Ca' Foscari
University
of Venice

Master Degree Programme
in Computer Science

Final Thesis

A Multithreaded SHA-512-Based Hashing Algorithm Using Merkle Trees

Design, Implementation, Performance Analysis, and Password-Cracking Evaluation

Supervisor
Ch. Prof. Flaminia Luccio

Supervisor
Ch. Prof. Riccardo Focardi

Graduand
Giovanni Ostanello
Matriculation Number 892631

Academic Year
2025 / 2026

Contents

Abstract

Today's hashing algorithms, such as SHA-256 or SHA-512, used to guarantee data security and integrity, are characterized by an iterative design and therefore do not take full advantage of the parallel capabilities of modern multicore processors. In this context, the objective of this thesis is to exploit server-side parallelism to increase the total amount of hashing work performed during password verification while maintaining an acceptable authentication latency. By exploiting Merkle trees, a binary-tree-based data structure that enables hash computations to be distributed across multiple CPU cores, a new hashing algorithm, `parallel_sha512`, has been developed. This algorithm divides the hashing process among different cores and combines partial results into a single final hash value. The algorithm and its variants are described through pseudocode and are implemented in the C++ programming language and in OpenCL. Their theoretical and experimental performance is evaluated against established iterative approaches. CPU and GPU implementations are compared, and custom hashcat modules and kernels are developed to assess the algorithms in password-cracking scenarios. Finally, the security implications of the Merkle-tree structure are analyzed by estimating the probability that a leaf modification leaves the final Merkle root unchanged.

Introduction

In recent decades, the importance and adoption of cryptographic techniques have grown considerably in response to the increasing amount of digital threats and the need to protect sensitive data. Cryptographic hash functions, such as SHA-256 and SHA-512, have become fundamental building blocks of modern security systems. They are used for ensuring data integrity, supporting security protocols, and constructing password-verification mechanisms [1, 2, 3]. Hash functions are also used as components of password-hashing schemes, which generally combine the password with a salt and introduce repeated computations to increase the cost of password-guessing attacks. However, many established hash functions and password-hashing constructions contain sequential dependencies that limit their ability to exploit modern multicore architectures. In particular, their structure makes them intrinsically difficult to parallelize, restricting the effective exploitation of available computational resources. This sequential behavior derives from the dependencies between successive rounds, message blocks, and intermediate states. Although such dependencies are part of the algorithmic design, they also limit the degree of parallelism available when computing a single digest.

In this context, this thesis investigates whether the parallel resources available on modern servers can be used to increase the computational cost of password verification without proportionally increasing the authentication latency experienced by legitimate users. The proposed approach is based on a Merkle-tree organization of independent SHA-512 computations, which can be distributed across multiple CPU cores [4, 5]. The primary objective is therefore not to reduce the cost of an individual SHA-512 evaluation. Instead, the aim is to perform a larger amount of hashing work during each password verification exploiting server-side parallelism to keep the elapsed verification time within an acceptable range. By increasing the work required for each password candidate, the proposed construction seeks to reduce the throughput of large-scale offline password-guessing attacks.

More specifically, the central research question addressed by this thesis is the following: given p processing units available to the defender, can the total hashing work performed for each password verification be increased by approximately a factor of p, while keeping the observed authentication latency comparable to that of a sequential construction performing only one share of that work?

Merkle trees are suitable for this purpose because independent subtrees can be evaluated concurrently. Their computation can therefore be assigned to different worker threads or processing units. Once the parallel phase has completed, the partial results are combined through a reduction phase until a single Merkle root is obtained. This root represents the final digest of the algorithm.

Starting from this intuitive approach, the thesis introduces the `parallel_sha512` algorithm and several related variants. In particular, `parallel_sha512_it` is very important because combines the Merkle tree reduction phase with an iterative hashing phase. The proposed algorithms are implemented for multicore CPUs using the C++ programming language and for GPUs through the OpenCL framework. Their performance is then evaluated theoretically and experimentally and compared with conventional iterative implementations.

The thesis also investigates how the proposed algorithms behave in a password-cracking scenario. For this purpose, the algorithms are also integrated into the hashcat framework by implementing custom CPU-side modules and OpenCL kernels [6, 7]. This integration allows to evaluate the algorithms using hashcat's candidate-generation, attack modes, and performance-measurement features. In addition to the performance analysis, the thesis considers the security implications of the Merkle tree structure by analyzing the Merkle root alteration probability when a leaf hash is modified.

Chapter 1 presents the theoretical and technical foundations required to understand the proposed algorithms. It introduces cryptographic hash functions, their most important security properties, their use in password verification, and

the principal attacks against password hashes. The chapter then describes the SHA-512 hash function, which is used as the underlying cryptographic primitive throughout this work. The choice of SHA-512 is motivated by its proven security properties and its attack and collision resistance. The `sha512crypt` password-hashing scheme is also introduced as an iterative SHA-512-based baseline for comparison with the proposed algorithms [8]. It is widely used in Unix and Linux systems and employs a more complex internal construction than raw SHA-512. The definition of Merkle trees and their properties is given. Merkle trees constitute the fundamental building block of the newly proposed parallel algorithm which allows the computational load distribution across different worker threads. OpenSSL is used to provide a common, highly efficient and optimized implementation of the SHA-512 hash function. The implementation of the algorithms on the GPU is developed using the OpenCL framework, whose execution model and main characteristics are described. Finally, hashcat, one of the most widely used password-cracking tools, is introduced. Hashcat is pivotal to this thesis because its OpenCL SHA-512 implementation was adopted to develop the GPU implementation of all the algorithms presented.

In Chapter 2, the pseudocode, the C++ and OpenCL implementations of the algorithm and its variants are presented. The chapter first introduces the intuitive idea behind the Merkle-tree-based parallelization strategy and then formalizes the algorithms through pseudocode. The C++ implementation description focuses on the practical optimization techniques adopted to provide a very efficient CPU algorithm implementation, such as data representation and workload distribution. The OpenCL implementation is described at a higher level, focusing on the mapping of the algorithm to the GPU execution model and on the key differences from the CPU C++ implementation, which are accurately discussed and motivated in the chapter. A limited number of representative code excerpts is used to describe the implementation choices.

A theoretical and experimental performance analysis is conducted in Chapter 3, comparing `parallel_sha512` and its variants with well-known iterative algorithms such as `mkpasswd` and `sha512_it`, an iterative SHA-512 implementation written in C++. In order to evaluate the behavior of the algorithms on a massively parallel architecture, the OpenCL implementation was also tested on the GPU. The CPU and GPU versions are then compared by analyzing how the execution time changes with respect to the number of leaves, cores, GPU work-items. Finally, the theoretical results are compared with the experimental results to determine whether the expected performance improvement is also reflected in practice.

Chapter 4 describes the integration of the proposed algorithms into the hashcat framework by defining both the CPU-side module and the OpenCL kernels. To better understand how the integration was performed, the chapter first examines the existing SHA-512 hash function integration. Then, it describes the custom modules and kernels implemented for `parallel_sha512` and `parallel_sha512_it`, including their hash formats, parameter parsing, kernel execution and digest comparison. The integrated modes are used to measure the number of hashes evaluated per unit of time and to analyze how the algorithm performs in a realistic password-cracking scenario.

Chapter 5 evaluates the security properties of the algorithms proposed. Starting from the Merkle tree analysis proposed in the paper [9], the chapter uses it to conduct a theoretical security evaluation of the probability that the Merkle root remains unchanged whenever one of the leaves is modified. The resulting probability values are estimated for the different parameter configurations adopted in the experimental analysis and are used to discuss the security properties of the algorithms.

In the final chapter, the results obtained from the theoretical, experimental and security analyses are summarized. The advantages and the limitations of the proposed Merkle-tree-based organization are discussed and opportunities for further development and future research are identified.

Chapter 1. Theoretical background

This chapter introduces the theoretical and technical background required to understand the design, implementation and evaluation of the algorithms presented in this thesis.

1.1 Hash functions and password hashing

A *cryptographic hash function* h is a function that, given a binary string x of arbitrary length, produces a fixed-length output $y = h(x)$, commonly called the *digest* or *hash value*. The digest is a relatively short binary string whose length depends on the hash function used. Formally, a cryptographic hash function can be defined as a mapping $h : X \to Y$, where X is a set of all possible messages and Y is a finite set of possible digests. Since typically $|X| \gg |Y|$, the function is many-to-one, and this implies that the existence of pairs of inputs with identical output is unavoidable. A pair $(x,y) \in X \times Y$ is said to be valid under a hash function h if $h(x) = y$. In cryptographic applications, it is desirable that generating a valid pair (x,y) is feasible only by choosing an input x and then computing its hash $y = h(x)$, while the opposite (finding an input corresponding to a given digest) is computationally infeasible [1, 3, 10]. For this reason, hash functions are required to satisfy the following three properties [1, 10]:

- **Preimage resistance**. Given a hash function $h : X \to Y$ and an element $y \in Y$, it should be computationally infeasible to find $x \in X$ such that $h(x) = y$.
- **Second preimage resistance**. Given a hash function $h : X \to Y$ and an element $x \in X$, it should be computationally infeasible to find $x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$. If this is achievable, also the pair $(x', h(x))$ is a valid pair.
- **Collision resistance**. Given a hash function $h : X \to Y$, it should be computationally infeasible to find $x, x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$. When such a pair exists, a *collision* is said to occur.

Common cryptographic hash functions are MD5, SHA-256, SHA-384, and SHA-512, which produce digests of length 128, 256, 384 and 512 bits respectively. Consequently, their output spaces contain 2^{128}, 2^{256}, 2^{384}, and 2^{512} possible hashes [2].

1.1.1 Password verification

Hash functions are widely used in many areas of computer science; one of the most common applications is password verification. Modern applications frequently require user authentication, and user passwords are never stored in plaintext in the application database. If an attacker were able to bypass the security mechanisms protecting the server where the passwords are stored, then they would gain immediate access to all user credentials, causing severe harm to the application's users. To prevent such security breaches, only password hashes are stored on the server or in the database. During the authentication phase, the hash of the password provided by the user is computed and compared with the hash stored on the server. If the two hashes match, the password is considered correct and access is granted. Using this security mechanism, even if an attacker obtained access to the password database, they would only see strings of pseudorandom characters. This makes the process of recovering the original passwords significantly more difficult, although not impossible, since brute-force attacks remain feasible. A *brute-force* attack consists of generating candidate passwords by trying all possible character combinations, hashing each candidate, and checking whether the resulting hash matches any stored hash. A

match reveals the corresponding user password. To mitigate brute-force attacks, two main techniques are commonly adopted [8, 11]:

- **Salting**: this technique consists of concatenating a random string (*salt*) to the password before hashing it. Even if an attacker gains access to the stored hashes, brute-force attacks become significantly harder because each password has a unique salt.
- **Iterating**: this technique consists of hashing the password once and then repeatedly hashing the resulting digest for a predefined number of iterations, each time using the previous digest as input. This procedure increases the computational cost of each password guess, slowing down brute-force attacks considerably.

1.2 The SHA family

The Secure Hash Algorithms are a family of cryptographic hash functions standardized by the National Institute of Standards and Technology (NIST) as part of the Federal Information Processing Standard (FIPS) [2]. It includes several standards that differ in security guarantees and inner design, they are:

- **SHA-0**: the original version of the 160-bit hash function published in 1993 under the name "SHA". It had significant flaws and was withdrawn shortly after publication, it was replaced by the SHA-1 standard.
- **SHA-1**: a 160-bit hash function that resembles the MD5 algorithm. It was designed by the National Security Agency (NSA). Following the discovery of practical cryptographic attacks against SHA-1, the standard was deprecated for most cryptographic applications after 2010.
- **SHA-2**: a family of two similar hash functions which differ by block and digest sizes known as SHA-256 and SHA-512. These were also designed by the NSA. SHA-2 is a family of hash functions based on two internal architectures operating on 32-bit and 64-bit words, respectively [2].
- **SHA-3**: the latest member of the SHA family of standards. It is internally different from the other members of the SHA standard. It was not developed by the NSA, it is a subset of a broader cryptographic primitive family called Keccak, designed by Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche.

1.2.1 The SHA-512 cryptographic hash function

SHA-512 is a cryptographic hash function of the SHA-2 family. It produces a 512-bit long digest from an arbitrary length input. It is widely used in security protocols and digital signatures to guarantee data integrity and authenticity. By its specifications, SHA-512 can be used to hash messages of length up to 2^{128} bits. The SHA-512 cryptographic hash function is used throughout this work as the underlying primitive for all the proposed algorithms.

SHA-512 algorithm overview

Figure 1.1 summarizes the main stages of the SHA-512 hashing process, which will be accurately described in the following sections.

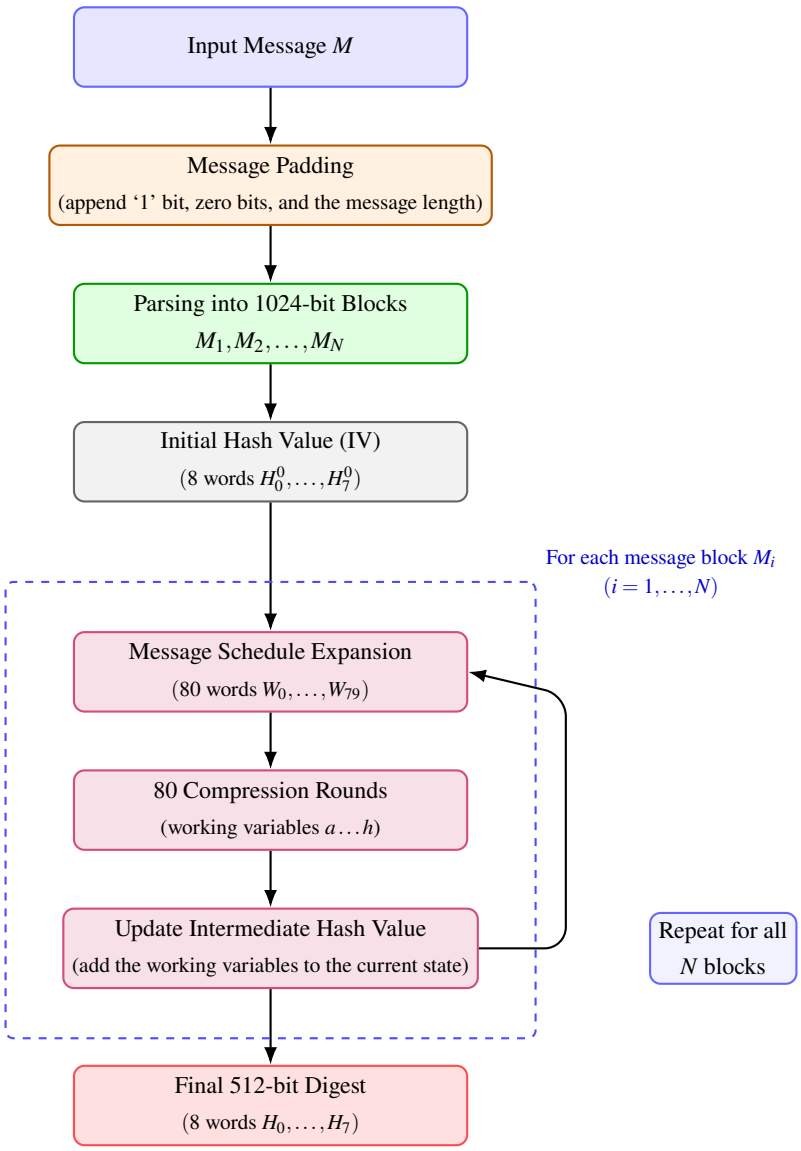


Figure 1.1: Overview of the SHA-512 hashing process, adapted from the Secure Hash Standard [2].

Symbols and operators

The following list summarizes the symbols and operators used in the Secure Hash Standard [2]. All of them operate on (w)-bit words.

- $\wedge$: bitwise AND;
- $\vee$: bitwise OR;
- $\oplus$: bitwise XOR;
- $\neg$: bitwise complement;
- $+$: addition modulo 2^w;
- $\ll$: left-shift operation with zero-padding on the right;
- $\gg$: right-shift operation with zero-padding on the left.
- $\text{ROTR}^n(x)$: rotate right (circular right shift) operation, it is defined by $\text{ROTR}^n(x) = (x \gg n) \vee (x \ll (w-n))$;
- $\text{ROTL}^n(x)$: rotate left (circular left shift) operation, it is defined by $\text{ROTL}^n(x) = (x \ll n) \vee (x \gg (w-n))$;
- $\text{SHR}^n(x)$: right shift operation, it is defined by $x \gg n$;

where x is a w-bit word and and n is an integer such that $0 \leq n < w$.

SHA-512 functions

SHA-512 uses six logical functions defined as in Table 1.1.

Function	Definition
$\mathrm{Ch}(x,y,z)$	$(x \wedge y) \oplus (\neg x \wedge z)$
$\mathrm{Maj}(x,y,z)$	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\Sigma_0^{\{512\}}(x)$	$\mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x)$
$\Sigma_1^{\{512\}}(x)$	$\mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x)$
$\sigma_0^{\{512\}}(x)$	$\mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x)$
$\sigma_1^{\{512\}}(x)$	$\mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x)$

Table 1.1: SHA-512 logical functions [2].

They all operate on 64-bit words and the result is also a new 64-bit word.

SHA-512 constants

SHA-512 uses a sequence of 80 constant 64-bit words, $K_0^{\{512\}}, \cdots, K_{79}^{\{512\}}$, derived from the fractional parts of the cube roots of the first eighty prime numbers. The complete list of constants is reported in Appendix A.8.

Secure Hash Algorithms preprocessing

The preprocessing step for Secure Hash Algorithms consists of three phases: message padding, message parsing, initial hash value setting.

SHA-512 message padding. Padding is performed to ensure that the message is a multiple of 1024 bits. Given a message M of length ℓ bits, the padding is performed by appending a single bit equal to 1 to the end of the message followed by k bits set to 0, where k is the smallest non-negative integer satisfying the equation $\ell + 1 + k \equiv 896 \pmod{1024}$. Then the binary representation of the number ℓ is appended as a 128-bit block. For instance, considering the message "abc" of length 24 bits, it will be padded with 871 zero bits and with the message length as shown in Figure 1.2.

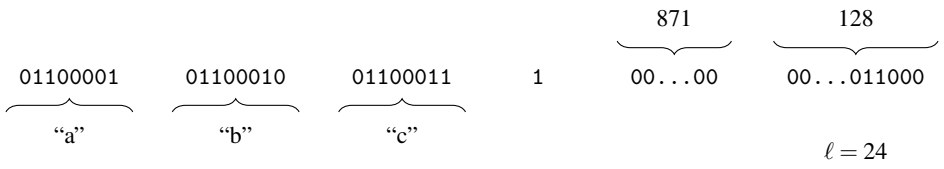


Figure 1.2: SHA-512 padding example for the ASCII string "abc".

SHA-512 message parsing. For SHA-512, the padded message is parsed into N 1024-bit blocks called $M^{(1)}, \cdots, M^{(N)}$.

SHA-512 initial hash value. For SHA-512, the initial hash value is composed of the following eight 64-bit words:

$$\begin{aligned} H_0^{(0)} &= \texttt{6a09e667f3bcc908} \\ H_1^{(0)} &= \texttt{bb67ae8584caa73b} \\ H_2^{(0)} &= \texttt{3c6ef372fe94f82b} \\ H_3^{(0)} &= \texttt{a54ff53a5f1d36f1} \\ H_4^{(0)} &= \texttt{510e527fade682d1} \\ H_5^{(0)} &= \texttt{9b05688c2b3e6c1f} \\ H_6^{(0)} &= \texttt{1f83d9abfb41bd6b} \\ H_7^{(0)} &= \texttt{5be0cd19137e2179} \end{aligned} \tag{1.1}$$

SHA-512 hash computation

To produce the final digest, the SHA-512 [2] algorithm uses:

- a message schedule of eighty 64-bit words labeled $W_0, \cdots, W_{79}$;
- eight 64-bit working variables labeled a, b, c, d, e, f, g, h;
- a hash value of eight 64-bit words labeled $H_0^{(i)}, \cdots, H_7^{(i)}$, where i is the round number;
- two temporary words T_1 and T_2.

For each message block $M^{(1)}, \cdots, M^{(N)}$:

1. *Prepare the message schedule*. For each message block, the first sixteen words of the message schedule correspond directly to the parsed message block:

   $$W_t = M_t^{(i)} \qquad 0 \leq t \leq 15.$$

   The remaining words are generated recursively from previously computed values.

   $$W_t = \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} \qquad 16 \leq t \leq 79. \tag{1.2}$$

   The message schedule expands the original sixteen 64-bit words into eighty words in order to increase diffusion throughout the compression process.

2. *Initialize the working variables*.

   $$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \\ f &= H_5^{(i-1)} \\ g &= H_6^{(i-1)} \\ h &= H_7^{(i-1)}. \end{aligned} \tag{1.3}$$

3. *For* $t = 0$ *to* 79: at each round, the working variables are updated through a combination of nonlinear logical

functions, modular additions, and bitwise rotations.

$$\begin{aligned} T_1 &= h + \Sigma_1^{\{512\}}(e) + \mathrm{Ch}(e,f,g) + K_t + W_t \\ T_2 &= \Sigma_0^{\{512\}}(a) + \mathrm{Maj}(a,b,c) \end{aligned} \tag{1.4}$$

$$\begin{aligned} h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2. \end{aligned} \tag{1.5}$$

4. *Compute the i^{th} intermediate hash value $H^{(i)}$.*

$$\begin{aligned} H_0^{(i)} &= H_0^{(i-1)} + a \\ H_1^{(i)} &= H_1^{(i-1)} + b \\ &\vdots \\ H_7^{(i)} &= H_7^{(i-1)} + h. \end{aligned} \tag{1.6}$$

After processing all the message blocks, the resulting 512-bit digest of the message M is given by:

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, \cdots \,||\, H_7^{(N)}. \tag{1.7}$$

SHA-512 computation example

This section presents an example of how the SHA-512 hashing process works using the string `abc` as a starting point. The example applies all the preprocessing and compression steps described in the previous sections without reporting all eighty compression rounds individually.

Given the input message $M =$ `abc`, it can be written as `61 62 63` using the hexadecimal ASCII encoding. Therefore, the original message has length $\ell = 3 \cdot 8 = 24$ bits. As shown in Figure 1.2, the padding procedure appends a single bit equal to 1, followed by 871 zero bits and the 128-bit representation of the original message length. The padded message consists only of a single SHA-512 block, so $N = 1$. Using hexadecimal notation, the resulting block is

$$\begin{aligned} M^{(1)} = {} & \texttt{6162638000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000000} \\ & \texttt{0000000000000000000000000000000018}. \end{aligned}$$

The block is then parsed into sixteen 64-bit words as follows

$$W_0 = M_0^{(1)} = \texttt{6162638000000000},$$

$$W_1 = W_2 = \cdots = W_{14} = 0,$$

$$W_{15} = M_{15}^{(1)} = \texttt{0000000000000018}.$$

The remaining words of the message schedule are computed using Equation 1.2. For example,

$$W_{16} = \sigma_1^{\{512\}}(W_{14}) + W_9 + \sigma_0^{\{512\}}(W_1) + W_0.$$

which gives $W_{16} = \texttt{6162638000000000}$, since W_{14}, W_9, and W_1 are all zero. The same recurrence is applied until all the $W_0, \ldots, W_{79}$ words have been generated.

In the next step, the working variables $a, \ldots, h$ are initialized as reported in Equation 1.1. Then, for each round $t = 0, \ldots, 79$, the temporary values T_1 and T_2 are computed using Equation 1.4, and the working variables are updated according to Equation 1.5. All additions are performed modulo 2^{64}.

After the eighty round processing, the eight working variables are added to the initial hash value as specified by Equation 1.6. Since the padded message contains only one block, the resulting intermediate hash value is also the final hash value.

The final SHA-512 digest of `abc` is therefore

$$\begin{aligned}\text{SHA512}(\texttt{abc}) = {} & \texttt{ddaf35a193617abacc417349ae204131} \\ & \texttt{12e6fa4e89a97ea20a9eeee64b55d39a} \\ & \texttt{2192992a274fc1a836ba3c23a3feebbd} \\ & \texttt{454d4423643ce80e2a9ac94fa54ca49f},\end{aligned}$$

which consists of 128 hexadecimal digits, or 512 bits. This simple example shows how a short input message is transformed into a fixed-length digest through all the SHA-512 different steps.

1.3 The `crypt` library, `mkpasswd`, and the `sha512crypt` algorithm

Modern Unix-like operating systems rely on password hashing schemes to securely store user credentials. Instead of saving passwords in plaintext, operating systems store the output of computationally expensive hashing algorithms combined with random salts. This approach reduces the impact of password leaks and lowers the effectiveness of brute-force and precomputed attacks such as rainbow tables [1, 3].

In Linux systems, password hashing is usually handled through the `crypt` function, which provides a standardized interface for multiple password hashing schemes [11]. Among the available schemes, `sha256crypt` and `sha512crypt` are particularly relevant for this thesis since they are based on the SHA-2 family of cryptographic hash functions and internally perform multiple SHA computations [2, 8].

1.3.1 The `crypt` library

`crypt` is a POSIX C library function commonly used in Unix-like operating systems to hash user passwords. The function returns as output a text string which contains all the information required for password verification, including the hashing algorithm, the salt, optional configuration parameters, and the final digest [11].

The same `crypt` function is used both to generate new password hashes and to verify passwords during authentication. To verify a password, the stored salt and parameters are extracted from the encoded string and reused to compute the digest. The output string follows the *Modular Crypt Format* (MCF), which is defined as:

```
$<id>[$<param>=<value>(,<param>=<value>)*][$<salt>[$<hash>]]
```
(1.8)

The MCF representation allows password hashes to be self-descriptive, since the output string embeds the algorithm identifier, configuration parameters, salt, and final digest [11]. The fields of the format have the following meaning:

- `id`, the hashing algorithm identifier;
- `param=value`, optional complexity parameters, such as rounds/iterations;
- `salt`, the salt used during the hashes process;
- `hash`, the final radix-64 encoded digest.

