 2, c$ do
10:             $T_k \leftarrow T_k \oplus \mathrm{PRF}(P, T_k)$
11:         end for
12:     end for
13:     return $T_1 \mid T_2 \mid ... \mid T_l[0..r - 1]$
14: 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" [7, section 5.2].

2.4 Scrypt

The scrypt key derivation function takes the following input parameters [11, 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 $O(N)$ [11, chapter 5, proof of theorem 1]. This allows for a time-memory trade-off, which can be exploited to gain better performance on GPUs [12, 13].

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 [11, 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 [11, 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 2.2).

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

3 Acceleration of algorithms using GPUs

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.

In the following section, we describe the architecture of GPUs in detail and discuss the considerations for implementing algorithms on these devices.

3.1 GPU architecture

A standard, general-purpose 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.

A GPU consists of several main parts – multiple *streaming multiprocessors*, the *global memory*, the *texture memory* and *constant memory*.

Each streaming multiprocessor contains a single *instruction unit* (equivalent to the CPU's control unit) with instruction cache, *constant cache* (a cache for constant memory), *shared memory* and several *thread processors*. Each thread processor is capable of executing

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 [19].
2. The ALU "performs arithmetic and bitwise logical operations on integer binary numbers" [18].

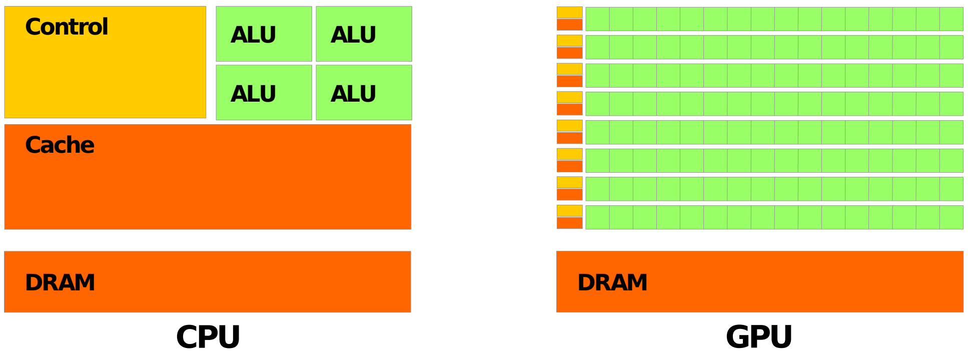

Figure 3.1: CPU vs GPU architecture[3]

several groups (called *warps*) of *threads*. All threads in a warp execute the same program, which is called a *kernel* [15].

Streaming multiprocessors are designed for *data parallelism* – threads should execute the same sequence of instructions over different data. Kernels may contain branches and loops – in this case both paths of each branch are executed in each thread. Branches should, however, be used carefully – if the threads diverge (that is, they end up executing different parts of the kernel) the computation becomes very inefficient. Thread scheduling is handled by hardware automatically, which minimizes scheduling overhead. 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 thread while a memory transaction is in progress [15].

3.1.1 Memory hierarchy

From ...

3. 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`

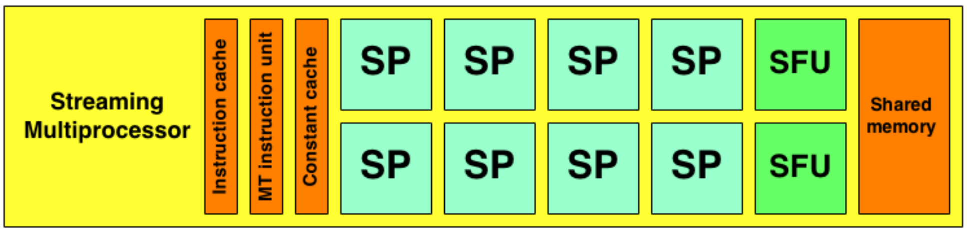

Figure 3.2: Streaming multiprocessor

4 Implementing PBKDF2 on GPUs

5 Comparison of CPU and GPU attack speeds

6 Conclusion

Bibliography

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

[2] 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].

[3] 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].

[4] Fruhwirth, C., and Brož, M. LUKS on-disk format specification, October 2011. [Online, accessed 27-April-2015, retrieved from https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf].

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

[6] 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].

[7] 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].

[8] 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].

[9] 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].

[10] 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].

[11] 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].

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

[13] 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].

[14] 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].

[15] 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].

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

[17] 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`].

[18] 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`].

[19] 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`].

[20] 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`].