Unlike standard Base64 encoding, the MCF specification uses a custom radix-64 alphabet [8, 11]:

```
./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

The salt should be a reasonably random string of arbitrary length, however the amount of actual salt characters used may differ depending on the hash function chosen. Table 1.2 shows the most common password hashing schemes supported by the `crypt` function.

Identifier	Password hashing scheme
–	DES (default legacy scheme)
1	`md5crypt`
2a, 2b	`bcrypt`
5	`sha256crypt`
6	`sha512crypt`
7	`scrypt`
y	`yescrypt`

Table 1.2: Password hashing schemes and their identifiers.

Among these schemes, `sha256crypt` and `sha512crypt` are particularly relevant to the analysis conducted in this thesis.

1.3.2 `mkpasswd`

The `mkpasswd` is a command-line utility commonly available on Linux systems that provides a high-level interface to the `crypt` function [11, 12]. It allows users to generate password hashes using different password hashing schemes supported by the underlying `crypt` function. Internally, the utility is responsible for:

- parsing command-line arguments;
- validating user parameters;
- generating random salts when none are provided;
- invoking the selected hashing scheme if provided, otherwise it uses DES as default one.

In this thesis, `mkpasswd` is used as the reference implementation for evaluating the execution time of iterative SHA-512-based password hashing schemes.

1.3.3 `sha512crypt`

`sha256crypt` and `sha512crypt` are two password hashing schemes developed by Ulrich Drepper for Red Hat [8]. The reason that led to the development of these methods was the absence of a password hashing scheme, provided by the `crypt` function, that used a NIST-approved algorithm. For this purpose, Ulrich Drepper developed a scheme based on the SHA-2 hash functions. The output digest of these hashes starts with `$5$` (for SHA-256) and `$6$` (for SHA-512) depending on which SHA variant is used [8, 11]. Unlike plain SHA-512 hashing, `sha512crypt` is specifically designed for password storage and therefore includes salting and configurable iterative rounds in order to increase computational cost.

When using SHA-based methods, the salt string can be a generic length string of which only the first 16 characters are used. The `<SALT>` string can optionally start with

`rounds=<N>$`

where N is an unsigned integer specifying the number of iterations performed by the algorithm. The default number of rounds for both SHA-based algorithms is $N = 5000$, while the allowed range is:

$$1.000 \leq N \leq 999.999.999.$$

These limits are enforced to guarantee a minimum security level while avoiding impractically large execution times. Any selection of N outside the allowed range will cause the usage of the minimum or maximum allowed value respectively.

The radix-64 encoded digest length is fixed to 43 characters for SHA-256 and to 86 characters for SHA-512 [8]. Therefore, the maximum length of the entire output string `$<id>[$<param>=<value>(,<param>=<value>)*][$<salt>[[$<hash>]]` (excluding the final NULL byte in C string representation) is:

- 80 characters for SHA-256;
- 123 characters for SHA-512.

1.3.4 `sha512crypt` internal structure

The `sha512crypt` algorithm computes multiple intermediate digests and auxiliary byte sequences denoted as A, B, DP, DS, P, and S [8]. The algorithm uses three primitives to produce a hash digest:

- *Digest initialization*: creation and initialization of a digest context data structure.
- *Digest update*: insertion of additional bytes into the digest context. This operation can be executed multiple times. Only when the required number of bytes to perform a round of the hash function is reached, then the round is executed, otherwise the bytes will simply be queued up. For SHA-256 and SHA-512 the respective sizes are 64 and 128 bytes.
- *Digest finalization*: padding of the remaining bytes, computation and output of the final digest.

Whenever the algorithm refers to adding the salt string, it means the salt string truncated to 16 characters. When the algorithm refers to adding a string, the terminating NULL character of the C string representation is not added.

Initial digest construction

The algorithm first computes two intermediate digests, denoted as A and B.

Digest A Digest A is initialized as follows:

1. initialize digest A;
2. add the password string;
3. add the salt string (this only the `<salt>` string itself without any identifier, optional parameter or enclosing $).

Digest B Digest B is constructed as follows:

1. initialize digest B;
2. add the password string;
3. add the salt string;
4. add the password string again;
5. finalize digest B.

Then, for each block of 32 or 64 bytes in the binary representation of the password string (depending on the algorithm adopted), digest B is repeatedly added to digest A. For the remaining n bytes of the password string, the first n bytes of digest B are added to digest A. For each bit b of the password string, starting from the lowest bit position:

- if $b = 1$, then digest B is added to digest A;
- if $b = 0$, then the password string is added to digest A.

Eventually, digest A is finalized.

Auxiliary byte sequences

Two auxiliary digests, DP and DS, are generated to produce the byte sequences P and S. Both byte sequences will have the same length as the password string.

Sequence P Digest DP is generated by repeatedly hashing the password string for a number of times equal to its byte number (excluding the final NUL byte). The byte sequence P is obtained by adding digest DP for the number of 32 or 64-byte block in the password string. And then, given the n remaining bytes of the password, the respective n bytes of digest DP are added to P.

Sequence S Digest DS is generated by repeatedly hashing the salt string for a number of times equal to $16 + \mathrm{A}[0]$, where A[0] is the digest A first byte interpreted as an unsigned 8-bit integer. The byte sequence S is produced in the same way as P, except that this time the salt string is used.

Iterative rounds

The algorithm core consists of an iterative loop executed N times according to the specified number of rounds. For each $0 \leq i \leq N-1$:

1. initialize digest C_i;
2. if i is odd, P is added to C_i;
3. if i is even, A/C_{i-1} is added to C_i;
4. if i is not divisible by 3, S is added to C_i;
5. if i is not divisible by 7, P is added to C_i;
6. if i is odd, A/C_{i-1} is added to C_i;
7. if i is even, P is added to C_i;
8. finalize digest C_i;

In the steps 3 and 6, digest A is used only when $i = 0$. This repeated recombination of password-derived and salt-derived byte sequences increases the computational complexity of the algorithm and reduces the effectiveness of precomputed attacks.

Final encoding

After the final iteration, the resulting digest is permuted according to a predefined byte-reordering scheme specified by the SHA-crypt standard. The reordered digest is then encoded using the custom radix-64 encoding and combined with:

- the algorithm identifier;
- the optional `rounds=<N>` specification;
- the salt string.

The final output therefore follows the Modular Crypt Format introduced previously. Since `sha512crypt` internally performs a large number of SHA-512 computations, it represents an interesting reference algorithm for evaluating the effectiveness of parallel hashing strategies.

`sha512crypt` computation example

The description of the `sha512crypt` algorithm is concluded reporting a practical example of its usage. Given the following input values:

- `password`: `password`;
- `salt`: `12345678`;
- `rounds`: `5000`, the default value;

the corresponding `sha512crypt` hash can be generated using the `mkpasswd` utility as follows:

```
mkpasswd --method=sha-512 --rounds=5000 --salt=12345678 password
```

Listing 1.1: Generation of a `sha512crypt` hash using `mkpasswd`.

The resulting output string is:

```
$6$rounds=5000$12345678$I8tr4xFAC6/TtjYWdp0LWEjQre2LcYm
2jdSMNLQDIyqRv.cKo7KMD5/HpzVVFKpUQlIekr/Vw.OdImtRM85fg/
```

Listing 1.2: Example of a `sha512crypt` output string.

This output string follows the Modular Crypt Format presented in Equation 1.8 and can be divided into the following fields:

- `$6$` identifies the `sha512crypt` scheme;
- `rounds=5000` specifies the number of rounds;
- `12345678` is the salt string used;
- and the final 86-character digest string which uses the radix-64 representation.

Internally, the `sha512crypt` scheme uses the exact procedure described in Section 1.3.3, computing several intermediate digests and byte sequences. After the final rounds, the resulting digest is reordered according to the `sha512crypt` specification and encoded using the custom radix-64 alphabet. The encoded string contains all the information required to verify the correctness of the password, except for the password itself. During the authentication phase, the implementation extracts all the information from the stored string, applies the same `sha512crypt` procedure and compares the newly generated encoded digest with the stored one. If they match, then the authentication succeeds.

1.4 Merkle trees

Merkle trees were first introduced by Ralph Merkle in 1979 [4, 5]. They provide an efficient and secure method to verify the integrity of very large data structures. A Merkle tree is a binary tree constructed from the bottom up in the following way:

- each leaf node contains the hash of a specific data block;
- each non-leaf node contains the hash of the concatenation of its children's hash values.

Let $D = \{D_1, \ldots, D_n\}$ be the set of data blocks. Each block D_i is given as input to a cryptographic hash function h, producing a set of leaf nodes $L = \{L_1, \ldots, L_n\}$, where $L_i = h(D_i)$. The internal nodes are derived by recursively hashing pairs of child nodes:

$$N_{ij} = h(N_i \,\|\, N_j),$$

where N_i and N_j are the child nodes, N_{ij} is their parent node, and $\|$ is the concatenation operator. This process is repeated until the root node of the tree is derived. This node, called the *Merkle root* R, represents the entire dataset hash [9]:

$$R = h(N_{\text{left}} \,\|\, N_{\text{right}}).$$

Figure 1.3 illustrates the fundamental structure of a Merkle tree.

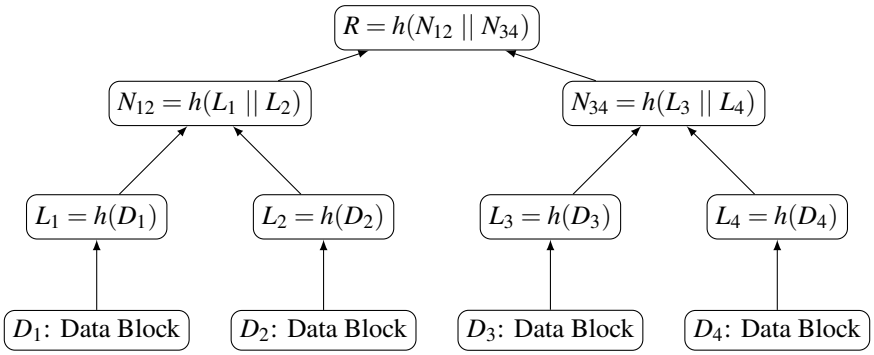


Figure 1.3: Merkle tree example with 4 leaves.

The authentication path (or *Merkle proof*) in a Merkle tree is the sequence of sibling hashes required to verify the integrity of a specific data block. For a given block D_i, its proof $P(D_i)$ is defined as

$$P(D_i) = N_{sib_1}, N_{sib_2}, \ldots, N_{sib_m},$$

i.e., the ordered set of sibling nodes whose combination with the hash of the data block allows the recomputation of the Merkle root. The verification procedure is as follows [9]:

1. **Initial Hashing**: compute the hash of the given data block:

$$L_i = h(D_i) \qquad N_{par_0} = L_i.$$

2. **Path Reconstruction**: for each sibling node N_{sib_j} in $P(D_i)$, compute the parent hash:

$$N_{par_j} = h(N_{par_{j-1}} \| N_{sib_j}).$$

3. **Root Comparison**: repeat the previous step until there are no more siblings inside $P(D_i)$, i.e. the root R' of the tree is obtained. The data block D_i is authentic and unaltered if and only if

$$R' = R.$$

Figure 1.4 represents graphically the process just described.

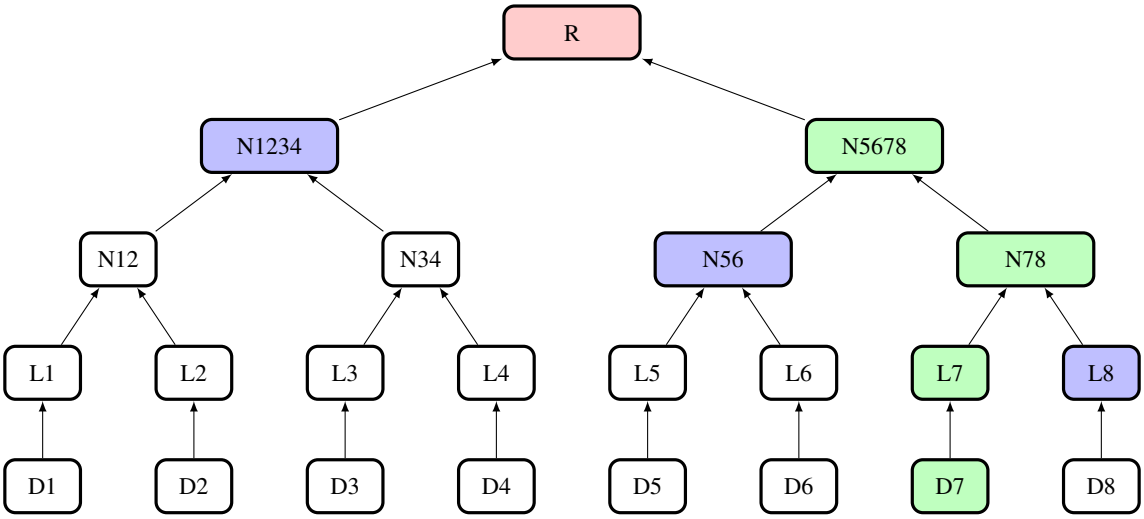


Figure 1.4: Merkle path visualization.

Merkle trees are widely used in computer science due to their favorable performance characteristics. The main complexity metrics relevant to this work are [4, 9]:

- *Verification*: $O(\log n)$ to prove the inclusion of a single data block.
- *Root Calculation*: $O(\log n)$ when updating a single leaf node.
- *Space Complexity*: $O(n)$ to store the entire tree.
- *Proof Size*: $O(\log n)$ hashes in a Merkle proof (i.e., hashes to transmit). More precisely, considering binary trees, the Merkle proof cardinality is given by $|P(D_i)| = \log_2(n)$.

Where n is the number of data blocks, or leaves.

In this work, Merkle trees are used to implement a multithreaded hashing algorithm that distributes the computational load across CPU cores. As a simple example, the Merkle tree represented in Figure 1.4 can be divided into two independent subtrees rooted at $N1234$ and $N5678$. These subtrees can be processed concurrently by two worker threads, each of which performs the hash computations until a single subtree root is obtained. The two resulting hashes are then combined to obtain the final Merkle root R. A detailed description of this parallel computation is provided in Chapter 2.

1.5 OpenSSL

OpenSSL is an open-source cryptographic toolkit and software library widely adopted in secure communication systems and Internet server infrastructures. It provides implementations of several cryptographic protocols, including SSL/TLS, Datagram TLS (DTLS), and QUIC, together with a general-purpose cryptographic library offering an extended set of cryptographic primitives [13]. The OpenSSL project was founded in 1998 to provide a free set of encryption tools for the Internet as a continuation of the SSLeay library developed by Eric A. Young and Tim J. Hudson. The core library is implemented in the C programming language and is available on most major operating systems, such as Linux, macOS, BSD, Microsoft Windows and OpenVMS [13, 14]. The OpenSSL toolkit includes:

- **libssl**, which provides the implementation of the SSL/TLS, DTLS and QUIC protocols;
- **libcrypto**, a general-purpose cryptographic library that implements several cryptographic primitives and algorithms, including symmetric and asymmetric encryption schemes, digital signatures, random number generators, and cryptographic hash functions such as MD5, SHA-256, SHA-512;
- **openssl**, a command line tool that can be used to perform several cryptographic operations, including certificate generation, encryption and decryption, digest computation, and protocol testing.

In this thesis, the `libcrypto` component of OpenSSL is used to compute SHA-512 digests [13, 14]. In particular, the library provides highly optimized, efficient and tested implementations of cryptographic primitives, allowing the experimental analysis to overlook the efficiency of the hash functions implementation itself and, instead, to focus on the effectiveness of the proposed parallelization strategies.

The usage of the OpenSSL library allows the reproducibility of the experiments conducted and ensures the reliability of the results presented. Other advantages of the OpenSSL library usage are its wide adoption and continuous maintainment. Furthermore, OpenSSL includes architecture-specific optimizations and low-level implementations that can exploit modern CPU features and instruction sets, improving significantly the hashing performances. The algorithms developed in this thesis rely on the SHA-512 implementation exposed by the **libcrypto** library [2, 14]. By delegating the cryptographic primitive implementation to a highly optimized and well-known library, the analysis presented in this work can be focused on the scalability, workload distribution, and performance characteristics of the proposed Merkle-tree-based parallel hashing approaches. All experiments discussed in this thesis were conducted using OpenSSL version 3.0.

1.6 OpenCL Framework

OpenCL (Open Computing Language) is an open-source standard for heterogeneous parallel programming across a wide range of computing devices, including CPUs, GPUs, DSPs, FPGAs, and others. The standard defines a custom

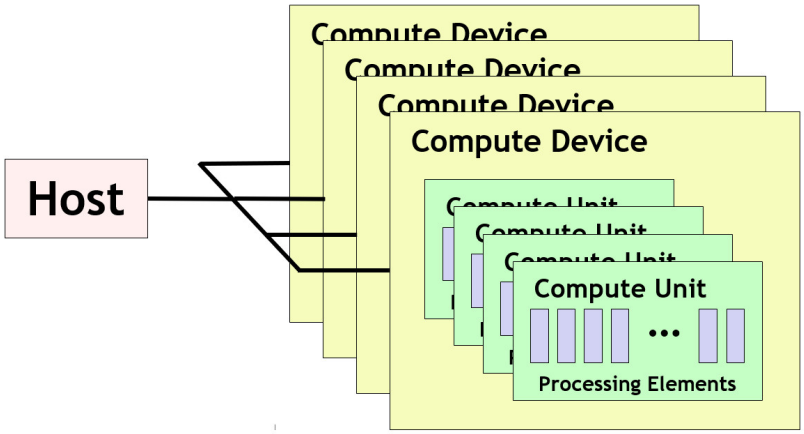


Figure 1.5: OpenCL platform model showing the relationship between the host and compute devices. Adapted from the Khronos OpenCL documentation [15].

programming language, called OpenCL C, and a set of application programming interfaces (APIs) to control the platform selection, program execution, memory management. The standard is maintained by the Khronos Group, which is a non-profit, open standards organization. The OpenCL framework gained popularity because it enables efficient parallel programming across a wide array of hardware platforms [15].

OpenCL views a computing system as set of compute devices, typically CPUs and GPUs, coordinated by a host program. A compute device is usually composed of many compute units, for instance a CPU core, capable of executing many work-items concurrently.

1.6.1 OpenCL Platform Model

The OpenCL Platform Model, shown in Figure 1.5, defines the relationship between the host, typically a CPU, and one or more compute devices such as GPUs, FPGAs, and hardware accelerators. The OpenCL runtime connects the host and the devices, allowing efficient parallel processing through coordinated execution [15]. The main components of the OpenCL Platform Model are:

- **Host program**: the host program runs on the CPU and is responsible for managing the execution of the OpenCL application. It coordinates memory allocation, kernel launches, synchronization, and data transfers.
- **Compute devices**: compute devices are hardware components capable of executing OpenCL kernels. They include:
  - *CPUs*: typically used to handle serial or control-oriented tasks.
  - *GPUs*: particularly suited for computations involving massive data sets and perform highly parallelizable tasks.
  - *FPGA (Field-programmable gate array)*: configurable hardware accelerators optimized for application-specific computations.
  - *Other accelerators*: dedicated hardware units designed to perform specific calculations very efficiently.

The Platform Model abstracts the underlying hardware architecture, allowing developers to write portable code that can be executed on different types of hardware without modification.

1.6.2 Execution model

The Execution Model in OpenCL, shown in Figure 1.6, describes how parallel programs are executed across compute devices using kernels, which are the fundamental work units in OpenCL [15]. Kernels are launched across multiple work-items and organized into work-groups.

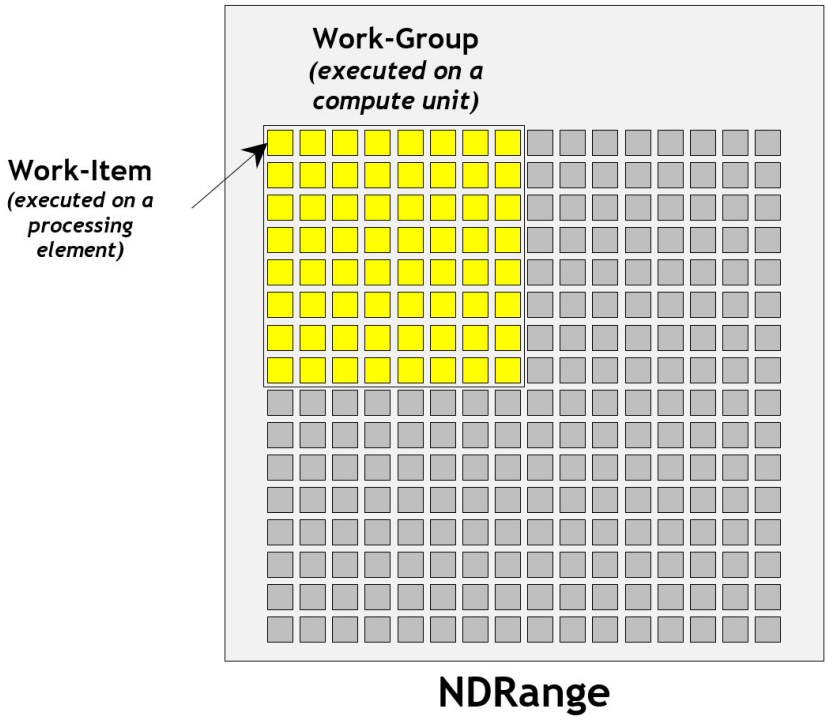


Figure 1.6: OpenCL execution model with NDRange decomposition into work-groups and work-items. Adapted from the Khronos OpenCL documentation [15].

- **Kernel**: a kernel is a function written in OpenCL C that is executed on a compute device. Each work-item executes an independent instance of the kernel.
- **NDRange (N-Dimensional Range)**: defines the global index space over which a kernel is executed. The execution domain can be one-, two-, or three-dimensional, depending on the problem being solved.
- **Work-item**: a work-item is the smallest unit of work in OpenCL. Each work-item executes an instance of a kernel, and the execution of all work-items occurs in parallel across available computing resources.
- **Work-group**: a work-group is a collection of work-items that execute together. Work-items within a work-group can share local memory and synchronize their execution.

OpenCL kernels are typically compiled at run-time for the target device, enabling portability across heterogeneous architectures.

1.6.3 Memory model

OpenCL defines a hierarchical memory model, shown in Figure 1.7, composed of four different memory regions [15]:

- *Global memory*: memory shared by all work-items, it is the slowest type of memory. Typically used to store large input or output buffers that need to be shared across work-items. (`__global` identifier)
- *Read-only memory*: small, low latency memory writable only by the host CPU, but not by computing devices. It is shared across all work-items and it is generally faster than global memory. (`__constant` identifier)
- *Local memory*: memory shared within work-items belonging to the same work-group. It provides faster access than global memory and is ideal for storing intermediate results. (`__local` identifier)
- *Private memory*: memory used by individual work-items and is the fastest memory type. It is used for storing temporary variables that are local to each work-item. (`__private` identifier)

Depending on the hardware architecture, devices may or may not share memory with the host CPU. The OpenCL API provides functions to transfer data between host and device memory spaces. Efficient usage of the memory hierarchy is

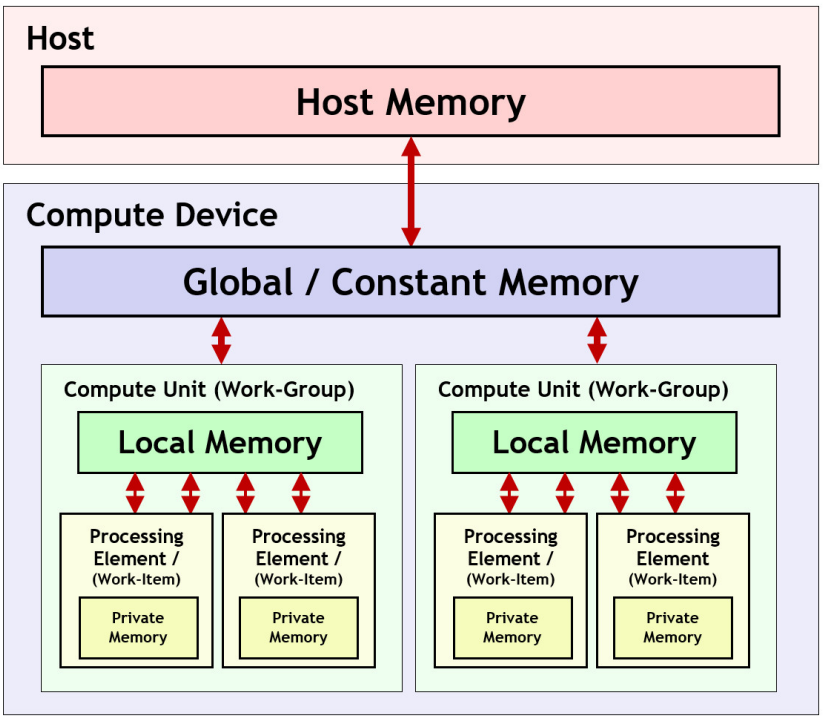


Figure 1.7: OpenCL memory hierarchy and visibility scopes. Adapted from the Khronos OpenCL documentation [15].

essential to achieve high performance in OpenCL applications, since memory latency and bandwidth are often the cause of bottlenecks.

1.6.4 OpenCL C programming language

The standard also defines a C-like programming language called OpenCL C, that is used to implement OpenCL kernels. OpenCL C is based on the C99 standard adapted to fit the device model in OpenCL. It supports the usage of memory type qualifiers: `__global`, `__constant`, `__local`, `__private`. OpenCL C functions are declared with `__kernel` to indicate that they are entry points into the program to be called from the host program. Standard C features such as function pointers, bit fields and variable-length arrays are not supported, and furthermore recursion is forbidden [15].

1.6.5 OpenCL for cryptographic applications

OpenCL is widely used in cryptographic applications to accelerate security-related tasks across heterogeneous hardware, such as password cracking and brute-force attacks. Popular password recovery tools such as Hashcat and John The Ripper provide OpenCL implementations of all the main cryptographic algorithms, hash functions, and protocols [6, 15, 16]. In this thesis, the OpenCL framework is used to provide a GPU-accelerated implementation of the multithreaded hashing algorithms proposed, leveraging the massive parallelism offered by modern graphics processing units.

1.7 hashcat

Hashcat is an open-source advanced password recovery tool [6, 7]. It is widely used in cybersecurity and password auditing and it supports a large number of cryptographic hash functions and password hashing schemes, including MD5, SHA-family algorithms, bcrypt, scrypt, WPA/WPA2, and several other authentication formats used in operating systems and network infrastructures [17, 18]. Hashcat supports both CPU- and GPU-based executions, exploiting modern parallel-computing frameworks such as OpenCL and CUDA to leverage GPUs processing power to accelerate hash computations [6, 15]. This makes hashcat one of the most adopted tools for evaluating large-scale password hashing and recovery. In this scenario, since cryptographic hash computations are typically independent, GPUs are particularly suitable to execute

independent jobs in parallel on thousands of processing units simultaneously. Compared to CPUs, which are optimized for low-latency sequential execution, GPUs are designed to maximize throughput on massively parallel workloads. However, not all algorithms equally benefit from GPU acceleration. Modern hashing schemes, such as bcrypt, scrypt, and Argon2, are specifically designed to reduce the effectiveness of parallel computation by introducing memory-hard operations and sequential dependencies, limiting the achievable speed-up on GPU architectures [17, 18, 19].

1.7.1 Hashcat architecture

Historically, hashcat was distributed in two separate variants:

- **hashcat**, the CPU oriented implementation;
- **oclHashcat/cudaHashcat**, the GPU-accelerated implementations based on OpenCL and CUDA respectively.

Starting from version 3.00, the two tools were merged into a single one named simply hashcat [6]. The GPU backend of hashcat relies on OpenCL kernels to implement the most common cryptographic hash functions and password hashing schemes [7, 15].

1.7.2 Attack modes

Hashcat offers multiple attack modes to explore effectively and efficiently large password keyspaces. Some of the most commonly used attack strategies are:

- **Brute-force attacks**: which consist of exhaustively testing all possible password combinations within a specified character set;
- **Dictionary attacks**: which consist of testing passwords contained in predefined wordlists;
- **Rule-based attacks**: which apply transformation rules to dictionary entries in order to generate new password candidates to test;
- **Mask attacks**: which allow the search space restriction by specifying known password patterns;
- **Hybrid attacks**: which allow the combination of different approaches, such as dictionary with brute-force attacks.

1.7.3 Hashcat usage in this thesis

Hashcat provides a freely available GitHub repository containing its complete source code [7]. In this thesis, the GPU implementations of the algorithms discussed adopt the OpenCL SHA-512 implementation provided by hashcat. The hashcat repository includes a dedicated folder containing all OpenCL kernels which implement the most common cryptographic hash functions. In particular, the OpenCL kernels files used by hashcat to compute SHA-512 hashes are integrated in the proposed GPU-based implementation of the `parallel_sha512` algorithm.

The adoption of hashcat's OpenCL kernels offers several advantages. First, hashcat provides highly optimized and tested GPU implementations. This allows the experimental analysis to focus on the proposed Merkle-tree-based parallelization algorithm rather than on the low-level GPU optimizations of the SHA-512 hash function. Furthermore, considering the wide usage of hashcat, relying on its OpenCL implementations improves the reliability and the reproducibility of the experimental evaluations presented. This also ensures that the performance measurements primarily reflect the scalability, workload distribution, and synchronization overhead of the proposed parallel hashing algorithm, rather than differences caused by custom SHA-512 implementations.

1.7.4 Hashcat SHA-512 OpenCL implementation

Hashcat implements the SHA-512 algorithm through a reusable collection of OpenCL data structures, macros, and kernels. The implementation is mainly distributed between the inc_hash_sha512.h header file, which defines the hashing context structure and the low-level primitives, and the source file inc_hash_sha512.cl, which provides a large number of digest initialization/update/finalization methods.

Hashcat divides the SHA-512 algorithm into several functions that resemble the initialization, update, compression, and finalization stages described in Section 1.2.1. This implementation is very similar to the approach adopted by commonly used cryptographic libraries, such as OpenSSL, to implement hash or encryption functions [14]. This allows different hashcat kernels and password-hashing schemes to use the same SHA-512 implementation.

The following discussion focuses on the components that are relevant to the algorithms developed in this thesis. The variants related to alternative character encodings will be omitted.

Hashing context and low-level primitives

The state of an ongoing SHA-512 computation is stored inside the `sha512_ctx` structure located in the header file. It contains the hash state, the input buffer and an integer counting the number of bytes processed. The buffer is split into 32-bit words because this is the most convenient layout for the data representation adopted by hashcat. Listing 1.3 reports a simplified version of the context structure.

```
typedef struct sha512_ctx {
    u64 h[8];       // 8 x 64-bit hash state (a..h)
    u32 w0[4];      // 128-byte input buffer, split into
    u32 w1[4];      // 8 groups of 4 x u32 = 16 bytes each
    ...
    u32 w7[4];
    int len;        // total bytes processed so far
} sha512_ctx_t;
```

Listing 1.3: `sha512_ctx` context structure.

The vectorized version `sha512_ctx_vector_t` of the context structure is also provided. In this structure the `u64`/`u32` data types are replaced with the respective vectorized types `u64x`/`u32x`, which allow SIMD processing of multiple passwords simultaneously.

Macros definitions. The header file also contains the core macros declaration, which are used to implement rotation/shifting/choice/majority functions introduced in Table 1.1. The functions Σ_0, Σ_1, σ_0, and σ_1 are implemented through 64-bit rotations, shifts, and exclusive-OR operations, while Ch and Maj are expressed using boolean formulations.

```
#define SHA512_STEP_S(F0,F1,a,b,c,d,e,f,g,h,x,K) {
    h += K;
    h += x;
    h += SHA512_S1_S(e);
    h += F0(e,f,g);     // Ch
    d += h;
    h += SHA512_S0_S(a);
    h += F1(a,b,c);     // Maj
}
```

Listing 1.4: Round step macro definition.

Listing 1.4 reports the implementation of the standard SHA-512 compression function specified in Equation 1.4 and Equation 1.5. This function accumulates into `h` the values of the K_t, W_t, $\Sigma_1(e)$, and $\mathrm{Ch}(e,f,g)$ terms and then computes the temporary value T_1. The operation `d += h` implements the round transformation $e = d + T_1$. The last two instructions compute $T_1 + T_2$, i.e. the additions of $\Sigma_0(a)$ and $\mathrm{Maj}(a,b,c)$. This function is implemented using an important optimization. Instead of performing the rotation on the `a...h` variables at each round, the macro takes all of them as parameters, allowing the caller to map the variables to different roles and avoiding several assignments per round. For

example, a,b,c,d,e,f,g,h in round N becomes h,a,b,c,d,e,f,g in round $N+1$ by simply changing the parameters order.

Initialization and input processing

The inc_hash_sha512.cl source file contains the actual OpenCL SHA-512 function implementations. The hashing context initialization is performed through sha512_init. Input data are then supplied to the sha512_update function and accumulated in the internal buffer. When a 128-byte block becomes available, the sha512_transform is invoked and it performs the intermediate digest update. Finally, the sha512_final function performs the final padding, processes the last block and produces the final digest.

Context initialization. The SHA-512 computation begins with the initialization of the sha512_ctx_t context structure. The sha512_init function initializes the SHA-512 state with the standard initial hash values specified in FIPS 180-4 and reported in Equation 1.1, zeros the 128-byte buffer and sets the length to 0. Listing 1.5 shows a simplified and abbreviated version of this function.

```
DECLSPEC void sha512_init(PRIVATE_AS sha512_ctx_t *ctx) {
    ctx->h[0] = SHA512M_A; ... ctx->h[7] = SHA512M_H;
    ctx->w0[0] = 0; ctx->w0[1] = 0; ... ctx->w7[3] = 0;
    ctx->len = 0;
}
```

Listing 1.5: sha512_init initialization function code.

Processing arbitrary-length input. After the initialization phase, the input message is passed to the context through a sha512_update invocation. This functions divides the input into 128-byte blocks and delegates the management of the internal block buffer to sha512_update_128. Listing 1.6 reports an abbreviated version of the sha512_update function.

```
DECLSPEC void sha512_update(PRIVATE_AS sha512_ctx_t *ctx, PRIVATE_AS const u32 *w, const
 ↪  int len) {
    u32 w0[4]; ... u32 w7[4];
    int pos1;
    int pos4;

    for(pos1 = 0, pos4 = 0; pos1 < len - 128; pos1 += 128, pos4 += 32) {
        w0[0] = w[pos4 +  0]; ... w7[3] = w[pos4 + 31];
        sha512_update_128(ctx, w0, ..., w7, 128);
    }

    w0[0] = w[pos4 +  0]; ... w7[3] = w[pos4 + 31];
    sha512_update_128(ctx, w0, ..., w7, len - pos1);
}
```

Listing 1.6: sha512_update function code.

The sha512_update function provides a public interface for sha512_update_128 that processes input of arbitrary length in 128-byte blocks. Hashcat provides several variants of this function to support different input representations and OpenCL address spaces. The _swap variants convert the byte order as expected by SHA-512, the _utf16le variants perform UTF-16 little-endian encoding conversion, and the _global variants read the input from global memory rather than private device memory. Having global and private memory separate functions is relevant for GPUs because it avoids

address space conversions and gives the OpenCL compiler explicit information about the location of each buffer. The reason why `_swap` variants are needed is that hashcat stores passwords as little-endian `u32` arrays, but the SHA-512 algorithm requires big-endian input.

Internal buffer management. The `sha512_update_128` function handles new data addition to the context buffer and it triggers a SHA-512 transform when the buffer fills up. The following list summarizes how the buffer management is performed.

- By computing `ctx->len & 127`, the function determines the current position in the context buffer.
- If the position is zero, then the buffer is empty and the new data can be copied directly.
- When the input fills up a 128-byte block, `sha512_transform` is immediately called. After its invocation the block is cleared.
- If the block already contains data, if the new data fit in the current block they are shifted by the appropriate amount and the merged into the buffer.
- If the input is larger than the remaining buffer space, the input is used to fill up the current block and the compression function is triggered. The remaining input bytes are then stored at the beginning of the next block.

An abbreviated version of the `sha512_update_128` function is reported in Listing 1.7.

```
1  DECLSPEC void sha512_update_128(PRIVATE_AS sha512_ctx_t *ctx, PRIVATE_AS u32 *w0, ...,
   ↪  PRIVATE_AS u32 *w7, const int len) {
2      if(len == 0) return;
3      const int pos = ctx->len & 127;
4      ctx->len += len;
5
6      if(pos == 0) {
7          ctx->w0[0] = w0[0]; ctx->w0[1] = w0[1]; ... ctx->w7[3] = w7[3];
8          if(len == 128) {
9              sha512_transform(ctx->w0, ..., ctx->w7, ctx->h);
10             ctx->w0[0] = 0; ctx->w0[1] = 0; ... ctx->w7[3] = 0;
11         }
12     } else {
13         if((pos + len) < 128) {
14             switch_buffer_by_offset_8x4_be_S(w0, ..., w7, pos);
15             ctx->w0[0] |= w0[0]; ctx->w0[1] |= w0[1]; ... ctx->w7[3] |= w7[3];
16         } else {
17             u32 c0[4] = { 0 }; ... u32 c7[4] = { 0 };
18             switch_buffer_by_offset_8x4_carry_be_S(w0, ..., w7, c0, ..., c7, pos);
19             ctx->w0[0] |= w0[0]; ctx->w0[1] |= w0[1]; ... ctx->w7[3] |= w7[3];
20             sha512_transform(ctx->w0, ..., ctx->w7, ctx->h);
21             ctx->w0[0] = c0[0]; ctx->w0[1] = c0[1]; ... ctx->w7[3] = c7[3];
22         }
23     }
24 }
```

Listing 1.7: `sha512_update_128` core buffer management function code.

Since the input is represented as a 32-bit words array, it cannot always be inserted into the buffer through a simple array copy. The `switch_buffer_by_offset_8x4_be_S` function is used to shift the word buffer by the required number of bytes while preserving the big-endian representation expected by SHA-512. This allows to merge the shifted words in the existing context buffer simply using bitwise OR operations.

Compression function

Whenever the update logic produces a complete 128-byte block, the block is passed to `sha512_transform`. This function implements the SHA-512 compression function and updates the intermediate digest stored in the context as described in Section 1.2.1.

The function first combines pairs of 32-bit words into 64-bit words through `hl32_to_64_S()`. Then it loads the current intermediate digest into the `a,b,...,h` variables. It performs the compression phase, which consists of eighty unrolled rounds. Rounds $0-15$ are performed without any expansion, rounds $16-79$ are expanded using the `SHA512_EXPAND_S` macro. After the compression phase, the working variables are added to the previous intermediate digest, producing the hash state to be used for the next block computation. There are several optimizations performed in the `sha512_transform` code:

- **Loop unrolling**: the eighty rounds are unrolled into individual statements. This removes loop-control instructions and it allows the GPU to schedule instructions more aggressively.
- **Interleaved expand+step**: from round 16 onward, each line computes the new word and uses it in the round step immediately after. This implementation may be enable better GPU instruction scheduling.
- **Variable rotation**: the `a,...,h` state variables are passed as parameters to `SHA512_STEP_S`, with the caller changing variable position, as mentioned before in the `SHA512_STEP_S` macro definition.

Listing 1.8 provides an abbreviated version of `sha512_transform`.

```
DECLSPEC void sha512_transform(PRIVATE_AS const u32 *w0, ..., PRIVATE_AS const u32 *w7,
 ↪  PRIVATE_AS u64 *digest) {
    // Step 1: u32 pairs to u64 words conversion
    u64 w0_t = hl32_to_64_S(w0[0], w0[1]);
    u64 w1_t = hl32_to_64_S(w0[2], w0[3]);
    ...
    u64 wf_t = hl32_to_64_S(w7[2], w7[3]);

    // Step 2: load state
    u64 a = digest[0]; ... u64 h = digest[7];

    // Step 3: 80 unrolled rounds
    // Rounds 0-15 without expansion
    SHA512_STEP_S(SHA512_F0o, SHA512_F1o, a,b,c,d,e,f,g,h, w0_t, SHA512C00);
    SHA512_STEP_S(SHA512_F0o, SHA512_F1o, h,a,b,c,d,e,f,g, w1_t, SHA512C01);
    // Rounds 16-79 with expansion
    w0_t = SHA512_EXPAND_S(we_t, w9_t, w1_t, w0_t); SHA512_STEP_S(..., w0_t, SHA512C10);
    w1_t = SHA512_EXPAND_S(wf_t, wa_t, w2_t, w1_t); SHA512_STEP_S(..., w1_t, SHA512C11);

    // Step 4: accumulation
    digest[0] += a; digest[1] += b; ... digest[7] += h;
}
```

Listing 1.8: `sha512_transform` compression function code.

Finalization

Listing 1.9 reports the simplified hashcat implementation of the `sha512_final` function.

```
DECLSPEC void sha512_final(PRIVATE_AS sha512_ctx_t *ctx) {
    const int pos = ctx->len & 127;
    append_0x80_8x4_S(ctx->w0, ..., ctx->w7, pos ^ 3);

    if(pos >= 112) {
        sha512_transform(ctx->w0, ..., ctx->w7, ctx->h);
        ctx->w0[0] = 0; ... ctx->w7[3] = 0;
    }

    ctx->w7[2] = 0;
    ctx->w7[3] = ctx->len * 8;
    sha512_transform(ctx->w0, ..., ctx->w7, ctx->h);
}
```

Listing 1.9: `sha512_final` finalization function code.

The `sha512_final` function implements the Merkle-Damgård padding:

- it appends the `0x80` byte after the last data byte;
- if the remaining space is less than 16 bytes (which means that `pos` is greater than 112), then a full extra block is needed;
- the 128-bit message length goes in the last 16 bytes, however only `w7[3]` is set which means that this implementation correctly handles only messages up to 2^{32} bits ($\sim$ 512 MB).

Example of the Hashcat SHA-512 workflow

This section illustrates how the hashcat OpenCL interface described in the previous sections can be used to compute a SHA-512 digest. Listing 1.10 summarizes the operations required to hash the ASCII string `abc` and illustrates the interaction between the main SHA-512 functions.

```
// 128-byte input buffer represented as 32-bit words
u32 message[32] = { 0 };
// ASCII encoding of "abc": 0x61 0x62 0x63
message[0] = 0x61626300;
sha512_ctx_t ctx;
sha512_init(&ctx);
sha512_update(&ctx, message, 3);
sha512_final(&ctx);
// ctx.h[0], ..., ctx.h[7] contain the SHA-512 digest
```

Listing 1.10: Simplified use of Hashcat's SHA-512 context interface.

The input buffer is represented as an array of 32-bit words, where the first word contains the ASCII representation of the `abc` string. The remaining words are all initialized to zero. The kernel then creates a SHA-512 context and initializes it using the `sha512_init` function. The three input bytes are supplied to the context through the update function. Since the input is shorter than one complete block, no compression is performed at this stage. The SHA-512 digest computation is completed by invoking `sha512_final`. The final content of the `ctx.h` array corresponds to the standard SHA-512 digest.

Chapter 2. Multithread hashing algorithm

This chapter presents the design and discusses the implementation of the parallel SHA-512-based hashing constructions proposed in this thesis. Traditional password-hashing constructions such as PBKDF2 and `sha512crypt` increase their computational cost primarily through sequential iteration [8, 20]. Increasing the iteration count consequently increases both the defender's password-verification latency and the work required by an attacker by approximately the same factor. The approach proposed in this thesis follows a different strategy. It creates multiple independent SHA-512-based workloads and organizes their outputs through a Merkle-tree reduction, allowing the workloads to be distributed across the processing units available to the defender. The objective of these constructions is to exploit the computational resources available on modern CPUs and GPUs in order to increase the amount of hashing work performed for each password candidate while limiting the corresponding increase in execution latency.

The use of parallelism in password hashing is not unique to the constructions proposed in this thesis. In particular, Argon2 exposes a configurable parallelism parameter that divides its computation into multiple lanes, which may be processed concurrently [19]. It therefore shares the general objective of exploiting the parallel resources available to the defender. The security goals of the two approaches are nevertheless different. Argon2 is primarily a memory-hard password-hashing construction designed to impose both computational and memory costs on an attacker. The algorithms developed in this thesis do not attempt to provide memory hardness. Instead, they use a Merkle-tree organization of independent SHA-512 computations to increase the amount of computational work performed for each password candidate while limiting the corresponding increase in verification latency.

Other modern hashing constructions, including ParallelHash and BLAKE3, also employ tree-based organizations to enable parallel processing [21, 22]. These constructions divide the processing of a large input into independent branches in order to reduce the time required to compute a digest and improve the hashing throughput. Although the proposed algorithms use a similar tree-based organization, their objective is fundamentally different. Rather than accelerating the processing of a fixed amount of input data, they deliberately increase the amount of work associated with each password candidate and use parallel execution to limit the resulting authentication latency. Thus, the common tree structure serves different performance goals: throughput improvement in ParallelHash and BLAKE3, and per-candidate cost amplification in the constructions proposed in this thesis.

These comparisons clarify the position of the proposed approach: it is a computationally intensive password-hashing construction that uses tree-based parallelism to increase per-candidate work rather than simply to accelerate a fixed hashing workload.

The remainder of the chapter first introduces the main idea underlying the proposed approach, describing how a hashing workload can be divided into independent Merkle subtrees and assigned to separate processing units. Then, the behavior of `parallel_sha512`, `parallel_sha512_it`, and their variants is formalized through pseudocode. The subsequent sections describe the C++ and OpenCL implementations. The CPU discussion focuses on the adopted optimization techniques, whereas the GPU discussion examines how the independent computations are mapped to OpenCL work-items.

2.1 Merkle-tree-based parallel hashing approach

The proposed multithreaded hashing algorithm is based on a standard Merkle tree construction [4, 5]. The underlying idea is described in the following list.

1. Given a password *pwd* and a parameter N, we compute SHA_512$(pwd, salt, i)$ for $i = 1, \ldots, N$ (the hash is computed on the concatenation of the three strings *pwd*, *salt*, *i*).

2. We organize the hashes as leaves of an incomplete binary tree and we compute the Merkle tree root hash. More precisely, given a list $h_1, \ldots, h_n$ where initially $n = N$, we construct a new list $h'_1, \ldots, h'_{\lceil n/2 \rceil}$ such that $h'_i = \text{SHA\_512}(h_{2i-1}, h_{2i})$ for $i = 1, \ldots, \lfloor n/2 \rfloor$ and, if $\lceil n/2 \rceil \neq \lfloor n/2 \rfloor$, then $h'_{\lceil n/2 \rceil} = h_n$. We iterate until we obtain a list with a single hash which is given as output.

Throughout this document, this algorithm will be referred to as `parallel_sha512`. The use of the SHA-512 hash function is not mandatory; the algorithm can be easily adapted to employ any desired collision-resistant hash function.

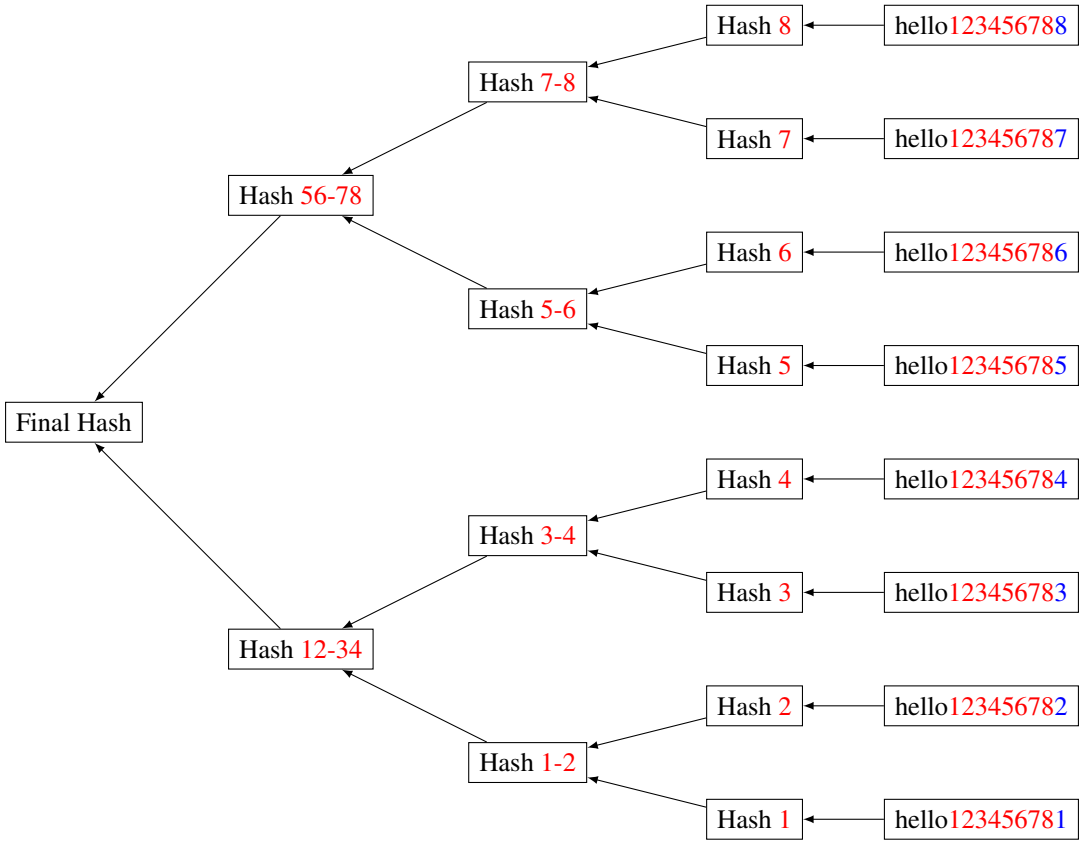


Figure 2.1: Merkle tree example with `parallel_sha512`.

Figure 2.1 represents the Merkle tree resulting from the execution of the `parallel_sha512` algorithm with *pwd* = *hello*, *salt* = 12345678 (highlighted in red), $i = 1, \ldots, 8$ (highlighted in blue) as input.

Considering the Merkle tree in Figure 2.1, the parallelization strategy adopted in this work consists of subdividing the tree into multiple independent subtrees, whose computation is assigned to separate worker threads corresponding to different CPU cores. Once all threads complete their computations, their results are processed by the main thread, which computes a reduced upper-level Merkle tree and returns the final hash. Two possible subdivisions of the Merkle tree in Figure 2.1 are described below:

- *Two-subtree subdivision*. The tree is divided into:

  1. a subtree rooted at `Hash 12-34`;
  2. a subtree rooted at `Hash 56-78`;
  3. a reduced tree composed of the nodes (`Final Hash`, `Hash 12-34`, `Hash 56-78`).

  The two subtrees (1–2) are computed by two separate worker threads, while the reduced tree (3) is computed by the main process. All involved processes execute the `parallel_sha512` algorithm on their respective subtree. This subdivision yields two subtrees, each containing four leaves.

- *Four-subtree subdivision*. The tree is divided into:

  1. a subtree rooted at `Hash 1-2`;
  2. a subtree rooted at `Hash 3-4`;
  3. a subtree rooted at `Hash 5-6`;
  4. a subtree rooted at `Hash 7-8`;
  5. a reduced tree composed of the nodes (`Final Hash`, `Hash 1-2`, `Hash 3-4`, `Hash 5-6`, `Hash 7-8`).

  Subtrees (1-4) are computed by four separate worker threads, while the reduced tree (5) is computed by the main process. All involved processes execute the `parallel_sha512` algorithm on their respective subtree. This subdivision produces four subtrees, each containing two leaves.

The subtree subdivision is specified through the program input. This allows the user to choose the number of worker threads (and thus subtrees) and the number of leaves per subtree. The specified number of leaves is the same for every subtree, ensuring that all subtrees contain the same number of leaves and, therefore, nodes.

This subdivision is not intended only to reduce the execution time of a fixed workload but also to allow the defender to increase the total amount of work performed for each password candidate while distributing that work across the available processing units. This objective can be expressed through the following idealized model.

Parallel cost-amplification model. Let p denote the number of processing units available to the defender, and let N denote the number of SHA-512 evaluations performed by a traditional sequential password-hashing scheme. Evaluating one password candidate with the sequential scheme requires approximately N SHA-512 computations both for the defender and for an attacker. The proposed approach creates p independent workloads, each performing approximately N SHA-512 evaluations, and assigns them to p processing units. The total amount of work performed for one candidate therefore becomes approximately pN. Under ideal conditions, these workloads are processed concurrently. Consequently, the elapsed verification time approaches the cost of one N-evaluation workload, plus the overhead associated with thread management, synchronization, and the final Merkle-tree reduction. The construction therefore aims to increase the total computational work by a factor close to p, while producing an acceptable increase in authentication latency. This behavior is formalized in Section 3.1 and compared with the experimental results in Section 3.2. This model describes the potential advantage available to the defender, but it also raises an important question: whether an attacker can obtain the same benefit by using parallel hardware.

Attacker model. An attacker may also have access to highly parallel hardware. However, password-cracking systems typically use this parallelism to evaluate many independent password candidates simultaneously. Increasing the work required for each candidate consumes a larger fraction of the available computational resources and consequently reduces the number of candidates that can be processed per unit of time. The proposed construction is therefore not intended to prevent an attacker from using parallel execution. Its objective is to transform server-side parallelism into additional per-candidate work, thereby reducing the attacker's password-guessing throughput, which is evaluated experimentally through the Hashcat integration presented in Chapter 4.

Algorithm variants. To investigate this design across different workload organizations and execution models, several related algorithms were developed:

- `parallel_sha512`: multithread hashing algorithm based on Merkle trees, as described in Step 1 and Step 2 [4, 5];
- `parallel_sha512_it`: multithread hashing algorithm in which each worker thread performs iterative hashing, and the partial results are then combined using the Merkle tree reduction [4, 5];
- `parallel_sha512_it_crypt`: multithread hashing algorithm similar to `parallel_sha512_it`, in which the iterations performed by each worker thread are based on the `sha512crypt` algorithm [8];
- `sha512_it`: a sequential iterated version of SHA-512, used as a comparison baseline [2];

- `sha512crypt`: the standard iterative password-hashing scheme used as an additional baseline [8, 11].

2.2 Pseudocode of the proposed algorithms

This section presents and describes the pseudocode of the `parallel_sha512`, `parallel_sha512_it` and `parallel_sha512_it_crypt` algorithms.

2.2.1 `parallel_sha512` algorithm pseudocode

The `parallel_sha512` algorithm can be divided into two procedures: SubtreeHash contains the code that will be executed by each core or worker thread, ParallelSHA512 is the code that will be executed by the main thread. This algorithm assumes that the number of leaves assigned to each subtree satifies the condition $n_{\text{leaves}} > 2$, so that each subtree always performs at least one pairwise reduction steps.

Algorithm 1 SubtreeHash (computed by a worker thread/core).

Require: Password *pwd*, salt *salt*, number of leaves $n_{leaves} > 2$, starting index *start*
Output: subtree root hash (64-byte array)

1 $results \leftarrow [\,]$
2 $ctrBytes \leftarrow \text{BE32}(start)$ ▷ 4-byte big-endian encoding

3 **for** $i \leftarrow 0$ **to** $n\_leaves - 1$ **do**
4 $\quad ctrBytes \leftarrow \text{BE32}(start + i + 1)$
5 $\quad results[i] \leftarrow \text{SHA512}(base \,\|\, ctrBytes)$
6 **end for**

7 $active \leftarrow n\_leaves$
8 **while** $active > 2$ **do**
9 $\quad$ **for** $j \leftarrow 0$ **to** $\lfloor active/2 \rfloor - 1$ **do**
10 $\quad\quad results[j] \leftarrow \text{SHA512}(results[2j] \,\|\, results[2j+1])$
11 $\quad$ **end for**
12 $\quad$ **if** *active* is odd **then**
13 $\quad\quad results[\lfloor active/2 \rfloor] \leftarrow results[active - 1]$ ▷ carry last
14 $\quad\quad active \leftarrow \lfloor active/2 \rfloor + 1$
15 $\quad$ **else**
16 $\quad\quad active \leftarrow active/2$
17 $\quad$ **end if**
18 **end while**

19 **return** $\text{SHA512}(results[0] \,\|\, results[1])$

SubtreeHash procedure description

- **Leaf construction**. Each worker thread computes a subtree hash of the complete Merkle tree. The subtree computation starts by building *n_leaves* leaf hashes. For the *i*-th leaf, the SHA-512 hash is generated by the fixed sequence $base = pwd \,\|\, salt$, concatenated with a 32-bit (4 bytes) counter encoded in big-endian. This encoding avoids the more expensive conversion of an integer index into its decimal ASCII representation, yielding constant-size appends and it follows the specification at Step 1.
- **Incremental hashing**. In the actual implementation, SHA-512 is computed incrementally using the following OpenSSL primitives: `SHA512_Init/Update/Final`; by first updating with *base* and then with the 4-byte counter. This is equivalent to hashing the complete concatenation.

- **Subtree reduction**. The list of leaf hashes is reduced with a Merkle-tree style fold: leaf hash pairs are concatenated and hashed ($SHA512(left \,\|\, right)$), and when the number of active hashes at a given level is odd, the last hash is carried forward to the next level of the Merkle tree unchanged. The reduction process repeats until only two hashes remain, which are finally concatenated and hashed to obtain the subtree root, which is then returned by the worker thread and follows the specification at Step 2.

Algorithm 2 ParallelSHA512 (Merkle-tree reduction across threads).

Require: Password *pwd*, salt *salt*, number of cores *n_cores*, number of leaves $n_{leaves} > 2$

Output: final hash (64-byte array)

```
1  base ← pwd || salt
2  hashes ← array of size n_cores

3  for all c ← 0 to n_cores − 1 in parallel do
4      start ← c · n_leaves                          ▷ uint32 offset for leaf counters
5      hashes[c] ← SubtreeHash(base, n_leaves, start)
6  end for

7  active ← n_cores
8  while active > 1 do
9      for i ← 0 to ⌊active/2⌋ − 1 do
10         hashes[i] ← SHA512(hashes[2i] || hashes[2i + 1])
11     end for
12     if active is odd then
13         hashes[⌊active/2⌋] ← hashes[active − 1]
14         active ← ⌊active/2⌋ + 1
15     else
16         active ← active/2
17     end if
18 end while

19 return hashes[0]
```

ParallelSHA512 procedure description

- **Initialization**. The *base* byte array is set to *pwd* || *salt* and the *hashes* array, containing all partial results produced by the worker threads, is initialized.
- **Parallel execution**. The threads are executed in parallel performing the SubtreeHash procedure.
- **Global reduction**. The main thread performs the same reduction process as in Step 2 starting from the *n_cores* subtree roots, which are now considered as the new leaves of the Merkle tree, to obtain the final digest.

2.2.2 `parallel_sha512_it` algorithm pseudocode

For the iterative version of the `parallel_sha512` algorithm, the SubtreeHash procedure is simplified. In addition to the *base* string, it requires the number of iterations *rounds* to perform and the core/ worker thread number.

Algorithm 3 SUBTREEHASH_IT (computed by a worker thread/core).

Input: *base*, *rounds*, *number*

Output: hash (64-byte array)

1 $id \leftarrow \text{BE32}(number + 1)$
2 $hash \leftarrow \text{SHA512}(base \,\|\, id)$

3 **for** $i \leftarrow 1$ **to** $rounds - 1$ **do**
4 $\quad hash \leftarrow \text{SHA512}(hash)$
5 **end for**

6 **return** *hash*

The worker procedure for `parallel_sha512_it` differs from SUBTREEHASH in Algorithm 2 because each thread computes a single iterative hash instead of a Merkle-tree reduction of a subtree. It computes *rounds* hash iterations starting from the password and salt concatenated with the core/thread number. The resulting digest is then hashed repeatedly for the remaining rounds. Finally, the main thread combines the partial hashes using the same Merkle reduction adopted by `parallel_sha512`.

Algorithm 4 PARALLELSHA512_IT.

Input: *pwd* (byte array), *salt* (byte array), *n_cores* (int), *n_rounds* (int)

Output: final hash (64-byte array)

1 $base \leftarrow pwd \,\|\, salt$
2 $hashes \leftarrow$ array of size *n_cores*

3 **for all** $c \leftarrow 0$ **to** $n\_cores - 1$ **in parallel do**
4 $\quad hashes[c] \leftarrow \text{SUBTREEHASH}(base, n\_rounds, c)$
5 **end for**

6 $active \leftarrow n\_cores$
7 **while** $active > 1$ **do**
8 $\quad$ **for** $i \leftarrow 0$ **to** $\lfloor active/2 \rfloor - 1$ **do**
9 $\quad\quad hashes[i] \leftarrow \text{SHA512}(hashes[2i] \,\|\, hashes[2i+1])$
10 $\quad$ **end for**
11 $\quad$ **if** *active* is odd **then**
12 $\quad\quad hashes[\lfloor active/2 \rfloor] \leftarrow hashes[active - 1]$
13 $\quad\quad active \leftarrow \lfloor active/2 \rfloor + 1$
14 $\quad$ **else**
15 $\quad\quad active \leftarrow active/2$
16 $\quad$ **end if**
17 **end while**

18 **return** $hashes[0]$

2.2.3 `parallel_sha512_it_crypt` algorithm pseudocode

The `parallel_sha512_it_crypt` algorithm follows the same global reduction structure as `parallel_sha512_it`. The main difference is that each worker thread computes the iterated digest using the `sha512crypt` hash function passing the specified number of rounds as input parameter. The resulting string is hashed one more time with SHA-512 to obtain a fixed-size 64-byte digest suitable for the general Merkle-tree reduction.

2.3 C++ hashing algorithm implementation

This section describes the main the C++ implementation choices and optimizations adopted for the proposed algorithms. The complete source code of all algorithms is available in the Appendix A. This section describes the main characteristics of the implementations and explains the motivations behind them.
All multithreaded hashing algorithm implementations are based on two functions:

- `main`: the function executed by the main thread. It is responsible for command-line parameter parsing, thread invocation, final Merkle tree reduction, and final digest printing.
- `parallel_hash`: the function executed by each core/thread. Its implementation varies depending on the algorithm considered. Generally, it has the following parameters:
  - `str`: the password and salt concatenation;
  - `str_len`: the number of characters of `str`;
  - `size`: the number of leaves of each core/thread;
  - `start`: the value from where the leaf numbering starts, this parameter is not present on the `parallel_sha512` iterative versions;
  - `number`: the core/thread number, it is used to store the final core/thread result in the correct position of the `hashes` array;
  - `hashes`: an array used to store all the core/thread partial results that will be combined to compute the final digest by the main process.

2.3.1 `parallel_sha512` C++ implementation

The `parallel_sha512` C++ implementation reflects very closely the pseudocode at Algorithm 2 and Algorithm 1, with additional implementation details introduced to improve performance. The implementation assumes that each subtree contains the same number of leaves and that $n_{leaves} > 2$, as specified in the pseudocode.

Optimization 1 - std::array

`std::array` is a C++ container that encapsulates fixed-size arrays. It has the same semantics as a struct holding a C-style array. `std::array` objects are used to store SHA-512 digests inside `std::vector` container in order to avoid dynamic memory allocations for each digest. This optimization is also applied in the main function.

```
1 vector<array<unsigned char, SHA512_DIGEST_LENGTH>> results(size);
```

Listing 2.11: Example of `std::array` usage in the `parallel_hash` function.

Optimization 2 - Compact leaf encoding

Each leaf has a unique number associated to it to make it distinguishable from other leaves. In the implementation, leaf identifiers are encoded directly into a 4-byte array through bit-shifting operations, avoiding the overhead associated with integer-to-string conversions and making this append operation constant time. Since the identifier is represented using 32 bits, each subtree can contain at most $2^{32} - 1$ distinct leaf identifiers, assuming identifiers start from 1. Moreover, the implementation assumes that all subtrees contain the same number of leaves. Listing 2.12 reports an example showing how this is performed in the actual implementation.

```
1 #define BYTES 4
2
```

```
void parallel_hash(const unsigned char* str, size_t str_len, unsigned int size, unsigned
    int start, unsigned int number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {
    ...
    unsigned char bytes[BYTES];

    // Big-endian encoding (most significant byte MSB first)
    bytes[3] = start & 0xFF;
    bytes[2] = (start >> 8) & 0xFF;
    bytes[1] = (start >> 16) & 0xFF;
    bytes[0] = (start >> 24) & 0xFF;

    SHA512_CTX ctx;
    for(size_t i = 0; i < size; ++i) {
        // Increment the fixed-size binary leaf counter.
        bytes[3]++;
        if(bytes[3] == 0) {
            bytes[2]++;
            if(bytes[2] == 0) {
                bytes[1]++;
                if(bytes[1] == 0)
                    bytes[0]++;
            }
        }
        ...
    }
    ...
}
```

Listing 2.12: Compact leaf encoding example.

Optimization 3 - Avoiding string concatenation

To minimize temporary memory allocations and string copies, message concatenation is performed by incrementally feeding data to the hashing context through multiple `SHA512_Update` calls. This optimization is also applied in the main function. Listing 2.12 reports a code excerpt showing how this is performed in the actual implementation.

```
void parallel_hash(const unsigned char* str, size_t str_len, unsigned int size, unsigned
    int start, unsigned int number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {
    ...
    SHA512_CTX ctx;
    for(size_t i = 0; i < size; ++i) {
        ...
        SHA512_Init(&ctx);
        SHA512_Update(&ctx, str, str_len);
        SHA512_Update(&ctx, bytes, BYTES);
        SHA512_Final(results[i].data(), &ctx);
    }
    ...
```

```
    while(active_hashes > 2) {
        for(size_t i = 0; i < active_hashes / 2; ++i) {
            SHA512_Init(&ctx);
            SHA512_Update(&ctx, results[2 * i].data(), SHA512_DIGEST_LENGTH);
            SHA512_Update(&ctx, results[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
            SHA512_Final(results[i].data(), &ctx);
        }
        ...
    }
    ...
}
```

Listing 2.13: Avoiding string concatenation through multiple `SHA512_Update` calls in the `parallel_hash` function.

Optimization 4 - In-place reduction

The Merkle reduction is performed in-place by saving intermediate hashes inside the same `std::vector` array, reducing memory consumption. This optimization is also applied in the main function.

```
void parallel_hash(const unsigned char* str, size_t str_len, unsigned int size, unsigned
↪   int start, unsigned int number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {

    vector<array<unsigned char, SHA512_DIGEST_LENGTH>> results(size);
    ...
    size_t active_hashes = size;
    while(active_hashes > 2) {
        for(size_t i = 0; i < active_hashes / 2; ++i) {
            SHA512_Init(&ctx);
            SHA512_Update(&ctx, results[2 * i].data(), SHA512_DIGEST_LENGTH);
            SHA512_Update(&ctx, results[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
            SHA512_Final(results[i].data(), &ctx);
        }
        ...
    }
    ...
}
```

Listing 2.14: Example of in-place reduction in the `parallel_hash` function.

The complete source code is available at Appendix A.1.

2.3.2 `parallel_sha512_it` C++ implementation

For the `parallel_sha512_it` algorithm, the main differences lie in the `parallel_hash` function reported in Listing 2.15. The following implementation of the `parallel_hash` function follows very closely Algorithm 3, and applies the same optimizations discussed for the `parallel_hash` algorithm.

```
void parallel_hash(unsigned char* str, size_t str_len, unsigned int rounds, unsigned int
↪   number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {
```

```
3
4       char bytes[4];
5       int n = number + 1;
6       // Little-endian encoding (least significant byte LSB first)
7       bytes[3] = n & 0xFF;                // 8 bits least significant
8       bytes[2] = (n >> 8) & 0xFF;         // intermediate byte
9       bytes[1] = (n >> 16) & 0xFF;        // intermediate byte
10      bytes[0] = (n >> 24) & 0xFF;        // most significant byte
11
12      SHA512_CTX ctx;
13      SHA512_Init(&ctx);
14      SHA512_Update(&ctx, str, str_len);
15      SHA512_Update(&ctx, bytes, 4);
16      SHA512_Final(hashes[number].data(), &ctx);
17
18      for (unsigned long i = 1; i < rounds; ++i) {
19          SHA512_Init(&ctx);
20          SHA512_Update(&ctx, hashes[number].data(), SHA512_DIGEST_LENGTH);
21          SHA512_Final(hashes[number].data(), &ctx);
22      }
23  }
```

Listing 2.15: `parallel_sha512_it` `parallel_hash` function.

The complete source code is available at Appendix A.2.

2.3.3 `parallel_sha512_it_crypt` C++ implementation

For the `parallel_sha512_it_crypt` algorithm the `parallel_hash` function, reported in Listing 2.16 constructs the salt string as required by the `sha512crypt` algorithm and defined in Equation 1.8, calls the `crypt` library to execute the `sha512crypt` algorithm, and then hashes the result using standard SHA-512 one more time. This is necessary to obtain a fixed-size 64-byte digest that can be used for the final Merkle tree reduction that will be performed by the main thread.

```
1   void parallel_hash(const char* pwd_str, const char* salt_str, unsigned int rounds,
     ↪  unsigned int number,
2           vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {
3
4       string salt = "$6$rounds=" + to_string(rounds) + "$" + salt_str;
5       crypt_data data{};
6       const char* hash = crypt_r(pwd_str, salt.c_str(), &data);
7       if(hash == nullptr) {
8           perror("crypt_r");
9           return;
10      }
11      SHA512(reinterpret_cast<const unsigned char*>(hash), strlen(hash),
         ↪  hashes[number].data());
12  }
```

Listing 2.16: `parallel_sha512_it_crypt` `parallel_hash` function.

The complete source code is available at Appendix A.3.

2.3.4 Iterative algorithms C++ implementation

For the `sha512_it` and `sha512crypt` algorithms, the C++ program implementing them is very straightforward. It handles command-line parameter parsing and then performs the iterations by calling the OpenSSL SHA-512 primitives inside a for loop for `sha512_it` or invoking the `crypt` library for `sha512crypt`.

2.4 OpenCL hashing algorithm implementation

This section presents the OpenCL implementation of the `parallel_sha512`, the `parallel_sha512_it` and the `sha512_it` algorithms. The OpenCL implementation follows the same high-level structure described in the pseudocode in Section 2.2, with the key difference that the computation of independent parts is assigned to different OpenCL work-items. Since the algorithmic behavior has already been presented through pseudocode and through the C++ CPU implementation, this section focuses on the main GPU-related implementation choices rather than on a line-by-line explanation of the source code.

2.4.1 Host-side structure

The host program is responsible for parsing the input parameters, preparing the password-salt concatenation buffer, compiling the OpenCL kernel, allocating the input and output buffers, executing the kernel, and reading the resulting partial hashes from the device. Each work-item computes one independent portion of the tree. Depending on the specific algorithm, a different OpenCL kernel is compiled and executed. For the parallel algorithms, the global size of the kernel launch determines the number of independent work-items, and therefore the number of partial digests computed by the GPU. The resulting digests are stored into a global output buffer located in the global memory. Finally, the host program transfers back the results from device memory to host memory, performs the final Merkle-tree reduction among the partial hashes and prints the final digest. Initially, the output format was the same as the one adopted by the CPU implementation. However, the integration of the algorithms into the hashcat framework required a different encoding format, as discussed in Section 4.2. Listing 2.17 shows how `n_subtrees` work-items are launched in the GPU.

```
err = clEnqueueNDRangeKernel(
    queue, kernel, 1, nullptr, &n_subtrees, nullptr, 0, nullptr, nullptr
);
```

Listing 2.17: Kernel launch using `n_subtrees` as the global work size.

2.4.2 Device-side structure

Three separate OpenCL kernels are implemented, one for each algorithm. Each kernel receives as input the password-salt buffer, the algorithm specific parameters, and an global output buffer where the partial results are stored. In the parallel algorithms, the `get_global_id(0)` function is used to retrieve the work-item's identifier. This identifier is used to determine the leaf numbering and in which position of the output array the partial results are stored.

2.4.3 OpenCL implementation of `parallel_sha512`

In the OpenCL version of the `parallel_sha512` algorithm, each work-item is responsible for the computation of one subtree root. As for the standard CPU version, the work-item computes the leaves of the subtree by hashing the password-salt input buffer concatenated with the corresponding leaf index, as in Step 1. The leaf hashes are stored in a fixed-size private memory array and are reduced through the same Merkle-tree reduction adopted in the CPU implementation and described in Step 2. At the end of the kernel execution, each work-item writes the final 512-bit subtree root to the global output buffer. The host program then reads all subtree roots and performs the final Merkle reduction on the CPU. Therefore,

the procedure described in Algorithm 2 is executed by the host program, and the procedure described in Algorithm 1 is now performed on the GPU through the OpenCL framework.

```
const u64 gid = get_global_id(0);
const u32 start = (u32) gid * size;
```

Listing 2.18: Mapping between a work-item and the first leaf of its assigned subtree.

`size` specifies the number of leaves per subtree, together with the global identifier `gid` is used to determine the subtree's leaf range.

```
const u32 base = (u32) gid * SHA512_DIGEST_LENGTH;
for(u32 i = 0; i < 8; i++) {
    const u64 word = ctx.h[i];
    const u32 out = base + i * 8;
    hashes[out + 0] = (u8) ((word >> 56) & 0xff);
    ...
    hashes[out + 7] = (u8) ((word >>  0) & 0xff);
}
```

Listing 2.19: Computation of the output offset for each work-item.

Listing 2.19 shows how result overwriting is prevented among different work-items. Since a SHA-512 digest is 64 bytes long, the correct offset value is computed by multiplying the work-item identifier by `SHA512_DIGEST_LENGTH`. This assures that each work-item stores its digest into a different region of the output buffer. Furthermore, the same byte-shifting technique described in Listing 2.12 is also adopted in the `parallel_sha512` OpenCL implementation.
The complete source code is available in Appendix A.4 and Appendix A.5.

2.4.4 OpenCL implementation of `parallel_sha512_it`

As for the OpenCL implementation of `parallel_sha512`, the OpenCL implementation of `parallel_sha512_it` assigns an independent iterative hash computation to each work-item. The work-item first hashes the base password-salt concatenated with a work-item-dependent leaf number, then repeatedly computes the SHA-512 digest for the specified number of rounds. The final digest produced by each work-item is written to the output array. The `parallel_sha512_it` host program shares the same logic as in the `parallel_sha512` implementation.
The complete source code is available in Appendix A.6 and Appendix A.7.

2.4.5 OpenCL implementation of `sha512_it`

In the OpenCL implementation of the `sha512_it` algorithm, only one work-item is launched. It executes a basic kernel that computes the iterative SHA-512 loop and writes a single final digest to the output buffer. This implementation does not exploit GPU parallelism, it is used as a GPU reference for comparison with other parallel OpenCL algorithms.

2.4.6 Design choices and limitations

To implement the algorithms in the OpenCL framework, several design choices and limitations due to the GPU nature were made and considered.

- The number of leaves per subtree is bounded by a compile-time constant, `MAX_LEAVES_PER_SUBTREE` whose value is 1024, because each kernel stores the intermediate leaf hashes in a fixed-size private array. Indeed, OpenCL C prevents dynamic memory allocations inside the kernels. Furthermore, the GPU memory generally is limited and large private array may reduce the number of work-items concurrently executable or their execution times.

- The final Merkle-tree reduction phase among subtree roots is always performed on the host (the CPU) avoiding synchronization issues among different work-items.
- The `sha512_it` kernel is launched with a single work-item because the algorithm is inherently sequential. This implementation does not benefit from GPU parallelism.
- All OpenCL kernels reuse SHA-512 primitives and macros from the hashcat OpenCL SHA-512 implementation presented in Section 1.7. This choice simplifies the implementaion and the later hashcat integration of the algorithms.

Chapter 3. Performance analysis

This chapter presents a performance analysis of the `parallel_sha512` algorithms from both theoretical and practical prespectives. The theoretical evaluation is based on complexity analysis and on a comparison of different algorithms, whereas the practical evaluation relies on execution-time measurements obtained from empirical tests.

3.1 Theoretical Analysis

In this section, a theoretical analysis of the different algorithms presented in this thesis will be conducted. Throughout the whole section, we assume that the time required to perform a single SHA-512 hash computation for a generic input, $t_{hashing}$, will be constant. All algorithms' execution times depend on $t_{hashing}$, however since all algorithms employ the same cryptographic hash function, the cost of a single hash evaluation can be assumed identical across all implementations and can therefore be omitted from the asymptotic comparison. The following analysis focuses on the dominant computational cost, namely the total number of SHA-512 evaluations required each algorithm.

3.1.1 `parallel_sha512` analysis

We start by analyzing the complexity of the `parallel_sha512` algorithm. The `parallel_sha512` algorithm has four parameters:

- pwd: the password to hash;
- $salt$: the salt to use in the hashing process;
- $n_{subtrees}$: the number of subtrees (or equivalently, worker threads) to divide the computation on;
- n_{leaves}: the number of leaves for each subtree.

To simplify the calculations and the analysis, we assume that n_{leaves} is always a power of 2. This assumption also represents the worst-case scenario, since a complete binary tree maximizes the number of internal nodes involved in the reduction process. Hence, the `parallel_sha512` execution time will be given by:

- $t_{parsing}$: constant time to parse the command line parameters;
- t_{concat}: time required to concatenate pwd and $salt$, we will assume this to be constant;
- $t_{subtree}$: time required to perform the Merkle tree reduction for a subtree (or worker thread), it can be divided into:
  - t_{init}: time required to hash each Merkle subtree leaf;
  - t_{rid}: time required to perform the Merkle tree reduction, i.e. the pairwise hashing of the lower subtree level.

  So we will have $t_{subtree} = t_{init} + t_{rid}$. Dividing in this way the times for each subtree, and assuming that each subtree will have complete binary trees, both t_{init} and t_{rid} are approximately 50% of $t_{subtree}$. This is due to the fact that, in a complete binary tree, the total number of nodes is $nodes_{tot} = nodes_{internal} + n_{leaves} = 2 \cdot n_{leaves} - 1$, because $nodes_{internal} = n_{leaves} - 1$. So we will have $\frac{n_{leaves}}{2 \cdot n_{leaves} - 1} \approx \frac{1}{2}$ for the leaves of a complete binary tree, and $\frac{n_{leaves} - 1}{2 \cdot n_{leaves} - 1} \approx \frac{1}{2}$ for the internal nodes of a complete binary tree.

  Hence, $t_{\text{subtree}} = (2 \cdot n_{leaves} - 1) \cdot t_{hashing} = O(n_{leaves})$.

- t_{final}: time required to reduce the Merkle tree having as leaves the hashes of the subtrees. This is given by the formula: $t_{final} = 2 \cdot n_{subtree} - 1$. When considering the `parallel_sha512` algorithm executed on CPUs, this time will be valued as constant since $t_{final} \in \mathbb{Z} : 1 \leq t_{final} \leq 255$, given that modern CPUs will rarely exceed 128 cores.
- t_{print}: constant time to print the final hash.

Summing all the times we have:

$$\begin{aligned} T_{\text{PARALLEL\_SHA512}}(n_{leaves}) &= t_{parsing} + t_{concat} + t_{subtree} + t_{final} + t_{print} \\ &= t_{parsing} + t_{concat} + (2 \cdot n_{leaves} - 1) \cdot t_{hashing} + t_{final} + t_{print} \\ &= O((2 \cdot n_{leaves} - 1) \cdot t_{hashing}) \\ &= O(n_{leaves}). \end{aligned} \tag{3.1}$$

Since $t_{hashing}$ is assumed constant, it can be omitted from the asymptotic notation. Assuming perfect parallelism, the subtree computation time contributes only once to the total execution time since the subtree computations are assumed to be executed concurrently.

3.1.2 `parallel_sha512_it` analysis

The `parallel_sha512_it` algorithm has the following parameters:

- *pwd*: the password to hash;
- *salt*: the salt to use in the hashing process;
- $n_{subtrees}$: the number of subtrees (or worker threads) to divide the computation on;
- n_{rounds}: the number of rounds to perform by each subtree.

The `parallel_sha512_it` execution time will be given by:

- $t_{parsing}$: constant time to parse the command line parameters;
- t_{concat}: time required to concatenate *pwd* and *salt*, we will assume this to be constant;
- t_{rounds}: time required to perform n_{rounds} hashes for each worker thread (or subtree). To compare `parallel_sha512_it` with `parallel_sha512`, n_{rounds} has to expressed in terms of n_{leaves}. Indeed, to perform a correct comparison between the two algorithms, the number of rounds performed by each worker thread in the `parallel_sha512_it` algorithm has to be equal to the total number of hashes computed by each subtree of the `parallel_sha512` algorithm, which corresponds to the total number of nodes of each Merkle subtree. The fair comparison relies on the fact that both algorithms perform the same number of hashes. Since in the `parallel_sha512` algorithm each subtree computes n_{leaves} leaf hashes and $n_{leaves} - 1$ internal hashes, then

  $$n_{rounds} = 2 \cdot n_{leaves} - 1. \tag{3.2}$$

  Therefore, $t_{rounds} = n_{rounds} \cdot t_{hashing} = (2 \cdot n_{leaves} - 1) \cdot t_{hashing} = O(n_{leaves})$.
- t_{final}: time required to reduce the Merkle tree having as leaves the hashes of the subtrees. This is given by the formula: $t_{final} = 2 \cdot n_{subtree} - 1$. This time will be valued as constant, as for the `parallel_sha512` algorithm.
- t_{print}: constant time to print the final hash.

Summing all the times we have:

$$\begin{aligned} T_{\text{PARALLEL\_SHA512\_IT}}(n_{rounds}) &= t_{parsing} + t_{concat} + t_{rounds} + t_{final} + t_{print} \\ &= t_{parsing} + t_{concat} + n_{rounds} \cdot t_{hashing} + t_{final} + t_{print} \\ &= O(n_{rounds} \cdot t_{hashing}) \\ &= O(n_{rounds}), \end{aligned}$$

assuming that the number of rounds for each core will dominate. Assuming a perfect parallelism, each subtree is processed by a dedicated core and for this reason $n_{subtree}$ will not affect the asymptotic execution time in any of the parallel algorithm considered. In this case, the `parallel_sha512_it` will only depend by the number of rounds n_{rounds} computed by each core. Furthermore, by using the equality 3.2, the `parallel_sha512_it` execution time can be expressed with respect to the number of leaves n_{leaves}, obtaining:

$$\begin{aligned} T_{\text{PARALLEL\_SHA512\_IT}}(n_{leaves}) &= O((2 \cdot n_{leaves} - 1) \cdot t_{hashing}) \\ &= O(n_{leaves} \cdot t_{hashing}) \\ &= O(n_{leaves}). \end{aligned} \tag{3.3}$$

Since the cost of a single SHA-512 invocation is assumed constant, $t_{hashing}$ is omitted from the asymptotic notation. Under the assumption of ideal parallelism, the dominant computational cost becomes independent of n_{subtrees}.

3.1.3 Analysis of iterative algorithms

A generic iterative hashing algorithm has three parameters:

- *pwd*: the password to hash;
- *salt*: the salt to use in the hashing process;
- n_{rounds}: the number of hashing iterations to perform.

The execution time of this class of algorithms is given by:

- $t_{parsing}$: constant time to parse the command line parameters;
- t_{concat}: time required to concatenate *pwd* and *salt*, we will assume this to be constant;
- t_{rounds}: time required to repeatedly hash the input string for the number of rounds specified n_{rounds}, which can be expressed as $n_{\text{rounds}} \cdot t_{\text{hashing}}$;
- t_{print}: constant time to print the final hash.

Summing all the times we have:

$$\begin{aligned} T_{\text{ITERATIVE}}(n_{rounds}) &= t_{parsing} + t_{concat} + t_{rounds} + t_{print} \\ &= t_{parsing} + t_{concat} + n_{rounds} \cdot t_{hashing} + t_{print} \\ &= O(n_{rounds} \cdot t_{hashing}) \\ &= O(n_{rounds}). \end{aligned} \tag{3.4}$$

For comparison purposes, it will be useful to express the execution time of iterative algorithms in terms of the number of leaves n_{leaves} of a complete binary tree to enable a fair comparison between the different algorithms considered in the analysis. We require all algorithms to compute the same total number of hashes, so the number of rounds of an iterative algorithm has to be equal to the total number of nodes of the entire Merkle tree of the `parallel_sha512` algorithm. Since iterative algorithms do not distribute the computation across multiple cores, the comparison must consider the total number of hashes computed by the entire parallel Merkle tree. When considering the entire Merkle tree, its total leaf number will be given by $n_{leaves\_tot} = n_{subtree} \cdot n_{leaves}$ which means that the total number of the tree is $nodes_{tot} = 2 \cdot n_{leaves_{tot}} - 1 = 2 \cdot n_{subtree} \cdot n_{leaves} - 1$, obtaining

$$\begin{aligned} T_{\text{ITERATIVE}}(n_{subtree}, n_{leaves}) &= O\big(t_{hashing} \cdot (2 \cdot n_{subtree} \cdot n_{leaves} - 1)\big) \\ &= O(n_{subtree} \cdot n_{leaves}). \end{aligned} \tag{3.5}$$

Since the cost of a single SHA-512 evaluation is assumed constant, t_{hashing} is omitted from the asymptotic notation. Unlike the parallel approaches, the execution time of iterative algorithms scales linearly with both the subtree size and the number of subtrees, since no parallel workload distribution is performed.

3.1.4 Theoretical comparison of the algorithms

Table 3.1 summarizes the theoretical execution complexity of the analyzed algorithms.

Algorithm	Asymptotic execution time	Dominant operation count
`iterative algorithms`	$O(n_{leaves} \cdot n_{subtree})$	$O(2 \cdot n_{leaves} \cdot n_{subtree} - 1)$
`parallel_sha512`	$O(n_{leaves})$	$O(2 \cdot n_{leaves} - 1)$
`parallel_sha512_it`	$O(n_{leaves})$	$O(2 \cdot n_{leaves} - 1)$

Table 3.1: Algorithm execution times comparison.

Analyzing the results shown in Table 3.1, we can conclude that:

- from a theoretical point of view, no asymptotic execution-time difference is expected between `parallel_sha512` and `parallel_sha512_it`, since both have the same dominant operation count;
- iterative algorithms perform a number of dominant operations proportional to $n_{leaves} \cdot n_{subtrees}$.
- the expected speed-up between iterative and parallel algorithms depends on the number of independent subtrees (or worker threads) $n_{subtrees}$ across which the computation is distributed. Hence, the theoretical speed-up of the parallel algorithms approaches $n_{subtrees}$ under ideal parallel execution assumptions with a $O(n_{leaves})$ per-thread workload.

These conclusions rely on ideal parallelism assumptions and exclude synchronization, scheduling, and memory-management overheads. As will be shown in the next section, analyzing the experimental results, these conclusions will not be fully reflected in practice.

3.2 CPU Execution Analysis

This section presents the experimental results obtained by executing the algorithms presented in the previous chapters on the CPU under different input configurations. The goal is to establish how the execution times change as the workload increases and to identify the advantage, if any, that parallel approaches provide over the iterative baselines. The results are presented both graphically and numerically: the plots allow to observe the general trend, while the tables provide the corresponding numerical values for a more precise comparison. All the experiments were performed on a physical machine with the following hardware specifications:

- **CPU**: Intel Core i7-9750H, with 6 cores and 12 threads. CPU extentions: AVX, AVX2, AES-NI.
- **RAM**: 32 GB of DDR4 RAM.
- **GPU**: NVIDIA Quadro T2000, with 4 GB of GPU dedicated memory and 1024 CUDA cores.

Each combination of cores, leaves, or iterations has been executed 100 times for each algorithm.

3.2.1 2 cores optimizations comparison

Figure 3.1 shows the average execution times of the `parallel_sha512`, `parallel_sha512_it` and `parallel_sha512_it_crypt` algorithms executed using 2 worker threads and different number of leaves. For all algorithms, the execution time increases approximately linearly with the number of leaves. As the number of leaves increases, the `parallel_sha512_it` algorithm achieves the lowest execution time.

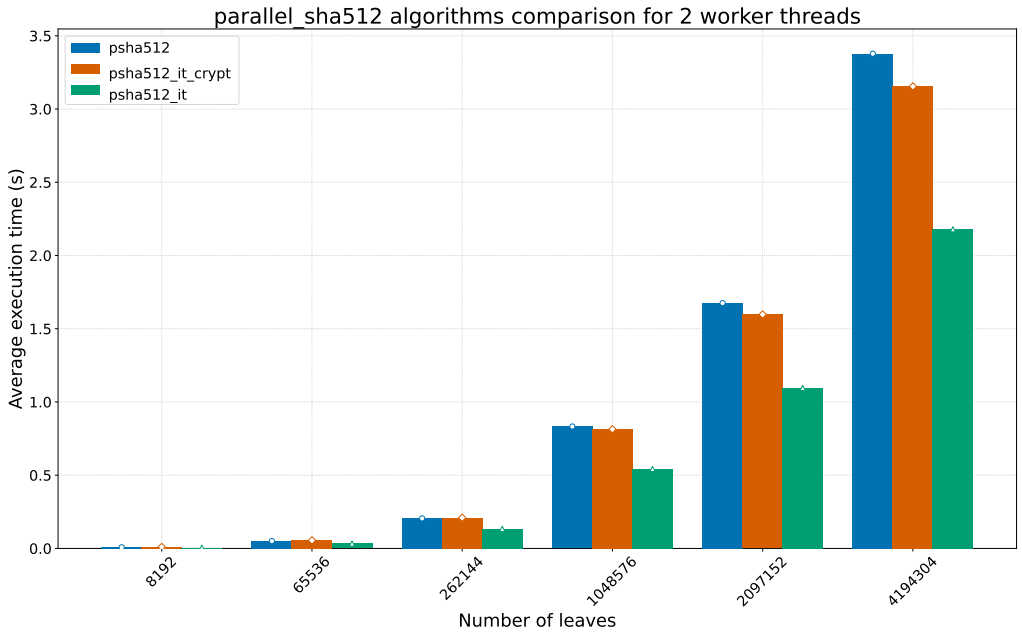


Figure 3.1: `parallel_sha512` algorithms comparison bar plot for 2 worker threads.

3.2.2 4 cores optimizations comparison

Figure 3.2 reports the same comparison using 4 worker threads to compute the subtree roots. The trend is consistent with the 2-thread case. The average execution time grows proportionally with the input size and `parallel_sha512_it` remains the fastest algorithm. The performance differences are clearer with large input configurations.

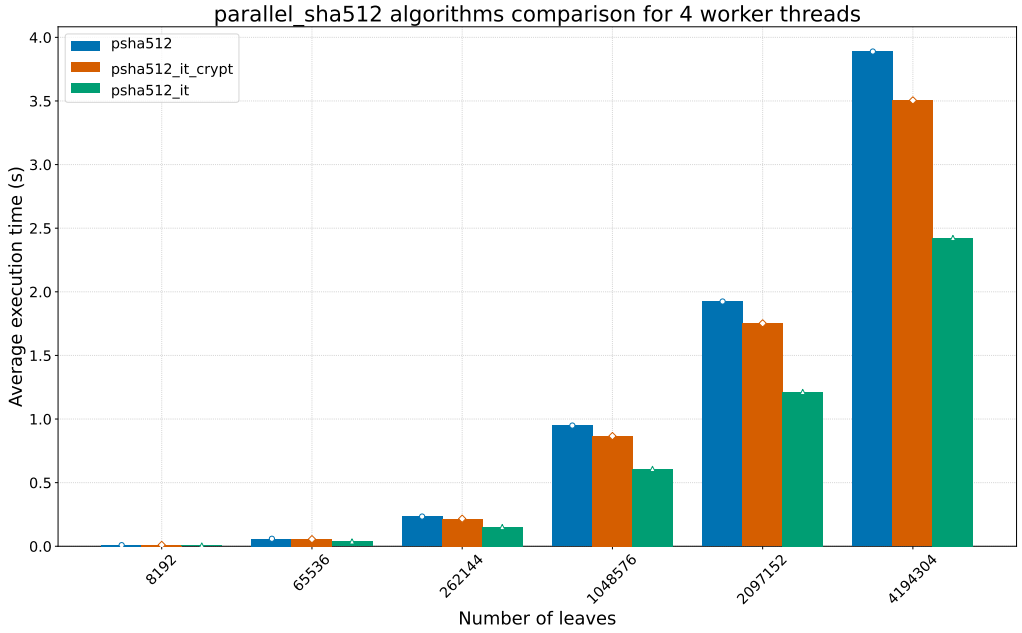


Figure 3.2: `parallel_sha512` algorithms comparison bar plot for 4 worker threads.

3.2.3 8 cores optimizations comparison

Figure 3.3 reports the execution times of the `parallel_sha512`, `parallel_sha512_it` and `parallel_sha512_it_crypt` algorithms executed with 8 worker threads and different number of leaves.

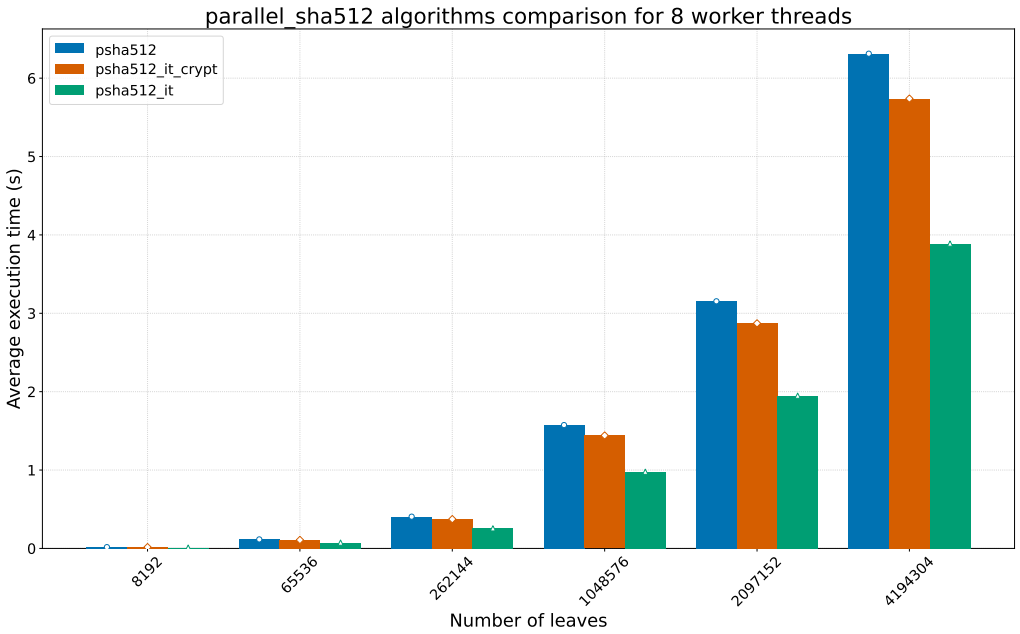
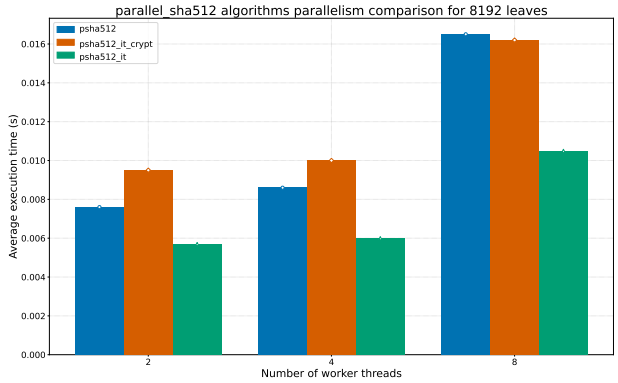


Figure 3.3: `parallel_sha512` algorithms comparison bar plot for 8 worker threads.

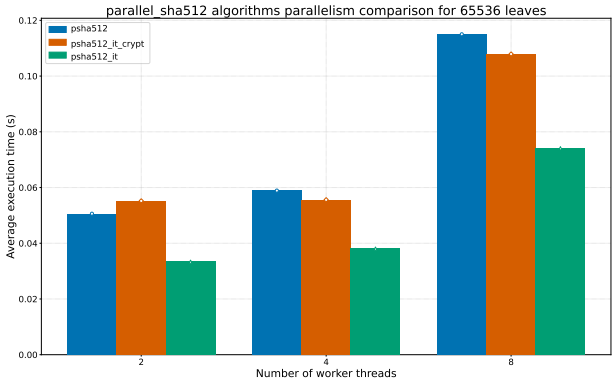
Analyzing Figure 3.1, Figure 3.2, and Figure 3.3, it is clear that the execution time increases approximately linearly: when doubling the input size also the execution time doubles. This confirms that the dominant cost is proportional to the amount of hashing work performed. Across all different input configurations, the `parallel_sha512_it` algorithm has the lowest execution times overall, making it the most promising on the parallel and iterative comparison.

3.2.4 Parallelism comparison

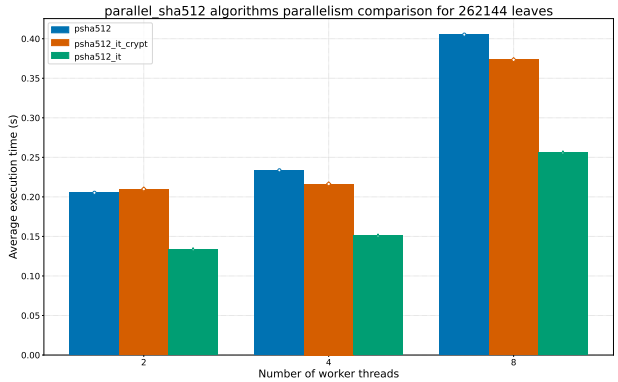
Figure 3.4 compares the behavior of the previous algorithms when executed with same number of leaves, but with different number of worker threads. The aim of this comparison is to show whether dividing the computation across different CPU cores maintains an actual parallel behavior in the average execution times.



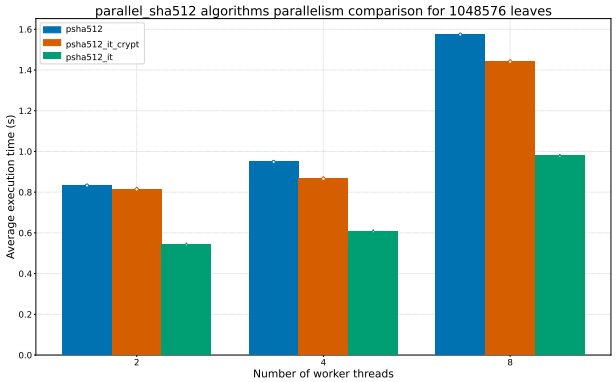
(a) `parallel_sha512` algorithms parallelism comparison bar plot for 8192 leaves.



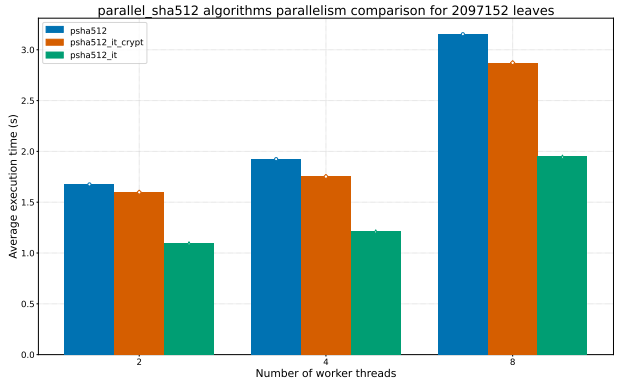
(b) `parallel_sha512` algorithms parallelism comparison bar plot for 65536 leaves.



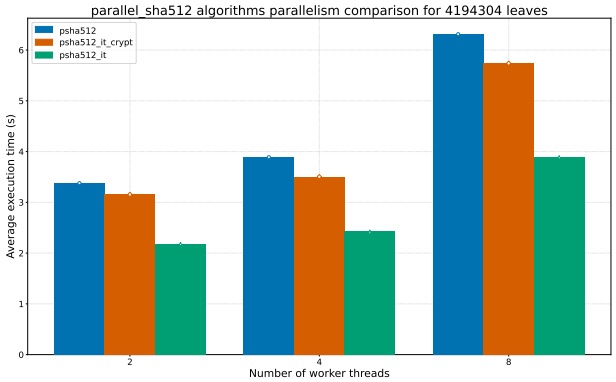
(c) `parallel_sha512` algorithms parallelism comparison bar plot for 262144 leaves.



(d) `parallel_sha512` algorithms parallelism comparison bar plot for 1048576 leaves.



(e) `parallel_sha512` algorithms parallelism comparison bar plot for 2097152 leaves.



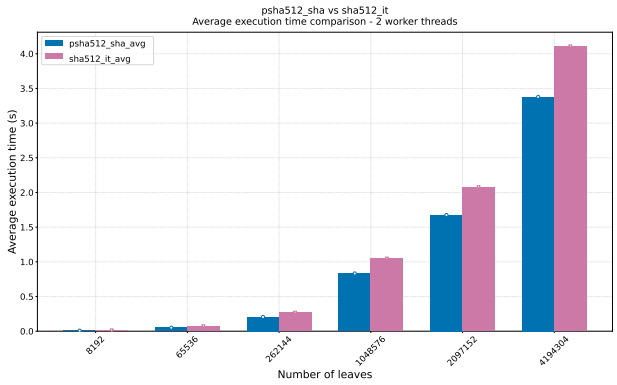
(f) `parallel_sha512` algorithms parallelism comparison bar plot for 4194304 leaves.

Figure 3.4: Parallelism comparison bar plot for different configurations of worker threads and leaves.

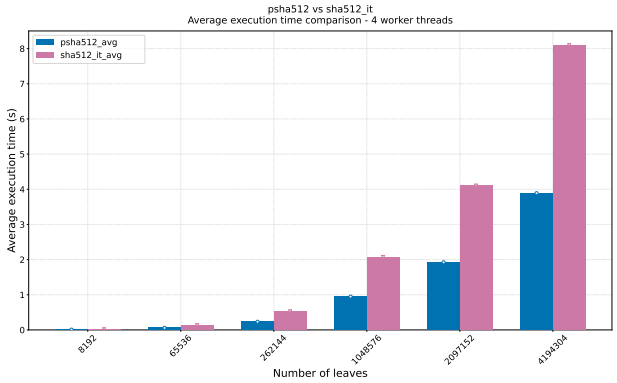
Analyzing the plot, it is clear that increasing the number of worker threads while maintaining the number of leaves fixed does not considerably affect execution times. This behavior is particularly evident for a low number of worker threads. For 8 worker threads a slight increase in the execution times recorded is noticeable, this may be mainly due to the fact that the CPU on which the algorithms where executed on has only 6 physical cores. Furthermore, a very subtle execution time increase it is also noticeable when the number of worker threads increases, however this behavior is very likely caused by the thread overhead increase and by the more expensive Merkle tree final reduction.

3.2.5 `parallel_sha512` and `sha512_it` speed-up

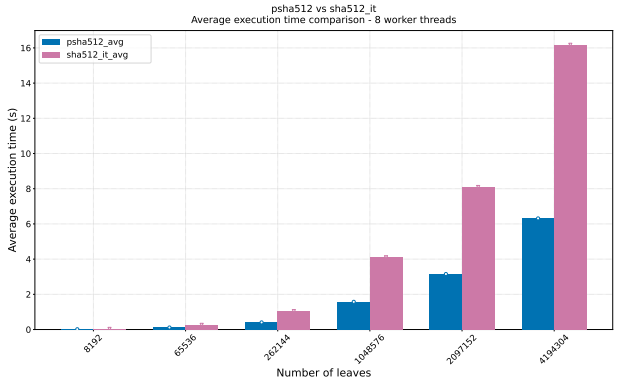
This section compares the average execution times of the `parallel_sha512` and `sha512_it` algorithms.



(a) `parallel_sha512` and `sha512_it` speed-up comparison with 2 worker threads.



(b) `parallel_sha512` and `sha512_it` speed-up comparison with 4 worker threads.



(c) `parallel_sha512` and `sha512_it` speed-up comparison with 8 worker threads.

Figure 3.5: `parallel_sha512` and `sha512_it` speed-up comparison

Figure 3.5 compares the average execution times of `parallel_sha512` and `sha512_it`. In all configurations, the parallel approach is faster than the iterative baseline. The gap increases as the number of worker increases outlining the parallelism advantage. Table 3.2, Table 3.3, and Table 3.4 report the results numerically with 2, 4, 8 worker threads respectively.

Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512` / `sha512_it` ratio
8192	0.0076	0.0104	0.7308
65536	0.0506	0.0694	0.7291
262144	0.2052	0.2659	0.7717
1048576	0.8325	1.0461	0.7958
2097152	1.675	2.0747	0.8073
4194304	3.3779	4.1054	0.8228

Table 3.2: `parallel_sha512` and `sha512_it` speed-up comparison with 2 worker threads.

Table 3.2 shows that the ratio between the two algorithms ranges from approximately 0.73 to 0.82. This means that the parallel algorithm requires about 73-82% of the execution time of the iterative version. However, the advantage is still moderate since the thread management overhead of the parallel structure has a noticeable impact.

Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512` / `sha512_it` ratio
8192	0.0086	0.0187	0.4599
65536	0.0589	0.1348	0.4369
262144	0.2336	0.5308	0.4401
1048576	0.9488	2.0747	0.4573
2097152	1.9237	4.1054	0.4686
4194304	3.8885	8.0976	0.4802

Table 3.3: `parallel_sha512` and `sha512_it` speed-up comparison with 4 worker threads.

As shown in Table 3.3, the performance advantage becomes more evident as the number of worker threads increases. The ratio is approximately between 0.44 and 0.48, therefore `parallel_sha512` takes less than half of the execution time of `sha512_it`, confirming that increasing parallelism also improves the performance.

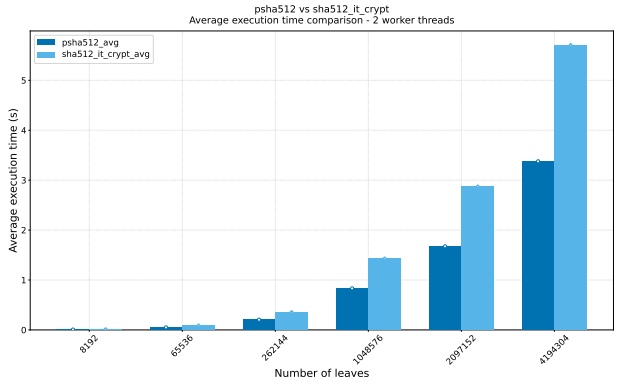
Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512` / `sha512_it` ratio
8192	0.0165	0.0355	0.4648
65536	0.115	0.2659	0.4325
262144	0.4052	1.0461	0.3873
1048576	1.5738	4.1054	0.3833
2097152	3.1531	8.0976	0.3894
4194304	6.3122	16.1773	0.3902

Table 3.4: `parallel_sha512` and `sha512_it` speed-up comparison with 8 worker threads.

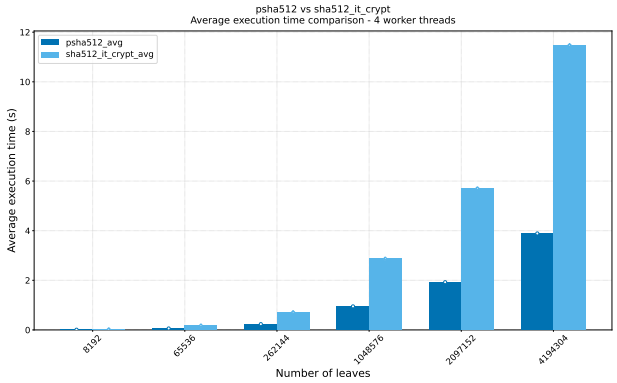
In Table 3.4, the ratio continues to decrease reaching approximately values of 0.39 for larger inputs. This corresponds to a speed-up of about 2.5× over the iterative algorithm. Even though this is a significative improvement, it is still far from the ideal 8× speed-up that would be expected.

3.2.6 `parallel_sha512` and `sha512_it_crypt` speed-up

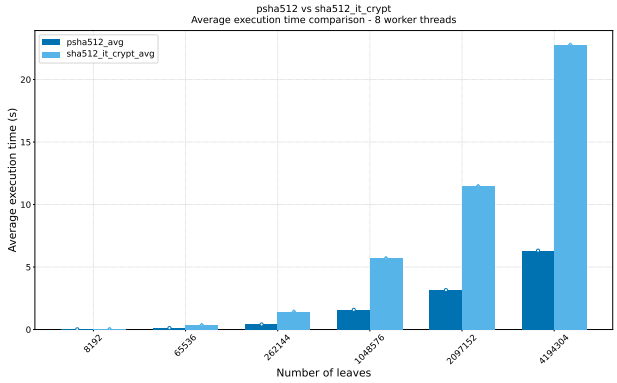
This section compares the `parallel_sha512` algorithm with the `sha512crypt` based iterative baseline.



(a) `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 2 worker threads.



(b) `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 4 worker threads.



(c) `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 8 worker threads.

Figure 3.6: `parallel_sha512` and `sha512_it_crypt` speed-up comparison

The advantage that `parallel_sha512` provides over `sha512_it_crypt` is evident with any number of worker threads.

As shown in Table 3.5, Table 3.6, and Table 3.7, the speed-up grows for a higher number of worker threads. With 4 worker threads, the ratio is close to 0.33 for most input sizes, corresponding to an approximate 3× speed-up. For 8 worker threads, as the input grows, the ratio stabilizes around 0.27-0.28, reaching a speed-up of about 3.5×.

Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512` / `sha512_it_crypt` ratio
8192	0.0076	0.0116	0.6552
65536	0.0506	0.0882	0.5737
262144	0.2052	0.3556	0.5771
1048576	0.8325	1.4297	0.5823
2097152	1.675	2.8715	0.5833
4194304	3.3779	5.7035	0.5923

Table 3.5: `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 2 worker threads.

Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512` / `sha512_it_crypt` ratio
8192	0.0086	0.0221	0.3891
65536	0.0589	0.1763	0.3341
262144	0.2336	0.7126	0.3278
1048576	0.9488	2.8715	0.3304
2097152	1.9237	5.7035	0.3373
4194304	3.8885	11.4759	0.3388

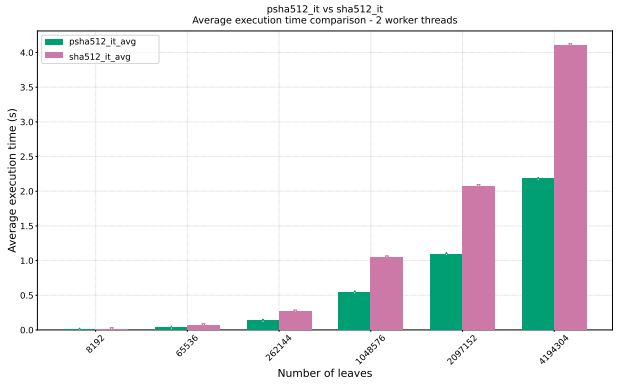
Table 3.6: `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 4 worker threads.

Number of leaves	`parallel_sha512` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512` / `sha512_it_crypt` ratio
8192	0.0165	0.0437	0.3776
65536	0.115	0.3556	0.3234
262144	0.4052	1.4297	0.2834
1048576	1.5738	5.7035	0.2759
2097152	3.1531	11.4759	0.2748
4194304	6.3122	22.7791	0.2771

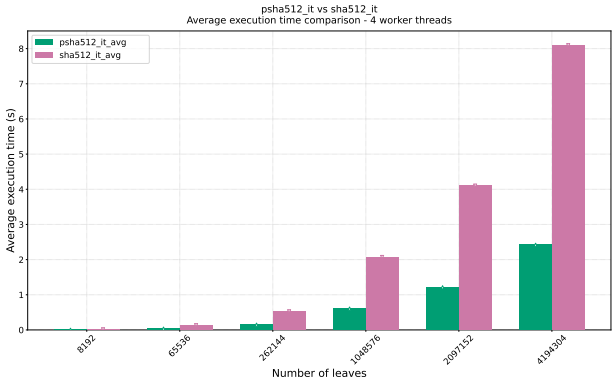
Table 3.7: `parallel_sha512` and `sha512_it_crypt` speed-up comparison with 8 worker threads.

3.2.7 `parallel_sha512_it` and `sha512_it` speed-up

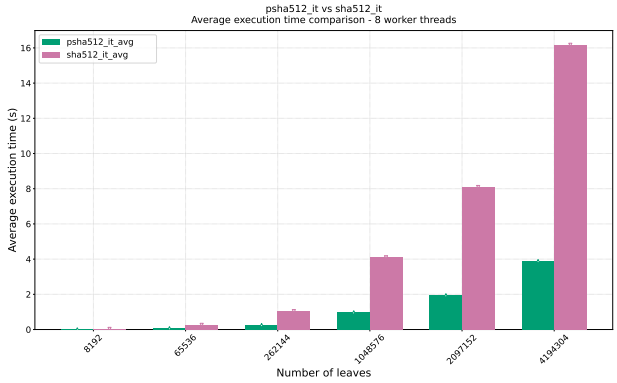
This section compares the `parallel_sha512_it` and the `sha512_it` algorithm. The results show that `parallel_sha512_it` consistently achieves lower execution times in all configurations, with a very important advantage with respect to iterative approches.



(a) `parallel_sha512_it` and `sha512_it` speed-up comparison with 2 worker threads.



(b) `parallel_sha512_it` and `sha512_it` speed-up comparison with 4 worker threads.



(c) `parallel_sha512_it` and `sha512_it` speed-up comparison with 8 worker threads.

Number of leaves	`parallel_sha512_it` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512_it` / `sha512_it` ratio
8192	0.0057	0.0104	0.5481
65536	0.0334	0.0694	0.4813
262144	0.1338	0.2659	0.5032
1048576	0.5436	1.0461	0.5196
2097152	1.0957	2.0747	0.5281
4194304	2.1792	4.1054	0.5308

Table 3.8: `parallel_sha512_it` and `sha512_it` speed-up comparison with 2 worker threads.

As shown in Table 3.8, the `parallel_sha512_it` algorithm reduces the execution times to approximately half of the iterative baseline, when executed on 2 worker threads.

Number of leaves	`parallel_sha512_it` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512_it` / `sha512_it` ratio
8192	0.006	0.0187	0.3209
65536	0.0381	0.1348	0.2826
262144	0.1512	0.5308	0.2849
1048576	0.6083	2.0747	0.2932
2097152	1.2131	4.1054	0.2955
4194304	2.4232	8.0976	0.2992

Table 3.9: `parallel_sha512_it` and `sha512_it` speed-up comparison with 4 worker threads.

With 4 worker threads, the ratio decreases to approximately 0.28-0.32, corresponding to a speed-up of about 3.5×. The algorithm benefits consistently from the increased parallelism.

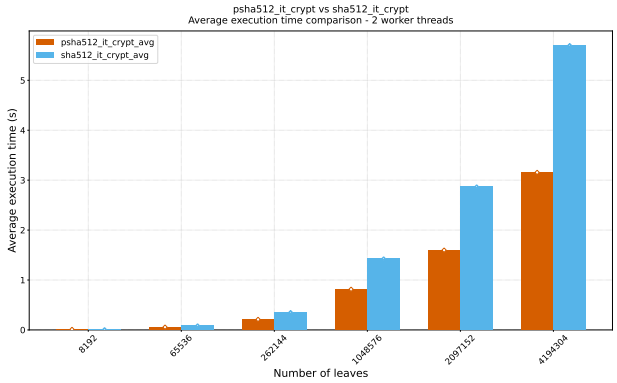
Number of leaves	`parallel_sha512_it` average execution time (s)	`sha512_it` average execution time (s)	`parallel_sha512_it` / `sha512_it` ratio
8192	0.0105	0.0355	0.2958
65536	0.0741	0.2659	0.2787
262144	0.2563	1.0461	0.245
1048576	0.9784	4.1054	0.2383
2097152	1.9487	8.0976	0.2407
4194304	3.8894	16.1773	0.2404

Table 3.10: `parallel_sha512_it` and `sha512_it` speed-up comparison with 8 worker threads.

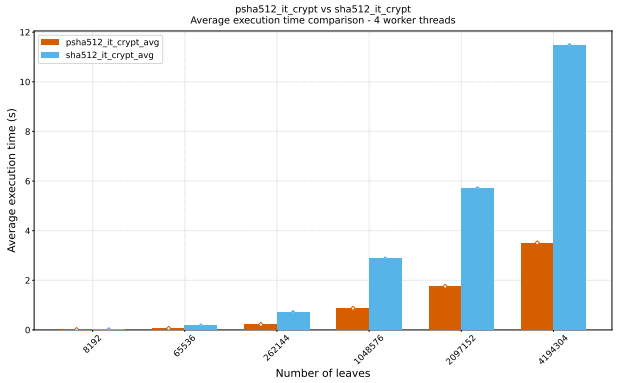
In the 8-worker thread configuration, `parallel_sha512_it` achieves the best relative results. As the workload increases, the ratio stabilizes around 0.24, corresponding to a speed-up of approximately 4×. This is still below the ideal 8× expected speed-up, however it still represents a significant practical improvement.

3.2.8 `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up

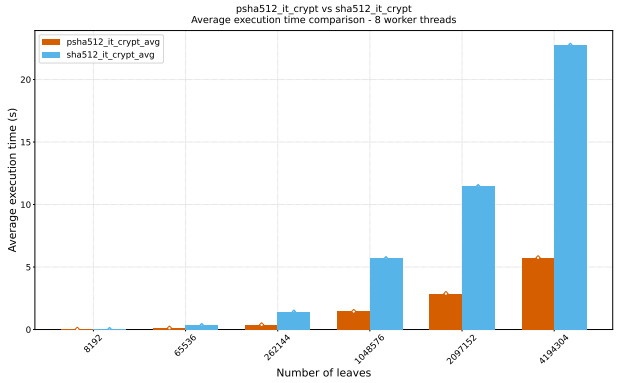
This section compares the crypt-based parallel iterative version against the standard crypt-based sequential version. This comparison evaluates whether the crypt-based iterative variant benefits from the same parallelization strategy. The results show that `parallel_sha512_it_crypt` is consistently faster than `sha512_it_crypt`. The advantage increases as the number of worker threads grows.



(a) `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 2 worker threads.



(b) `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 4 worker threads.



(c) `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 8 worker threads.

In Table 3.11, the speed-up becomes clearer as the number of leaves increases. The ratio decreases from 0.819 for the smallest input to approximately 0.553 for the largest input, suggesting that fixed overheads are more relevant for small workloads. Table 3.12, shows that with 4 worker threads, the ratio stabilizes around 0.30, achieving a 3× speed-up. With 8 worker threads, the ratio reaches approximately 0.25, meaning that the parallel `sha512crypt`-based implementation is about four times faster than the standard iterative `sha512crypt` algorithm. The speed-up reached is very similar to the `parallel_sha512_it` and `sha512_it` speed-up discussed in the previous section.

Number of leaves	`parallel_sha512_it_crypt` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512_it_crypt` / `sha512_it_crypt` ratio
8192	0.0095	0.0116	0.819
65536	0.0552	0.0882	0.6259
262144	0.2098	0.3556	0.59
1048576	0.8138	1.4297	0.5692
2097152	1.5974	2.8715	0.5563
4194304	3.1547	5.7035	0.5531

Table 3.11: `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 2 worker threads.

Number of leaves	`parallel_sha512_it_crypt` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512_it_crypt` / `sha512_it_crypt` ratio
8192	0.01	0.0221	0.4525
65536	0.0556	0.1763	0.3154
262144	0.2163	0.7126	0.3035
1048576	0.8657	2.8715	0.3015
2097152	1.752	5.7035	0.3072
4194304	3.5044	11.4759	0.3054

Table 3.12: `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 4 worker threads.

Number of leaves	`parallel_sha512_it_crypt` average execution time (s)	`sha512_it_crypt` average execution time (s)	`parallel_sha512_it_crypt` / `sha512_it_crypt` ratio
8192	0.0162	0.0437	0.3707
65536	0.1079	0.3556	0.3034
262144	0.3734	1.4297	0.2612
1048576	1.4408	5.7035	0.2526
2097152	2.8717	11.4759	0.2502
4194304	5.7392	22.7791	0.252

Table 3.13: `parallel_sha512_it_crypt` and `sha512_it_crypt` speed-up comparison with 8 worker threads.

3.2.9 `mkpasswd` and `sha512_it` comparison

The following plots compare `mkpasswd` and `sha512_it`, both in the standard SHA-512 iterative version and in the `sha512crypt`-based version, as the number of rounds increases.

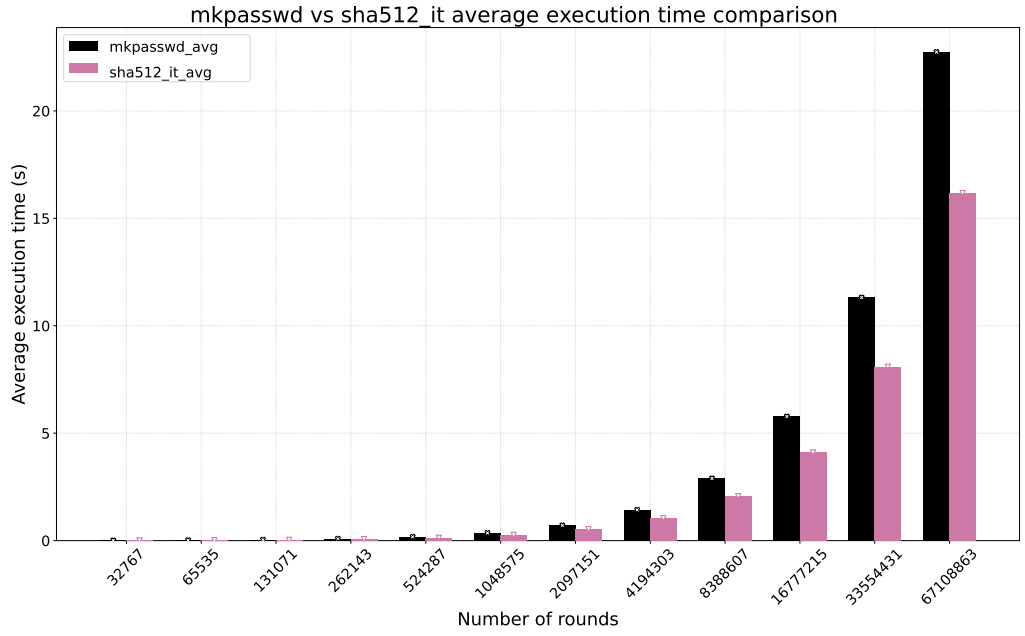


Figure 3.9: `mkpasswd` and `sha512_it` comparison.

Figure 3.9 compares the average execution time of `mkpasswd` and `sha512_it` as the number of rounds increases. Both algorithms' execution times increase approximately linearly with respect to the number of rounds. However, `mkpasswd`

appears consistently slower than `sha512_it` as it would be expected since, as described in Section 1.3.3, the `sha512crypt` function used by the `mkpasswd` utility performs several other operations in addition to the standard SHA-512 hash function.

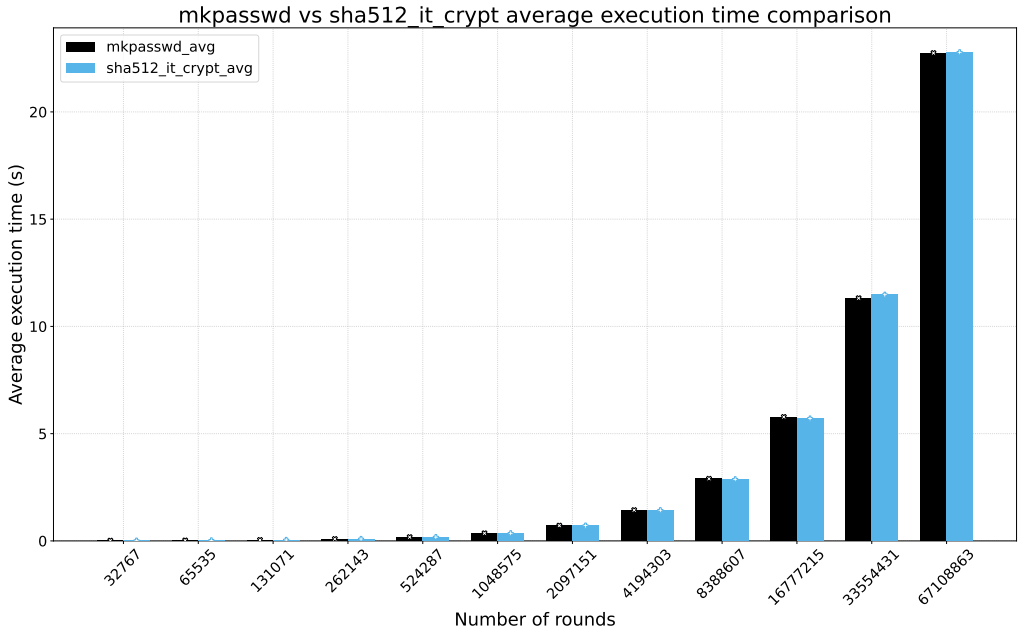


Figure 3.10: `mkpasswd` and `sha512_it_crypt` comparison.

Figure 3.10 compares `mkpasswd` and `sha512_it_crypt` as the number of rounds increases. In this case, the two implementations show very similar execution times across all the configurations. This is expected, since both rely on the `sha512crypt` hash function.

3.2.10 All algorithms comparison

This section presents a global comparison of all tested algorithms. The goal is to show a general overview of the average execution times of the algorithms in the same plot to address the actual advantages. Figure 3.11, Figure 3.12, and Figure 3.13 report the results for 2, 4, and 8 worker threads respectively.

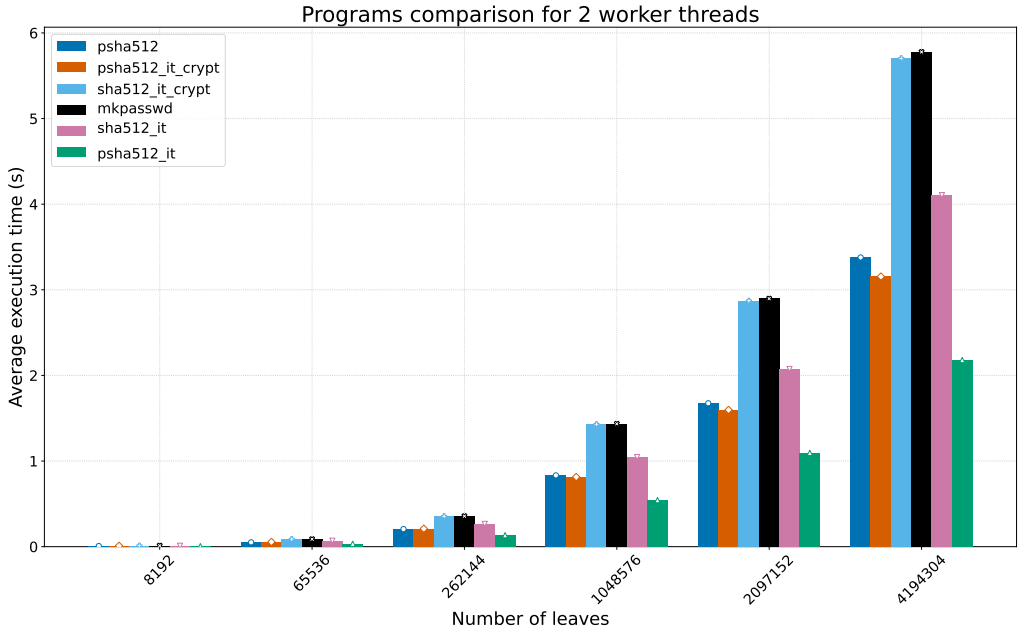


Figure 3.11: All algorithms comparison plot for 2 worker threads.

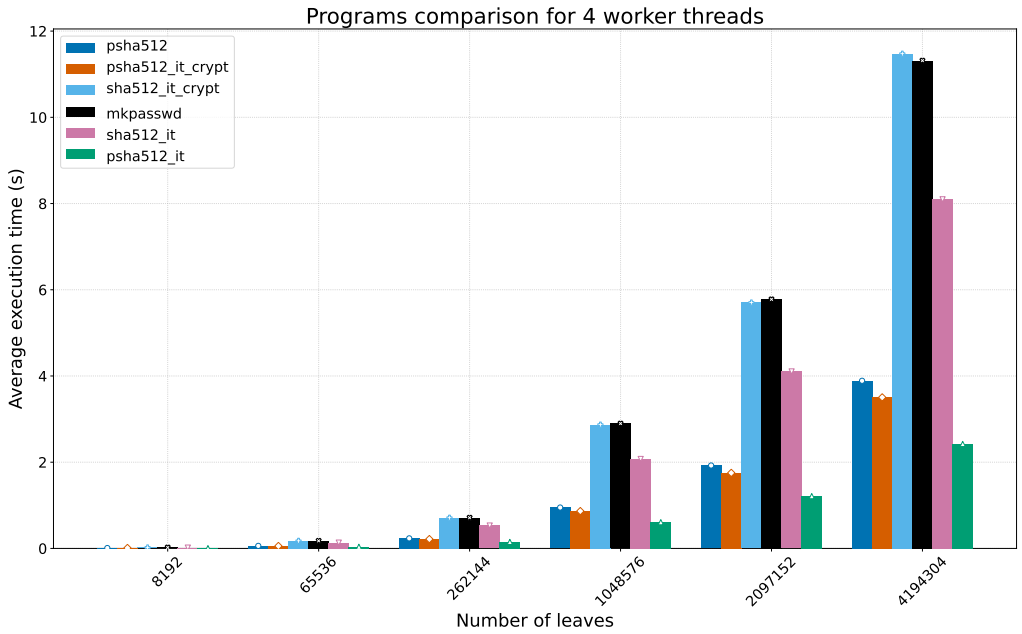


Figure 3.12: All algorithms comparison plot for 4 worker threads.

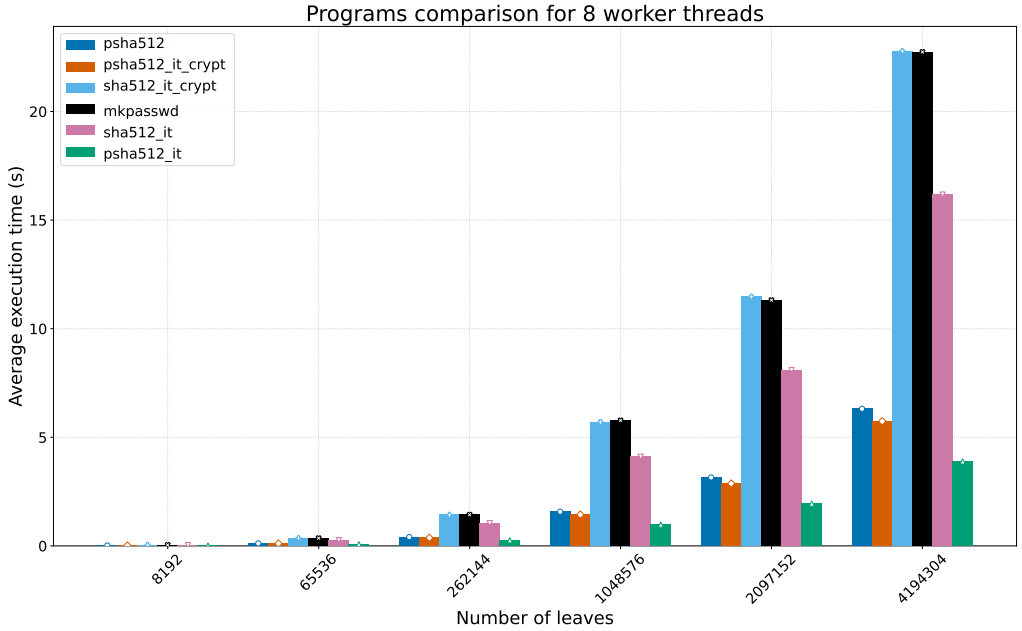


Figure 3.13: All algorithms comparison plot for 8 worker threads.

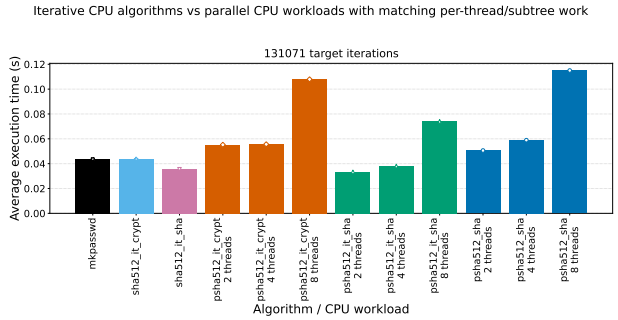
For large workloads, iterative algorithms show higher execution times across all configurations. Generally, parallel implementation are faster, with the `parallel_sha512_it` algorithm achieving the lowest execution times and consistently remaining the best-performing algorithm among the tested implementations.

3.2.11 CPU cost amplification under matched per-worker workloads

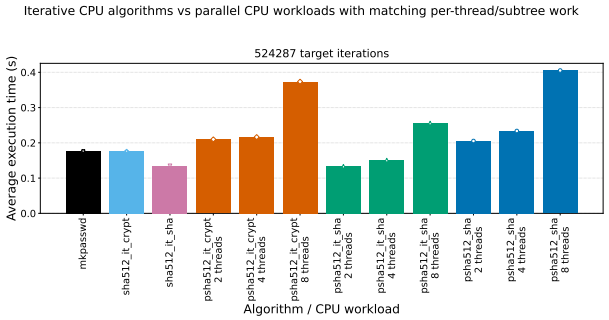
The central research question of this thesis addresses the problem of whether the parallel resources available to the defender can be used to increase the total amount of hashing work performed for each password candidate without proportionally increasing the corresponding authentication latency. To evaluate this property, the experiments reported in this section compare the iterative baselines with parallel CPU configurations having the same per-worker workload.

For each target workload N, the sequential algorithms perform approximately N iterative hash computations. In the corresponding parallel configurations, each worker thread or subtree performs approximately the same workload N.

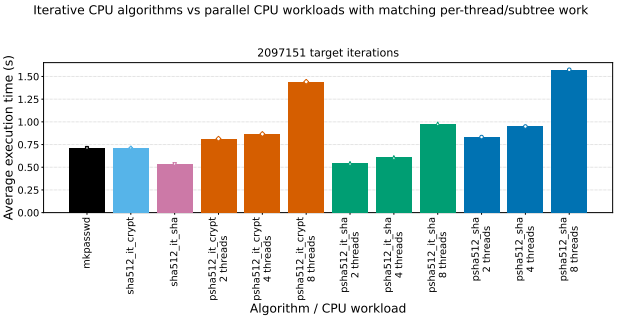
Therefore, a configuration using p worker threads performs approximately pN total computations, excluding the additional operations required for the final Merkle-tree reduction. Under ideal parallel execution, increasing the number of worker threads would increase the total work performed by a factor of p while keeping the execution time close to that of the sequential baseline. The results in Figure 3.14 show how closely the tested CPU implementations approach this behavior for workloads ranging from 131071 to 8388607 target iterations.



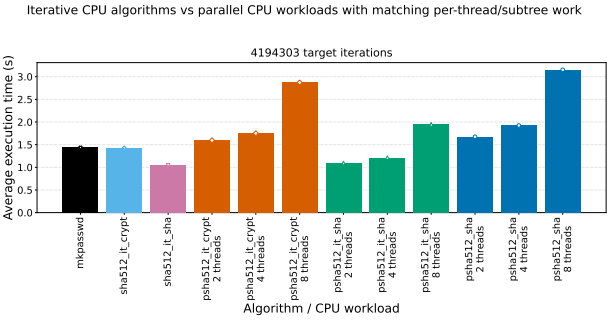
(a) Matched per-worker workload of 131071 hash computations.



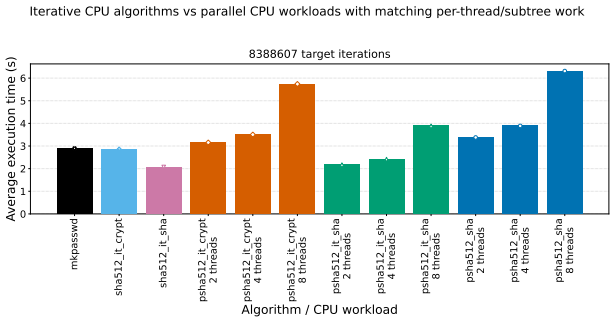
(b) Matched per-worker workload of 524287 hash computations.



(c) Matched per-worker workload of 2097151 hash computations.



(d) Matched per-worker workload of 4194303 hash computations.



(e) Matched per-worker workload of 8388607 hash computations.

Figure 3.14: Average CPU execution times of the iterative baselines and parallel algorithms under matched per-worker or per-subtree workloads.

Across all the workloads reported, the most relevant results are obtained for the `parallel_sha512_it` algorithm. Its two-threaded configuration consistently achieves execution times that are close to that of the sequential `sha512_it` baseline throughout the tested range, despite performing approximately twice the hashing work. For 131071 iterations, `parallel_sha512_it` is even faster than `sha512_it`. However, this result may be influenced by measurement variability and may be machine-dependent.

The four-thread configuration of `parallel_sha512_it` also introduces only a limited increase in execution time. For example, for 8388607 iterations, the sequential execution requires approximately 2.1 seconds, whereas the four-thread parallel configuration requires approximately 2.45 seconds, resulting in only about a 17% latency increase.

For the eight-thread configuration, its latency increases more substantially. For 8388607 iterations, it requires approx-

imately 3.9 seconds, compared with approximately 2.1 seconds for the sequential baseline. This loss of efficiency is likely related to the hardware configuration of the test machine, which provides only six physical CPU cores. Therefore, using eight worker threads introduces additional scheduling and contention overhead.

The `parallel_sha512_it_crypt` configurations show the same general trend, although their absolute execution times are higher because of the more complex internal structure of `sha512crypt`. The `parallel_sha512` configurations instead produce the largest execution times among the parallel algorithms for all workloads. This is caused by the construction's design, since it must generate independent leaves and perform the internal and final Merkle-tree reductions. These additional operations explain why its latency does not remain as close to that of the sequential baseline.

Overall, the results partially validate the parallel cost-amplification model. The strongest evidence is provided by `parallel_sha512_it`: two- and four-thread configurations increase the total iterative workload by approximately factors of two and four, respectively, while introducing only a limited increase in the execution time. However, the results also show that the benefit is bounded by the available physical resources. Once the number of worker threads exceeds the number of effectively available CPU cores, the authentication latency increases considerably.

3.3 GPU Execution Analysis

This section analyzes the average execution times of the GPU implementations of the proposed algorithms. The purpose of this analysis is to detemine whether moving the computation to the GPU provides a performance advantage over the CPU implementations. This analysis results particularly useful to understand how the algorithms can be exploited by common tools, such as hashcat and John the Ripper, in real-case password cracking scenarios.

3.3.1 `parallel_sha512` performance evaluation

This section evaluates the performance of the CPU and GPU implementations of `parallel_sha512` and compares their execution times for equivalent total workloads. The GPU implementation was tested up to 4096 OpenCL work-items, although the physical device only contains 1024 CUDA cores.

Figure 3.15 shows the average GPU execution time of the `parallel_sha512` algorithm as a function of the number of OpenCL work-items and the total number of leaves. For small workloads, the surface remains relatively flat, with values around 0.2 seconds. This suggests the presence of a fixed and unavoidable GPU overhead that dominates the execution times for small computations. As the number of leaves increases, the kernel computation becomes more significant and the execution times start to rise. Considering configurations with the same total workload shows that increasing the number of work-items does not always reduce the execution time. A large number of work-items decreases the number of leaves processed by each work-item, but it may also increase scheduling and management overhead. Consequently, the optimal configuration represents a trade-off between the amount of work assigned to each work-item and the overhead associated with launching and scheduling a larger global workload.

Figure 3.15: Complete average GPU execution time of `parallel_sha512` as a function of the number of OpenCL work-items and the total number of leaves.

Figure 3.16 reports only the OpenCL kernel execution times. In contrast to the results shown in Figure 3.15 the kernel time increases more directly with the workload size. For small workloads, the kernel execution time is close to zero, even though the corresponding total time is about 0.2 seconds. This confirms that the constant overhead is not caused by the hashing computation itself, but by GPU-related operations such as device initialization, data transfer, kernel launch and resource deallocation. The kernel time increases with the total number of leaves and reaches approximately 0.175 seconds for the largest configuration, while the corresponding total execution time is approximately 0.38 seconds. Therefore, the constant overhead remains relevant even for large workloads as indicated by the execution times recorded. The analysis of Figure 3.16 and Figure 3.15 demonstrate why kernel-only measurements should always be reported together with the total execution time of the GPU implementation.

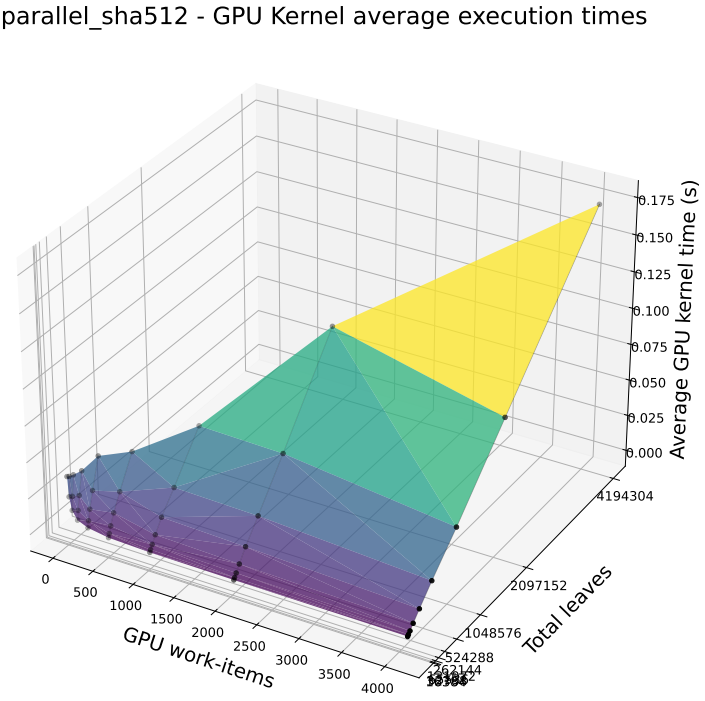
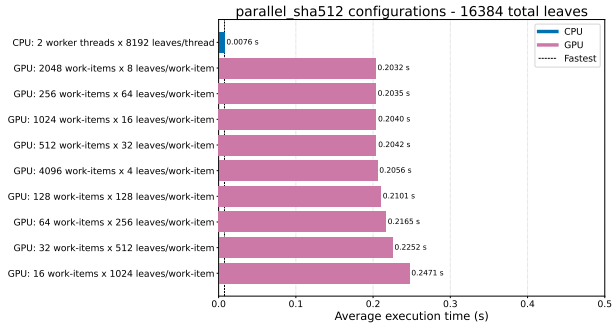
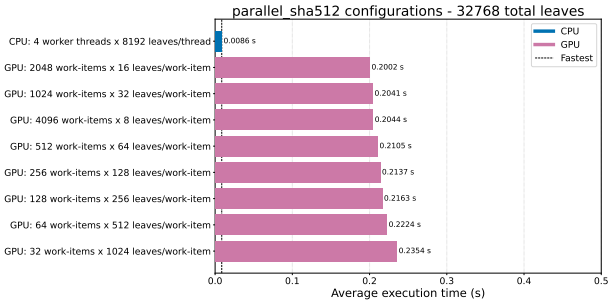


Figure 3.16: Average OpenCL kernel execution time of `parallel_sha512` for different numbers of work-items and total leaves.

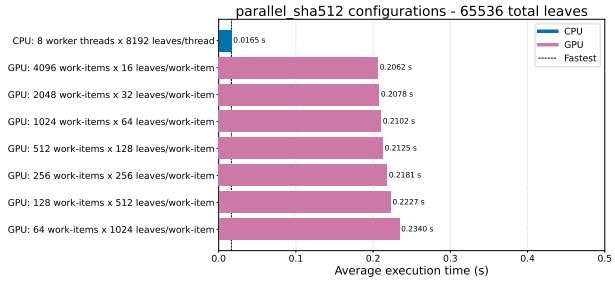
The subfigures in Figure 3.17 compare the different CPU and GPU configurations of number of worker-threads and leaves per worker-thread for an equal total number of leaves. Each plot corresponds to a different workload, ranging from 16384 to 4194304 leaves. For small workloads, the CPU implementation is considerably faster than every GPU configuration. For instance, with 16384 leaves the CPU computation requires 0.0076 seconds, while the fastest GPU configuration requires approximately 0.2032 seconds. A similar behavior is also observed for 32768 and 65536 leaves. As the workload increases, the CPU execution time grows more rapidly, while the GPU time increases more gradually and remains close to its initial value for the smaller workloads. This trend reflects also the results shown in Figure 3.16 and Figure 3.15. The performance of the two architectures start to be comparable for 1048576 leaves, where the fastest CPU configuration requires 0.2336 seconds and the fastest GPU configuration requires 0.2491 seconds. For large workloads, the GPU becomes the fastest architecture. For 2097152 leaves, the best GPU configuration, composed of 4096 work-items processing 512 leaves each, completes the computation in 0.2942 seconds, whereas the best CPU configuration requires 0.4052 seconds. As the number of leaves grows, the GPU advantage becomes more and more pronounced. Analyzing execution times of the different GPU configurations increases, the trend shows that generally distributing the leaves among a larger number of work-items improves the performance. This is more evident for smaller workloads where configurations with few work-items and lots of leaves achieve worse execution times than configurations with a large number of work-items and fewer leaves per work-item. Overall, the figure identifies a clear transition between the two architectures. On the tested hardware, the crossover occurs between 1048576 and 2097152 leaves. Beyond this region, the greater parallel processing capability of the GPU compensates for the fixed overhead and provides a significant performance advantage.



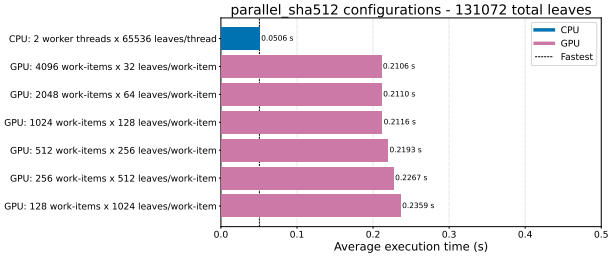
(a) 16384 total leaves.



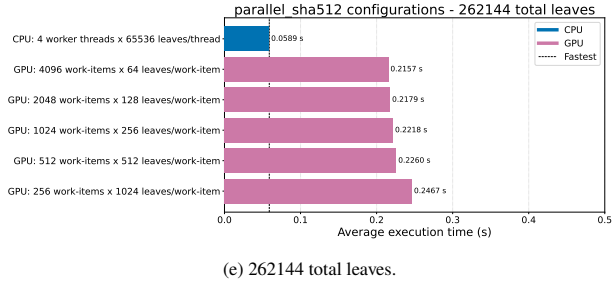
(b) 32768 total leaves.



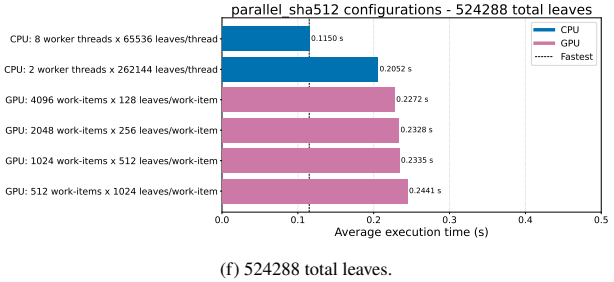
(c) 65536 total leaves.



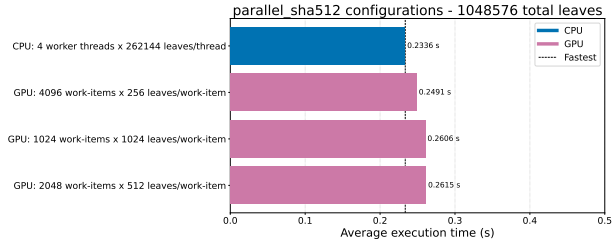
(d) 131072 total leaves.



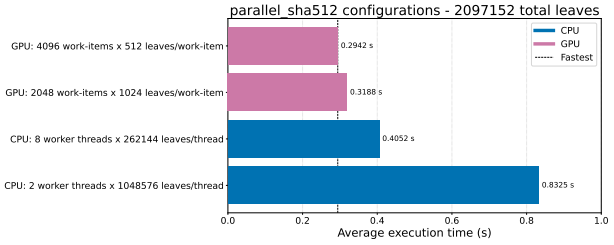
(e) 262144 total leaves.



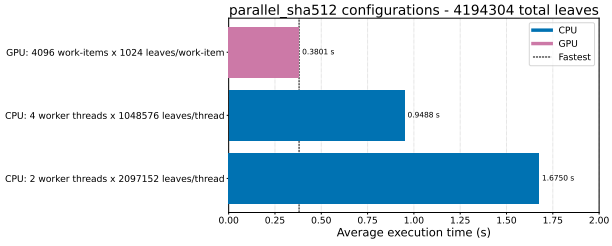
(f) 524288 total leaves.



(g) 1048576 total leaves.



(h) 2097152 total leaves.



(i) 4194304 total leaves.

Figure 3.17: Comparison of CPU and GPU configurations of `parallel_sha512` for different total numbers of leaves.

Figure 3.18 compares the fastest CPU and GPU configurations for the same total number of leaves. The plot shows that the CPU implementation is significantly faster for small and medium workloads, from 16384 to 524288 leaves. In this case, the CPU execution time remains below approximately 0.12 seconds, whereas the GPU time is significantly higher and also appears to be constant, with values around 0.2 seconds. This behavior suggests that the fixed GPU execution overhead, associated with host-to-device and device-to-host transfers, device initialization, kernel launch and resource deallocation, dominates the kernel computation. As the workload increases, the difference between the two

implementations progressively decreases. The execution times become comparable at about 1048576 leaves. For large workloads, e.g. for 4194304 leaves, the GPU implementation outperforms the CPU version: the GPU execution requires approximately 0.38 seconds, compared with approximately 0.95 seconds for the CPU. In the region between 1048576 and 2097152 leaves, the greater parallelism available on the GPU compensates the fixed execution overhead which caused the slowdown for small workloads.

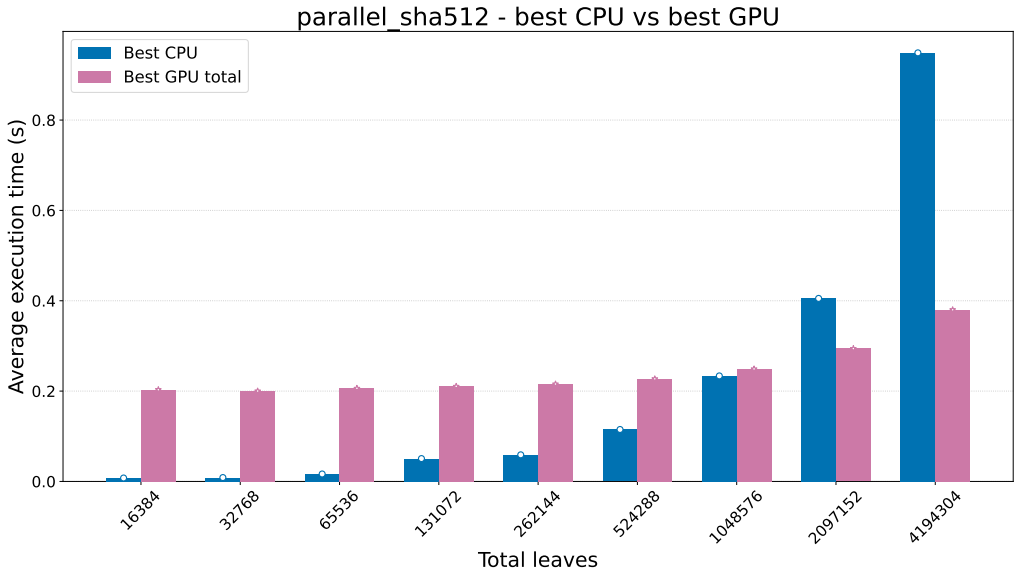


Figure 3.18: Average execution times of the fastest CPU and GPU `parallel_sha512` configurations for equal total workloads.

Figure 3.19 reports the speed-up of the best GPU configuration over the best CPU configuration, computed as the ratio between CPU and GPU execution times. Values lower than one indicate that the CPU is faster, whereas values above one indicate that the GPU is faster. As already shown in Figure 3.18, for small workloads the ratio clearly favors the CPU implementation. The constant GPU overhead has a significant impact on the performance and makes GPU execution considerably slower for small workloads. The speed-up increases with the total number of leaves and approaches one at 1048576 leaves. The crossover occurs in the 1048576 and 2097152 number of leaves region. Beyond this point, the GPU is consistently faster that the CPU implementation, achieving speed-ups of 1.4× for 2097152 leaves and 2.5× for 4194304 leaves. This trend shows that the GPU implementation benefits more from larger workloads allowing the independent computation to amortize the fixed execution overhead observed.

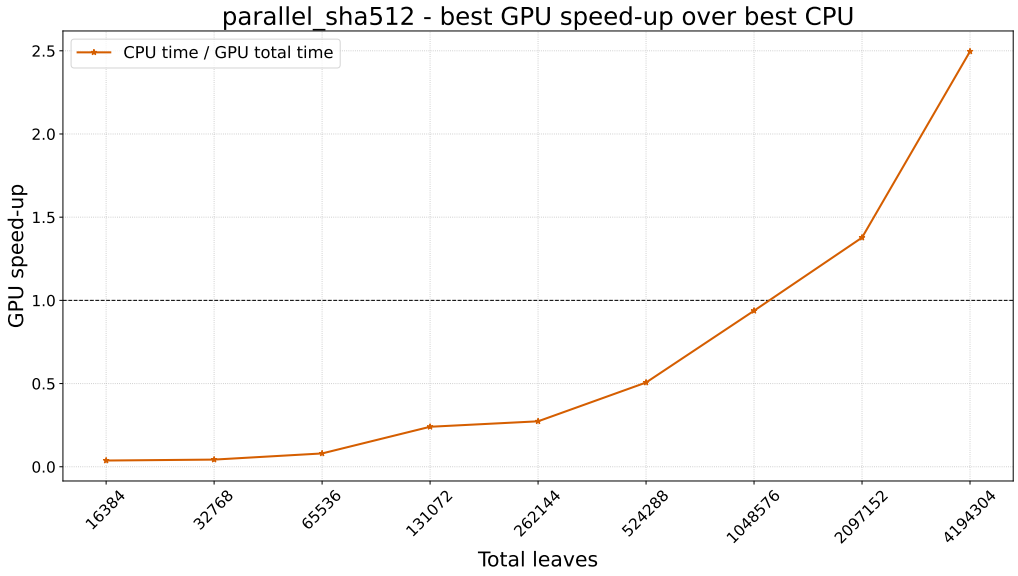
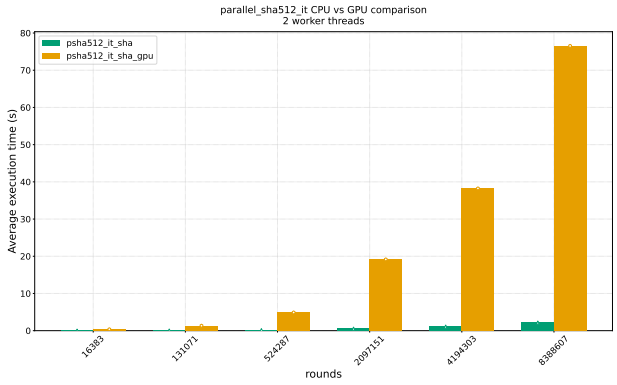


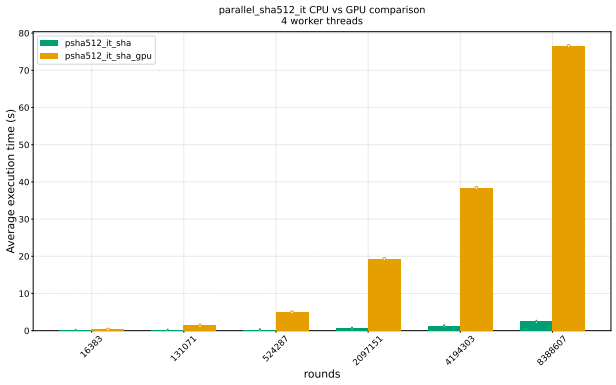
Figure 3.19: Speed-up of the fastest GPU configuration over the fastest CPU configuration of `parallel_sha512`.

3.3.2 CPU and GPU performance comparison of `parallel_sha512_it`

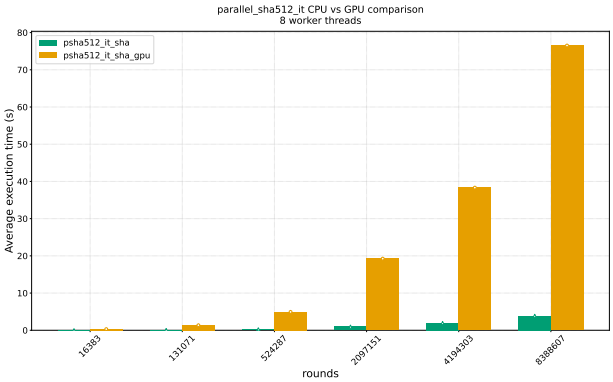
This section compares the CPU and GPU execution times of `parallel_sha512_it` for different round values and different number of worker threads. For all configurations, the GPU implementation is significantly slower than the CPU implementation. The difference becomes particularly evident when the number of rounds increases. These results suggest that the current GPU implementation does not exploit the GPU architecture efficiently for this algorithm. This result is particularly relevant from a practical security perspective. It suggests that, in a real password-cracking scenario, tools like hashcat or John the Ripper, designed to exploit GPU parallelism, would not be able to compute a large number of password candidates efficiently for this algorithm. Therefore, the limited effectiveness of the GPU implementation represents an advantage of the proposed approach.



(a) `parallel_sha512_it` algorithm execution time comparison CPU and GPU with 2 worker threads.



(b) `parallel_sha512_it` algorithm execution time comparison CPU and GPU with 4 worker threads.



(c) `parallel_sha512_it` algorithm execution time comparison CPU and GPU with 8 worker threads.

Figure 3.21 presents the ratio between GPU and CPU execution time for `parallel_sha512_it`. In every tested configuration, all values are greater than 1, meaning that the GPU implementation is consistently slower than the CPU implementation. The ratio is particularly high for small workloads, whereas it decreases and becomes more stable as the number of rounds increases.

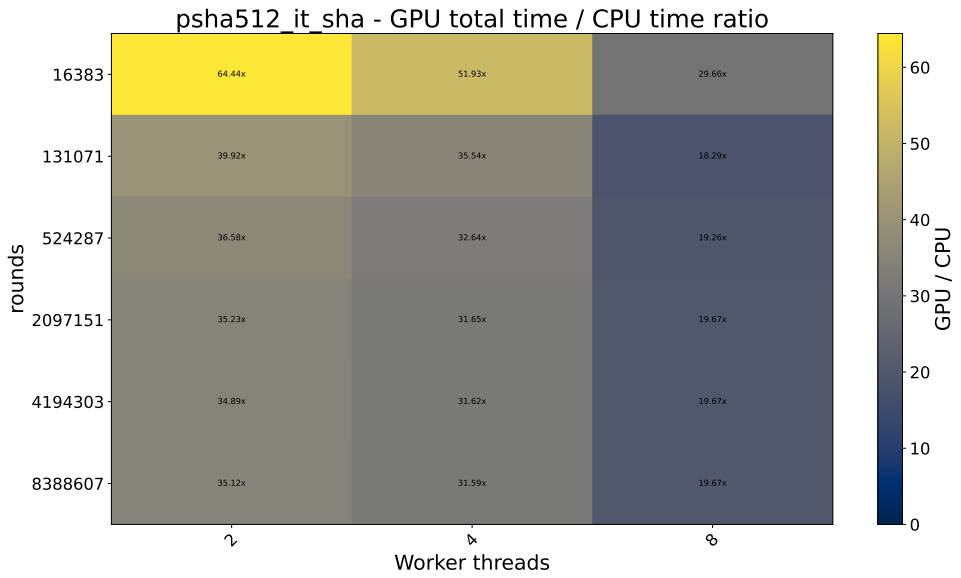


Figure 3.21: `parallel_sha512_it` GPU/CPU ratio heatmap.

3.3.3 CPU and GPU performance comparison of `sha512_it`

Figure 3.22 compares the CPU and GPU implementations of `sha512_it`. For all the numbers of rounds tested, the GPU implementation is consistently slower than the CPU version. As the number of rounds increases, the difference is more and more noticeable. The `sha512_it` algorithm does not benefit from the GPU architecture. Since each iteration depends from the previous one, the algorithm does not exploit parallelism in any way.

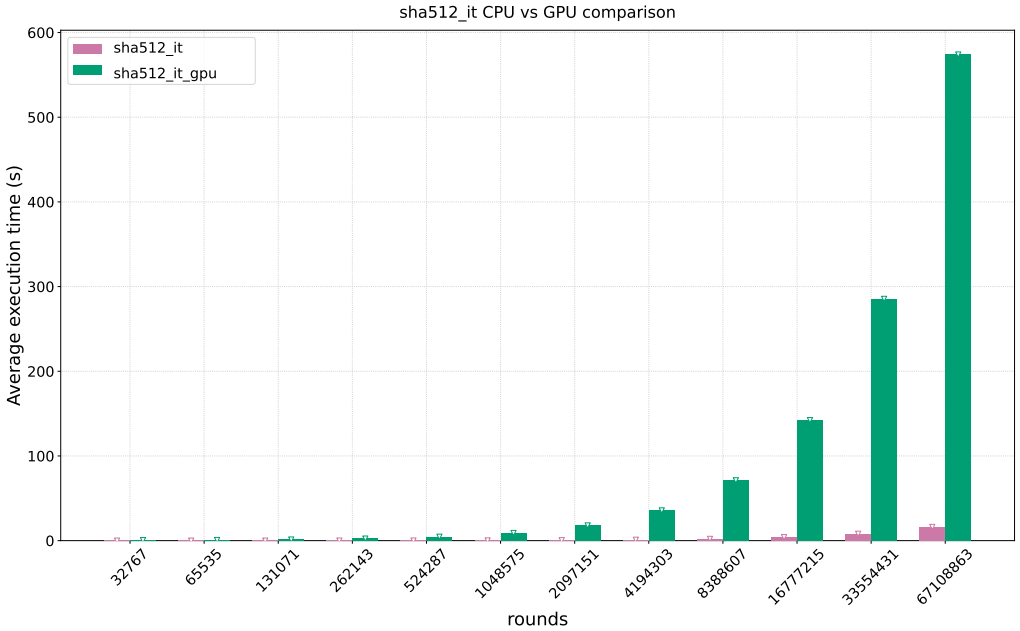


Figure 3.22: `sha512_it` CPU vs GPU comparison.

Figure 3.23 shows the GPU/CPU execution time ratio for `sha512_it`. The GPU implementation is more than 30× slower than the CPU version for all input configurations. As the number of rounds increases, the ratio becomes relatively stable indicating that as the workload raises, the impact of GPU overheads is reduced.

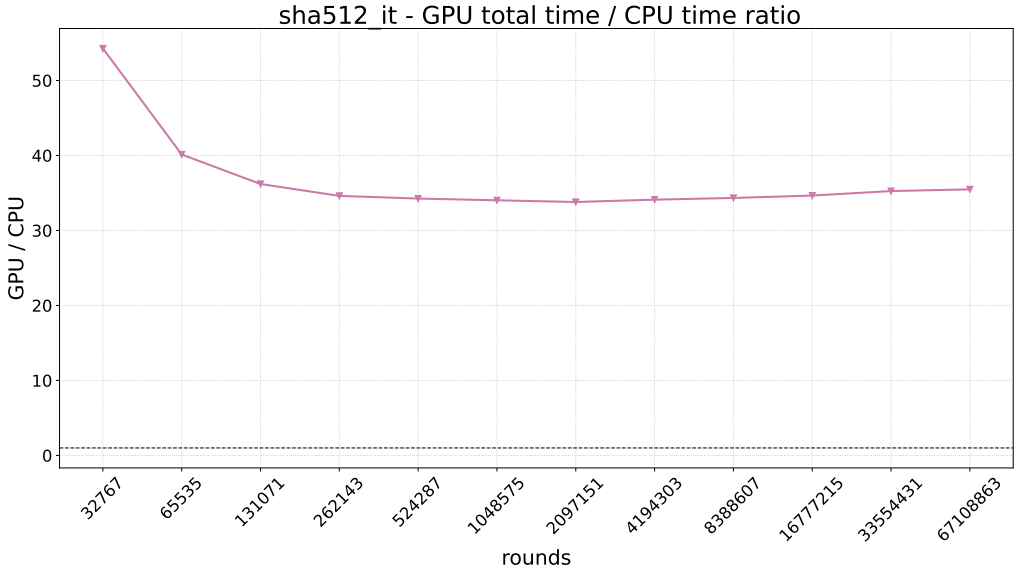
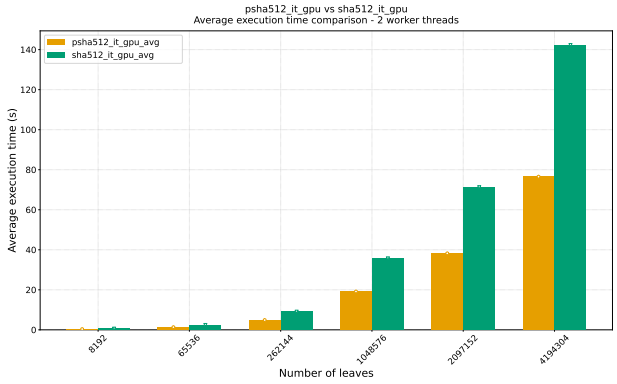


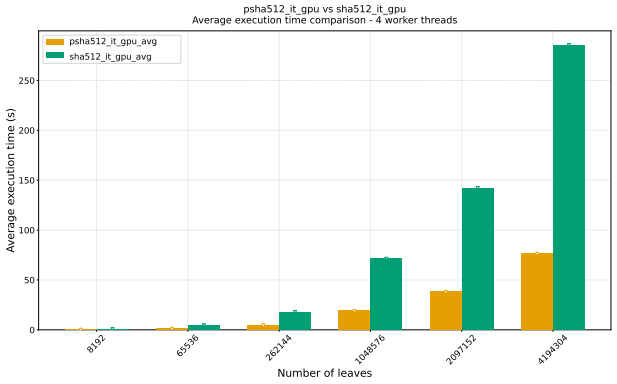
Figure 3.23: `sha512_it` GPU/CPU ratio.

3.3.4 `parallel_sha512_it` and `sha512_it` comparison

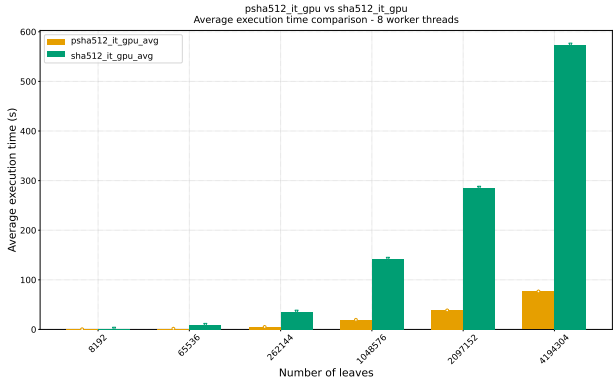
This section compares the GPU implementations of `parallel_sha512_it` and `sha512_it`. This comparison is useful to evaluate whether the parallel structure provides an advantage in the GPU environment itself.



(a) `parallel_sha512_it` and `sha512_it` execution time comparison with 2 worker threads.



(b) `parallel_sha512_it` and `sha512_it` execution time comparison with 4 worker threads.



(c) `parallel_sha512_it` and `sha512_it` execution time comparison with 8 worker threads.

Number of leaves	`parallel_sha512_it` GPU average execution time (s)	`sha512_it` GPU average execution time (s)	`parallel_sha512_it` / `sha512_it` GPU ratio
8192	0.3673	0.5644	0.6508
65536	1.3334	2.4020	0.5551
262144	4.8946	9.0466	0.5410
1048576	19.1502	35.6808	0.5367
2097152	38.2256	71.2373	0.5366
4194304	76.5294	142.2649	0.5379

Table 3.14: `parallel_sha512_it` and `sha512_it` GPU execution time comparison with 2 worker threads.

Number of leaves	`parallel_sha512_it` GPU average execution time (s)	`sha512_it` GPU average execution time (s)	`parallel_sha512_it` / `sha512_it` GPU ratio
8192	0.3116	0.7505	0.4152
65536	1.3542	4.6175	0.2933
262144	4.9350	17.9389	0.2751
1048576	19.2513	71.2373	0.2702
2097152	38.3561	142.2649	0.2696
4194304	76.5523	285.4798	0.2682

Table 3.15: `parallel_sha512_it` and `sha512_it` GPU execution time comparison with 4 worker threads.

Number of leaves	`parallel_sha512_it` GPU average execution time (s)	`sha512_it` GPU average execution time (s)	`parallel_sha512_it` / `sha512_it` GPU ratio
8192	0.3114	1.2849	0.2424
65536	1.3550	9.0466	0.1498
262144	4.9364	35.6808	0.1383
1048576	19.2480	142.2649	0.1353
2097152	38.3335	285.4798	0.1343
4194304	76.5200	573.9961	0.1333

Table 3.16: `parallel_sha512_it` and `sha512_it` GPU execution time comparison with 8 worker threads.

As reported numerically in Table 3.14, Table 3.15, and Table 3.16, it is noticeable that, as the number of worker threads increases, also the speed-up increases correspondingly. With 2, 4, and 8 worker threads, the achieved speed-up is approximately of 1.86×, 3.7×, and 7.5× respectively. This shows that the parallel algorithmic design of `parallel_sha512_it` is particularly beneficial on the GPU. The GPU speed-ups are much closer to the expected theoretical performance gain than those observed in the CPU implementation.

3.4 Discussion of the CPU and GPU experimental results

The experimental analysis confirms that the proposed parallel algorithms provide a substantial performance advantage with respect to the corresponding iterative baselines. The theoretical analysis suggests that the expected speed-up is proportional to the number of independent subtrees, assuming ideal parallel execution and ignoring memory-management, thread scheduling and synchronization overheads. The experimental results show that this theoretical behavior is only

partially reflected in practice when the algorithms are executed on the CPU. The parallel implementations are consistently faster than the iterative ones, however, the measured speed-up does not reach the expected theoretical values.

When considering the CPU implementations, `parallel_sha512_it` achieves the best overall performance under all input configurations. Its execution time is consistently lower than the other tested algorithm variants, especially as the workload increases the difference becomes more and more noticeable. This result confirms that distributing the iterative hashing computation among independent subtrees is a very effective technique to reduce the total execution time. Conversely, increasing the number of threads was observed to raise the thread synchronization and management overhead and the final Merkle tree reduction impact. This is particularly evident analysing the execution times when executing the parallel algorithms on 2 and 4 worker threads. Moreover, since the machine used to run the tests has only 6 physical CPU cores, input configurations using 8 worker threads did not benefit from the same efficiency as configurations that remain within the number of physical cores.

The matched per-worker experiments provide more direct evidence about the central research question. In particular, the four-thread configuration of `parallel_sha512_it` performed approximately four times the hashing work of the corresponding sequential baseline while increasing the execution time by only about 17%, for the largest tested workload. This result shows that, when the number of worker threads remains within the number of CPU cores, server-side parallelism can be converted into a substantial increase in per-candidate computational work without having a proportional increase in verification latency.

For the `parallel_sha512` algorithm, the CPU implementation achieves lower execution times with respect to the GPU implementation for small and medium workloads. This is caused by the fixed overhead associated with GPU operations such as device initialization, memory transfers, kernel launch, resource deallocation and host-side processing. As the number of leaves increases, however, the GPU execution time grows more gradually than the CPU execution time. The two implementations show comparable performance at approximately 1048576 leaves, while the GPU becomes faster for larger workloads. These results show that `parallel_sha512` can effectively exploit the greater parallelism offered by the GPU, provided that the workload is sufficiently large to compensate for the additional execution overhead.

The GPU experiments provide a different perspective. GPU implementations of the `parallel_sha512_it` and of the `sha512_it` algorithms are slower than the corresponding CPU implementations in the tested configurations, suggesting that these algorithms do not exploit the GPU architecture effectively. This result is expected especially for `sha512_it` due to its inherent sequential design. However, when comparing the GPU execution times of these two algorithms, the observed speed-ups are much closer to the expected theoretical performance gain than those obtained on the CPU. Although in absolute terms the GPU implementation is slower, the parallel structure of the algorithms is better reflected on the GPU side.

This result is also particularly relevant from a practical security perspective. Password cracking tools such as hashcat [6, 7] and John the Ripper [16] typically rely on GPUs to evaluate a very large number of password candidates in parallel. Since the `parallel_sha512_it` does not map efficiently on the GPU architecture as suggested by the experimental results, attackers may not be able to effectively exploit the massive parallelism of modern GPUs when dealing with the proposed algorithms.

Overall, `parallel_sha512_it` represents the most promising variant for CPU execution consistently achieving the lowest execution times, whereas `parallel_sha512` benefits more clearly from GPU execution when the workload is sufficiently large. The matched per-worker results provide direct support for the central hypothesis of the thesis: available CPU parallelism can be used to increase the amount of hashing work performed for each password candidate while limiting the corresponding increase in verification latency.

Chapter 4. Integration of the algorithms into hashcat

After describing the functioning and evaluating the execution times of the proposed algorithms, this chapter focuses on their integration into hashcat. The goal is to implement new hash modes to compute and verify the hashes produced by the proposed algorithms. Hashcat provides a modular architecture that allows new hash modes to be added implementing a CPU-side module and a GPU-side kernel. This chapter describes how the `parallel_sha512` and `parallel_sha512_it` algorithms were integrated into this framework in order to evaluate their behavior in a real-case password cracking scenario.

4.1 SHA-512 cracking example

In Section 1.7 we discussed the implementation of the SHA-512 cryptographic hash function in hashcat using the OpenCL framework. That implementation allows hashcat to compute SHA-512 digests on the GPU, however it does not directly corresponds to the structure used by hashcat during the cracking process. In hashcat, the implementation of a cryptographic hash function is divided into two components: a CPU-side module and a GPU-side kernel. The CPU-side module is responsible for configuring the hash mode, parsing the input hash, extracting parameters, defining the digest comparison, and providing self-test information. The GPU-side OpenCL kernel, instead, contains the implementation of the algorithm which is then executed on the selected compute device and represents the computationally intensive part of the cracking process. Furthermore, each encryption algorithm or hash function is identified by a unique 5-digit number with leading zeros that is used in the filename of each file related to that specific algorithm or function.

For raw SHA-512, its numeric identifier is 1700, which is also called hash mode. Its implementation files are `module_01700.c`, located in the `src/modules/` folder, and `m01700_a0-pure.cl`, located in the `OpenCL/` folder.

4.1.1 `module_01700.c` module description

`module_01700.c` represents the CPU-side module associated with hashcat mode 1700. Its purpose is to describe how this hash type must be handled by the hashcat framework. The following paragraphs provide an overview of the most relevant parts of the module.

The module defines several static configuration values, including: hash name, kernel type, digest size and comparison positions, salt type, optimization flags. It also provides a test password and the corresponding hash, which is used to verify that the implementation works correctly before starting the cracking process. Listing 4.20 reports a simplified excerpt of the static configuration values defined by the module.

```
1 static const u32   ATTACK_EXEC    = ATTACK_EXEC_INSIDE_KERNEL;
2 static const u32   DGST_POS0      = 14;
3 static const u32   DGST_POS1      = 15;
4 static const u32   DGST_POS2      = 6;
5 static const u32   DGST_POS3      = 7;
6 static const u32   DGST_SIZE      = DGST_SIZE_8_8;
7 static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH;
```

```
static const char *HASH_NAME      = "SHA2-512";
static const u64   KERN_TYPE      = 1700;
static const u32   OPTI_TYPE      = OPTI_TYPE_ZERO_BYTE | ... | OPTI_TYPE_RAW_HASH;
static const u64   OPTS_TYPE      = OPTS_TYPE_STOCK_MODULE | ... |
 ↪ OPTS_TYPE_PT_ADDBITS15;
static const u32   SALT_TYPE      = SALT_TYPE_NONE;
static const char *ST_PASS        = "hashcat";
static const char *ST_HASH        =
 ↪ "82a9dda829eb7f8ffe9fbe49e45d47d2dad9664fbb7adf72492e3c81ebd3
e29134d9bc12212bf83c6840f10e8246b9db54a4859b7ccd0123d86e5872c1e5082f";
```

Listing 4.20: `module_01700.c` static configuration values definition.

The module contains three main functions:

- `module_hash_decode`: this function receives the textual hash representation, checks its validity, converts it into eight 64-bit words, and applies the byte-order conversion that is required.
- `module_hash_encode`: this function implements the inverse operation with respect to the previous function, meaning that, given the internal digest representation, it converts it back into the standard hexadecimal format.
- `module_init`: this function connects all module-specific functions to the general hashcat module context so that hashcat knows which functions to call to obtain all the information about this specific hash function.

```
void module_init(module_ctx_t *module_ctx) {
    module_ctx->module_context_size             = MODULE_CONTEXT_SIZE_CURRENT;
    module_ctx->module_interface_version        = MODULE_INTERFACE_VERSION_CURRENT;
    module_ctx->module_attack_exec              = module_attack_exec;
    ...
    module_ctx->module_dgst_pos0                = module_dgst_pos0;
    module_ctx->module_dgst_pos1                = module_dgst_pos1;
    module_ctx->module_dgst_pos2                = module_dgst_pos2;
    module_ctx->module_dgst_pos3                = module_dgst_pos3;
    module_ctx->module_dgst_size                = module_dgst_size;
    ...
    module_ctx->module_hash_decode              = module_hash_decode;
    module_ctx->module_hash_encode              = module_hash_encode;
    ...
}
```

Listing 4.21: `module_01700.c` module_init() function excerpt.

Therefore, the `module_01700.c` can be seen as the interface layer between the generic hashcat framework and the specific SHA-512 implementation.

4.1.2 `m01700_a0-pure.cl` kernel description

`m01700_a0-pure.cl` is the pure OpenCL kernel for hashcat mode 1700, which corresponds to raw SHA-512. It is specific for attack mode 0 which means straight/dictionary attack, possibly with rule-based candidate transformations. Inside this kernel, two main functions are defined:

- `m01700_mxx()`: used when hashcat is cracking multiple target hashes;
- `m01700_sxx()`: used when hashcat is cracking only one target hash.

The two functions share the same hashing logic, the main difference lies in how the hash comparison is handled. Each function computes one hash at a time, but when dealing with multiple target hashes the computed hash has to be compared with multiple values. When dealing with a single target hash, the compared value is loaded directly inside the function. Here is an overview of how the cracking process for SHA-512 works.

- Each OpenCL work-item processes one base password candidate which is retrieved using the work-item's global identifier.
- The candidate password is loaded from the `pws` buffer through the `COPY_PW` macro.
- Then, using a for loop, the original password is restored at each iteration using the `PASTE_PW` macro, modified through the `apply_rules` function, hashed using OpenCL SHA-512 primitives described in Section 1.7.
- Finally, the generated password digest is compared with the target digest.

This design has been adopted to allow hashcat to generate password candidate variations directly inside the GPU, reducing the data transferred between different devices.

4.2 Integration of the `parallel_sha512` algorithms into hashcat

This section focuses on how the `parallel_sha512` and the `parallel_sha512_it` algorithms were integrated inside the hashcat cracking process. Two new modules and two corresponding OpenCL kernels were implemented. It discusses the fundamental implementation without providing a line by line code description. This section describes the structure of the hashcat integration and on the main computational steps performed by the OpenCL kernels. To implement the hashcat integration of both algorithms and to make the parameters required by each algorithm available during the cracking process, their output string has been redefined to embed the algorithm's parameters as follows:

- for the `parallel_sha512` algorithm, the `n_subtrees`, the `n_leaves`, and the `salt` are embedded using the following output format:

  `$psha512$n_subtrees=<N_SUBTREES>$n_leaves=<N_LEAVES>$<SALT>$<DIGEST>`.

- for the `parallel_sha512_it` algorithm, the `n_subtrees`, the `rounds`, and the `salt` are embedded using the following output format:

  `$psha512_it$n_subtrees=<N_SUBTREES>$rounds=<ROUNDS>$<SALT>$<DIGEST>`.

In these formats, `<N_SUBTREES>`, `<N_LEAVES>`, `<ROUNDS>`, `<SALT>`, `<DIGEST>`, are placeholders that are replaced by the corresponding values. A relevant difference between the standalone OpenCL implementation described in Section 2.4 and the hashcat kernel is how the parallelism is exploited. In the standalone implementation, the computation of the leaves and subtree roots is distributed among several work-items. In the hashcat integration, instead, each work-item is associated with a different password candidate. As a consequence, the computation of the Merkle tree for a single candidate is performed sequentially inside one work-item, while the GPU parallelism is exploited by evaluating different password candidates at the same time. This will result in a inevitable hash computation slow down.

4.2.1 Hashcat mode 92000 for the `parallel_sha512` algorithm

Hashcat mode 92000 was defined for the `parallel_sha512` algorithm. In particular, the files `module_92000.c` and `m92000_a0-pure.cl` provide the C module and OpenCL kernel implementation respectively. Inside the module, a custom kernel type is assigned, the hash name is defined, the digest size and comparison positions are specified. Moreover, the accepted textual hash format is defined as explained at the beginning of this section. The decoding and encoding functions are defined to parse the algorithm's output string. In the decoding function, the module parses the `n_subtrees`, the `n_leaves`, and the `salt` fields, validates their values, stores them, decodes the salt and converts the hexadecimal digest

into the internal representation used by hashcat. The encoding function performs the inverse operation, reconstructing the correct output string from the internal representation. The `m92000_a0-pure.cl` OpenCL kernel implements the computation performed for each password candidate. Each GPU work-item is associated with one candidate password generated by Hashcat, to which also the selected mutation rules are applied, if any. The digest is computed exactly as described in Algorithm 2 and in Algorithm 1. However, within the hashcat kernel, the Merkle-tree computation of a single password candidate is performed sequentially by a single work-item. The implementation still preserves inter-candidate parallelism, since many password candidates are processed in parallel by the GPU, but it does not exploit intra-candidate parallelism for the computation of the leaves and the subtree roots. The individual subtree roots are now produced through a for loop that computes them one at a time and saves the partial results in a predefined static array. Since the partial subtree roots are stored in a statically allocated array, the implementation supports at most 128 subtrees. Consequently, hash strings specifying a larger value of number of subtrees are rejected during the decoding and validation phase. Once all the subtree roots are computed, they are reduced using the usual Merkle reduction procedure, producing the final root of the whole tree. Finally, the final digest is compared against the target digest using hashcat's comparison macros. This represents an important limitation with respect to the standalone OpenCL implementation presented previously.
The full source code for the hash mode 92000 is available at Appendices A.9 and A.11.

4.2.2 Hashcat mode 92001 for the `parallel_sha512_it` algorithm

The structure of the `module_92001.c` module is very similar to that of `module_92000.c` implementation, but it adapts the parsing and validation logic to the parameters of the `parallel_sha512_it` algorithm. The module now parses and validates the `rounds` parameter instead of the number of leaves, but the core structure remains unchanged. The `m92001_a0-pure.cl` OpenCL kernel also presents very few differences from the `m92000_a0-pure.cl` implementation. The key difference lies in how each subtree computes its subtree root digest, following the specification of the algorithm as presented in Algorithm 4 and in Algorithm 3. In this case, each subtree root is obtained by repeatedly applying SHA-512 for the specified number of rounds, instead of through a Merkle tree reduction. As with the previous algorithm, the subtree root computation is performed sequentially one at a time, and the number of subtrees limited to 32. The Merkle root computation and comparison remain unchanged with respect to the `parallel_sha512` hashcat kernel.
The full source code for the hash mode 92001 is available at Appendices A.10 and A.12.

4.3 Hashcat tests and conclusions

This section evaluates the performance of the `parallel_sha512` and `parallel_sha512_it` algorithms after their integration into hashcat. The objective of this evaluation is to determine how the input parameters affect the number of password candidates that can be processed per second in a typical hashcat password-cracking scenario.

This throughput-based evaluation reflects the attack model introduced in Chapter 2. Password-cracking tools already exploit the available GPU parallelism by evaluating many password candidates concurrently. Therefore, the proposed algorithms are not intended to prevent an attacker from using parallel hardware. Instead, they aim to increase the computational work required for each candidate, thereby reducing the number of candidates that can be evaluated per second.

The experiments were performed using an automated testing script that, for each parameter configuration, generated valid password hashes and executed the newly introduced custom hashcat modes using the `-speed-only` option, which allows the hashing throughput to be measured without performing a complete cracking session. Each configuration was executed and measured 50 times in order to reduce the influence of measurement variability. The resulting speeds were all converted into hashes per second, filtered to discard failed or incomplete measurements and aggregated by algorithm and parameter configuration.

For `parallel_sha512`, the analysis considers the number of subtrees and the number of leaves assigned to each subtree. For `parallel_sha512_it`, it considers the number of subtrees and the number of iterative rounds. The following plots show the results obtained, focusing in particular on how the hashing speed changes as these parameters vary.

Figure 4.1 and Figure 4.2 report the hashcat performance of the `parallel_sha512` as a function of the number of subtrees and the number of leaves assigned to each subtree. The heatmap represents the decimal logarithm of the hashing speeds, since the recorded speeds span several orders of magnitude. The line plot also uses logarithmic axes. In both figures it is noticeable that the hashing rate decreases as either the number of worker-threads or of leaves increases. The most significant results are the most computationally expensive configurations, since they produce the lowest cracking throughput. The configuration with 32 worker-threads and 1024 leaves per worker-thread is the slowest one, which also uses the largest tested values of both parameters, and reaches only approximately 14 hashes per second. When fixing the number of leaves, configurations with fewer worker-threads are consistently faster. Similarly, when fixing the number of worker-threads, increasing the number of leaves causes a progressive decrease in throughput. For the largest configurations, the hashing speed is reduced to only a few tens or hundreds of hashes per second.

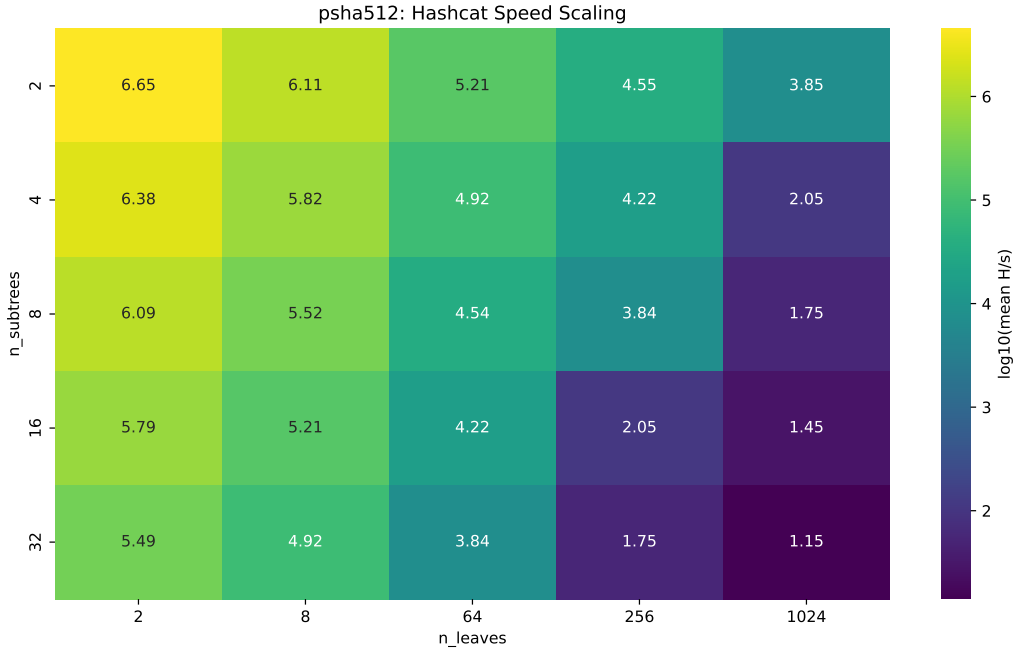


Figure 4.1: Logarithm of the mean Hashcat speed of `parallel_sha512` for different numbers of subtrees and leaves per subtree.

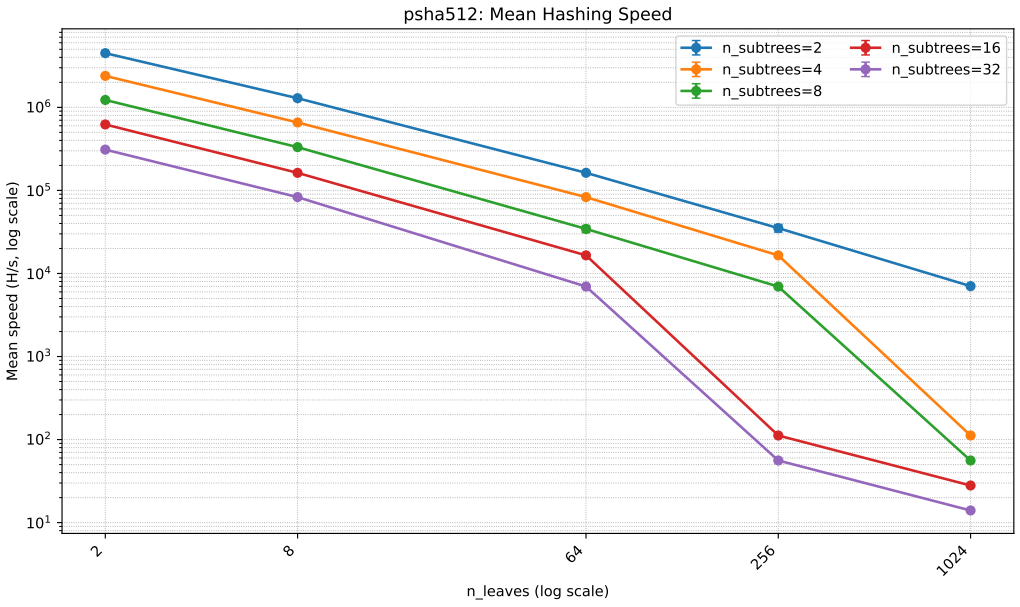


Figure 4.2: Mean Hashcat speed of `parallel_sha512` as a function of the number of leaves per subtree. Each curve represents a different number of subtrees.

Figure 4.3 and Figure 4.4 show the corresponding results for `parallel_sha512_it`, where in this case the speed depends on the number of worker-threads and on the number of rounds. For this algorithm as well, the hashing rate decreases as both the number of rounds and of worker-threads increases. The effect of the number of rounds on the execution time becomes more evident for larger values. For instance, considering the case of 2 worker-threads, the speeds recorded decrease from several million hashes per second for a small number of rounds to approximately 10-100 hashes per second as the rounds increase.

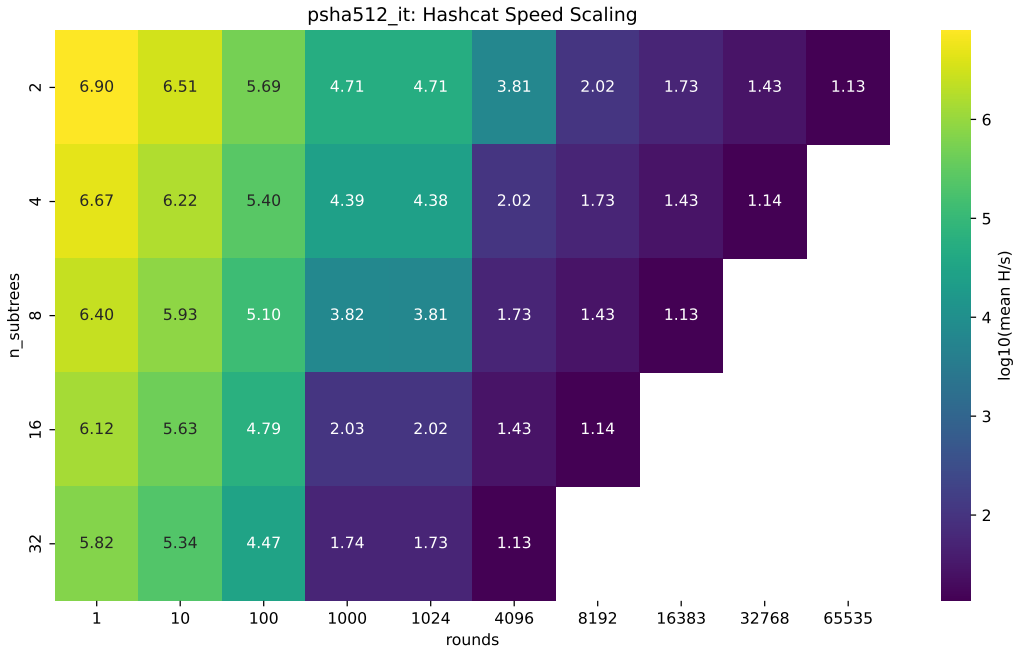


Figure 4.3: Logarithm of the mean Hashcat speed of `parallel_sha512_it` for different numbers of subtrees and iterative rounds.

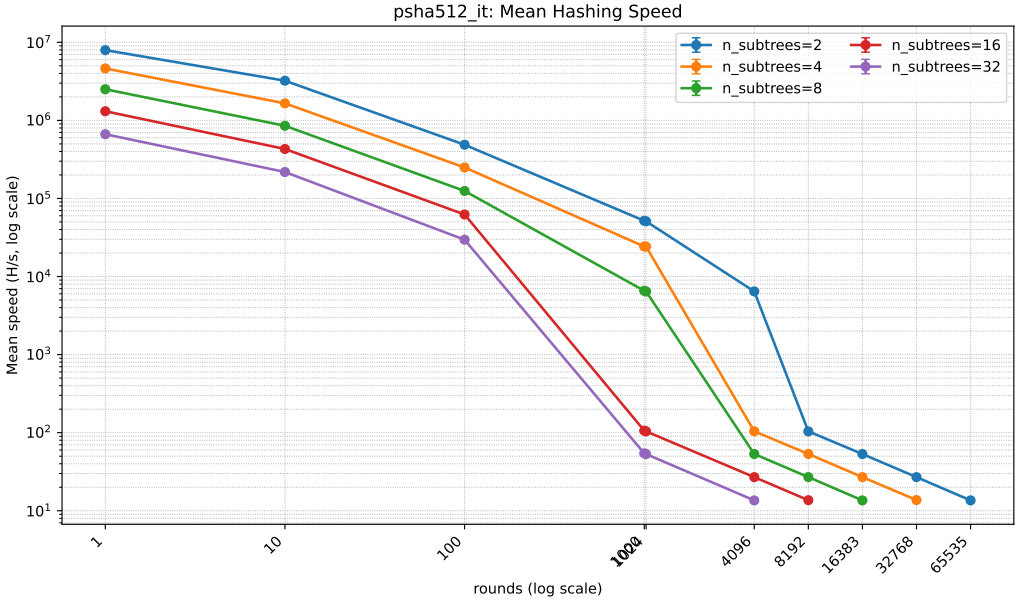


Figure 4.4: Mean Hashcat speed of `parallel_sha512_it` as a function of the number of iterative rounds. Each curve represents a different number of subtrees.

Overall, the hashcat performance of both algorithms strongly depends on their respective configuration parameters. Generally, increasing either parameter increases the computational work required per candidate and consequently reduces throughput. Considering that hashcat assigns to each work-item a different password candidate, this behavior is expected. Indeed, increasing the algorithmic work factor for each work-item directly reduces the number of assessable candidates

per second.

In order to compare the `parallel_sha512` and `parallel_sha512_it` hashcat integration performance with the performance of a conventional baseline such as raw SHA-512, an additional experiment on hashcat mode 1700 was conducted. The measurement was performed using the same `-speed-only` workflow and was repeated 50 times. On the tested hardware, hashcat processed approximately 13578348 raw SHA-512 candidates per second.

Figure 4.5 reports the throughput slowdown of the most computationally expensive `parallel_sha512` and `parallel_sha512_it` configurations relative to raw SHA-512. For a custom configuration, the slowdown factor is computed as the ratio between the mean throughput measured for hashcat mode 1700, i.e. 13578348 candidates evaluated per second, and the mean throughput measured for each algorithm.

The selected configurations exhibit slowdown factors between approximately 2.55×10^5 and 1.00×10^6. For both algorithms, input configurations that require approximately the same number of total SHA-512 hashes exhibit very similar slowdown factors. For instance, several `parallel_sha512_it` configurations whose parameters result in approximately 131071 SHA-512 computations per candidate have slowdown factors close to 10^6.

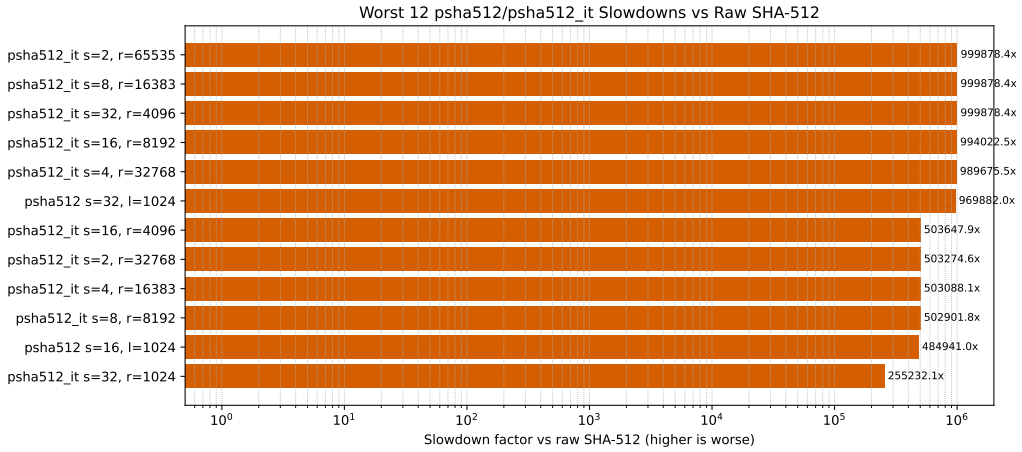


Figure 4.5: Throughput slowdown of the most computationally expensive `parallel_sha512` and `parallel_sha512_it` configurations relative to raw SHA-512.

To obtain a more informative comparison, the slowdown factors reported in Figure 4.6 were computed by dividing the values presented in Figure 4.5 by the total number of SHA-512 hashes computed by each algorithm configuration. For the `parallel_sha512` algorithm, with s subtrees and l leaves per subtree, the total number of SHA-512 computations is given by

$$N_{\texttt{parallel\_sha512}} = 2 \cdot s \cdot l - 1.$$

For the `parallel_sha512_it` algorithm, with s subtrees and r rounds per subtree, the total number of SHA-512 computations is given by

$$N_{\texttt{parallel\_sha512\_it}} = s \cdot r + s - 1.$$

Figure 4.6 shows that the slowdown factor of the selected `parallel_sha512_it` configurations ranges from approximately 7.55× to 7.78×. By contrast, the two selected `parallel_sha512` configurations both exhibit a larger slowdown of approximately 14.8×. The difference between the values of the two algorithms suggests that the leaf-generation and tree-reduction structure of `parallel_sha512` introduces a higher cost for each SHA-512 computation than the parallel-iterative structure of `parallel_sha512_it`.

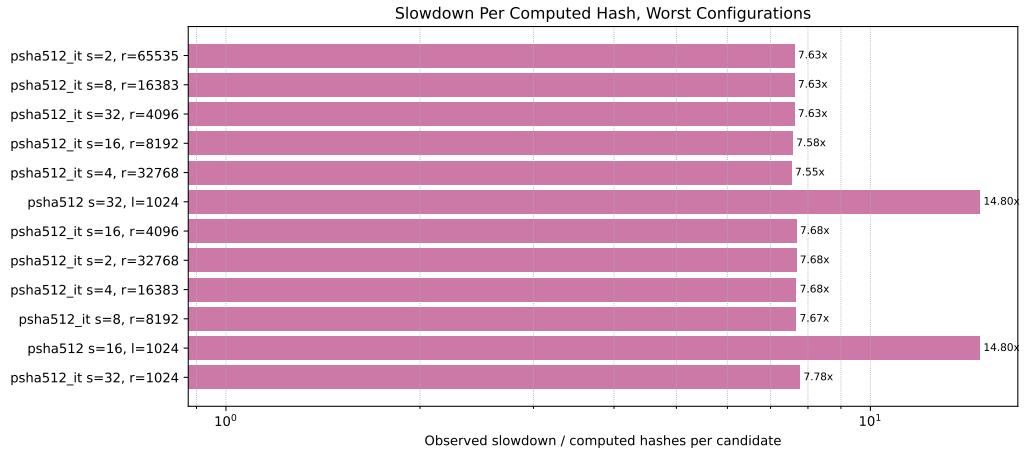


Figure 4.6: Slowdown relative to raw SHA-512 normalized by the estimated number of internal SHA-512 computations per candidate.

Overall, the experiments confirm that increasing the number of subtrees, leaves, or iterative rounds progressively reduces the number of password candidates that can be evaluated per second. Moreover, even for configurations that require a comparable number of internal SHA-512 computations, the proposed algorithms show a higher computational cost than raw SHA-512. This may be mainly due to the additional operations required for input construction, iterative processing and Merkle-tree reduction. On the tested hardware and the evaluated implementations, this results in a lower password-candidate evaluation rate than raw SHA-512.

These findings support the motivation of the proposed construction from the attacker's perspective. Hashcat already exploits GPU parallelism by evaluating many password candidates concurrently; therefore, increasing the computational work associated with each candidate consumes a larger share of the available processing resources and reduces the achievable guessing throughput. The experiments consequently show that the additional work introduced by the proposed constructions cannot be entirely hidden through candidate-level parallelism. However, the measured slowdown is specific to the tested hardware, hashcat kernels, and parameter configurations and should not be interpreted as a direct measure of cryptographic security.

Chapter 5. Security evaluation of `parallel_sha512` and `parallel_sha512_it`

This chapter presents a security evaluation of the proposed hashing constructions. All algorithms use SHA-512 as their underlying cryptographic primitive and do not introduce new compression functions or rely on new cryptographic assumptions. Therefore, their security inherits the SHA-512 fundamental cryptographic properties: preimage resistance, second-preimage resistance, and collision resistance. However, these properties alone are not sufficient to establish the security of the complete constructions, since the effects introduced by the Merkle-tree organization must also be considered.

The analysis presented in this chapter focuses on a specific integrity property: the probability that the final Merkle root remains unchanged after the modification of a leaf. For this purpose, the probability model proposed in [9] is applied to the tree sizes and parameter configurations used in the experimental evaluation of `parallel_sha512` and `parallel_sha512_it`. The resulting estimates are used to assess whether the Merkle-tree structure introduces a significant probability that a modification remains undetected.

Nevertheless, this analysis does not constitute a complete cryptographic proof of the proposed password-hashing constructions. Fundamental properties such as preimage resistance, second-preimage resistance, collision resistance, parameter selection, and resistance to possible structural attacks would require additional investigation.

In [9], the authors present the following formula to compute the probability that, the Merkle root remains unchanged after modifying a leaf of the tree.

$$P_{falsification} = \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right)\left(1 - \left(1 - \frac{1}{2^b}\right)^m\right) \tag{5.1}$$

where:

- b is the digest length, in bits, of the chosen cryptographic hash function;
- m is the number of elements in the Merkle path $P(D_i)$, so it can be seen as its cardinality $m = |P(D_i)|$.

Equation 5.1 can be rewritten as follows:

$$\begin{aligned} P_{falsification} &= \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right)\left(1 - \left(1 - \frac{1}{2^b}\right)^m\right) \\ &= \frac{1}{2^b} + 1 - \left(1 - \frac{1}{2^b}\right)^m - \frac{1}{2^b} + \frac{1}{2^b}\left(1 - \frac{1}{2^b}\right)^m \\ &= 1 - \left(1 - \frac{1}{2^b}\right)^m + \frac{1}{2^b}\left(1 - \frac{1}{2^b}\right)^m \\ &= 1 - \left(1 - \frac{1}{2^b}\right)\left(1 - \frac{1}{2^b}\right)^m \\ &= 1 - \left(1 - \frac{1}{2^b}\right)^{m+1} \end{aligned} \tag{5.2}$$

To evaluate this probability numerically, [9] adopts the following approximation. For sufficiently small x, we have $1-x \approx e^{-x}$. Since $\frac{1}{2^b}$ is extremely small for SHA-512:

$$e^{-\frac{1}{2^b}} \approx 1 - \frac{1}{2^b}.$$

Therefore, Equation 5.2 can be approximated as

$$P_{falsification} \approx 1 - \exp\left(-\frac{m+1}{2^b}\right), \tag{5.3}$$

using:

$$\left(1 - \frac{1}{2^b}\right)^{m+1} \approx \left(e^{-\frac{1}{2^b}}\right)^{m+1} = e^{-\frac{m+1}{2^b}}.$$

We recall that, for a complete binary Merkle tree with n leaves, where n is a power of two, the Merkle path length is $|P(D_i)| = \log_2(n)$. For the `parallel_sha512` algorithm, the total number of leaves is given by $n_{subtrees} \cdot n_{leaves}$. Therefore

$$m_{\texttt{parallel\_sha512}} = |P(D_i)| = \log_2(n_{subtrees} \cdot n_{leaves}).$$

For the `parallel_sha512_it` algorithm, instead, the Merkle path depends only on the number of subtrees:

$$m_{\texttt{parallel\_sha512\_it}} = |P(D_i)| = \log_2(n_{subtrees}).$$

`parallel_sha512` falsification probabilities

Running a Python script to compute the falsification probabilities using the approximation described above for the `parallel_sha512` algorithm with the same subtree and leaf input values used in the experimental setup. The numerical results are presented in Table 5.1, Table 5.2, and Table 5.3.

$n_{subtrees}$	n_{leaves}	m	$P_{falsification}$
2	8,192	14	$1.12 \cdot 10^{-153}$
2	65,536	17	$1.34 \cdot 10^{-153}$
2	262,144	19	$1.49 \cdot 10^{-153}$
2	1,048,576	21	$1.64 \cdot 10^{-153}$
2	2,097,152	22	$1.72 \cdot 10^{-153}$
2	4,194,304	23	$1.79 \cdot 10^{-153}$

Table 5.1: Falsification probabilities with 2 subtrees for the `parallel_sha512` algorithm.

$n_{subtrees}$	n_{leaves}	m	$P_{falsification}$
4	8,192	15	$1.19 \cdot 10^{-153}$
4	65,536	18	$1.42 \cdot 10^{-153}$
4	262,144	20	$1.57 \cdot 10^{-153}$
4	1,048,576	22	$1.72 \cdot 10^{-153}$
4	2,097,152	23	$1.79 \cdot 10^{-153}$
4	4,194,304	24	$1.86 \cdot 10^{-153}$

Table 5.2: Falsification probabilities with 4 subtrees for the `parallel_sha512` algorithm.

$n_{subtrees}$	n_{leaves}	m	$P_{falsification}$
8	8,192	16	$1.27 \cdot 10^{-153}$
8	65,536	19	$1.49 \cdot 10^{-153}$
8	262,144	21	$1.64 \cdot 10^{-153}$
8	1,048,576	23	$1.79 \cdot 10^{-153}$
8	2,097,152	24	$1.86 \cdot 10^{-153}$
8	4,194,304	25	$1.94 \cdot 10^{-153}$

Table 5.3: Falsification probabilities with 8 subtrees for the `parallel_sha512` algorithm.

`parallel_sha512_it` falsification probabilities

For the `parallel_sha512_it` algorithm, the falsification probabilities were computed using the same values of $n_{subtrees}$ adopted in the experimental setup. Table 5.4 reports the results obtained.

$n_{subtrees}$	m	$P_{falsification}$
2	1	$1.49 \cdot 10^{-154}$
4	2	$2.24 \cdot 10^{-154}$
8	3	$2.98 \cdot 10^{-154}$

Table 5.4: Falsification probabilities for the `parallel_sha512_it` algorithm.

Conclusions

From the results reported, the combination of $n_{subtrees}$ and n_{leaves} that minimizes $P_{falsification}$ for the `parallel_sha512` algorithm is 2 and 8192 respectively. For the `parallel_sha512_it` algorithm, $P_{falsification}$ is minimized for $n_{subtrees} = 2$. The conclusion of [9] is that increasing the depth of the Merkle tree, and therefore the Merkle path cardinality, also increases $P_{falsification}$. This claim is confirmed by the numerical results obtained in this work. All the computed probabilities are of the order of 10^{-153} for the `parallel_sha512` algorithm and of the order of 10^{-154} for the `parallel_sha512_it` algorithm. Therefore, within the scope of this analysis, the Merkle-tree reduction does not appear to introduce a practically exploitable weakness related to undetected leaf modifications.

To achieve numerical stability, the Python implementation does not compute $1 - \left(1 - \frac{1}{2^b}\right)^{m+1}$ directly. Instead, it uses the equivalent formulation $P_{falsification} = 1 - \exp\left((m+1)\ln(1 - \frac{1}{2^b})\right)$, where the term $\ln(1-a)$ is evaluated using `log1p` and the final expression $1 - \exp(x)$ is computed as $-$`expm1`(x).

Conclusions

This thesis proposed the use of Merkle trees to develop a new parallel hashing algorithm based on the SHA-512 cryptographic hash function. Conventional iterative hashing functions contain dependencies between successive operations and therefore cannot exploit the parallelism provided by modern processing units. The proposed approach consists of dividing the hashing workload into independent subtrees, whose computation is assigned to different CPU cores or GPU work-items. The resulting subtree roots are then combined by the main process through a standard Merkle tree reduction.

The construction should be interpreted as a parallel password-hashing scheme rather than as a new general-purpose cryptographic hash function. Its principal contribution is the use of server-side parallelism to transform available processing capacity into additional computational work for password verification. More specifically, the construction allows to increase the total number of SHA-512 evaluations performed for each password while maintaining the resulting authentication latency within an acceptable range. From the attacker's perspective, the increased per-candidate work reduces the password-guessing throughput achievable and therefore improves resistance against large-scale offline password-guessing attacks.

To investigate this design, the `parallel_sha512` and the `parallel_sha512_it` algorithms were implemented for both CPU and GPU architectures. The CPU implementations were developed using the C++ programming language using worker threads to handle the subtree computation, whereas the GPU versions were implemented in the OpenCL framework. An additional variant of these algorithm based on the `sha512crypt` function was also developed, together with an iterative SHA-512-based hashing algorithm used as comparison baseline.

Both a theoretical analysis and experimental measurements were conducted to evaluate the algorithms performance. The execution times of the new parallel approaches were compared with those of traditional iterative approaches, and the effects of the various input parameters, e.g. number of worker-threads, leaves, rounds and GPU work-items, were examined.

The proposed algorithms were also integrated into the hashcat framework, implementing custom modules and OpenCL kernels, in order to evaluate their behavior in a practical password-cracking scenario. Finally, a theoretical security evaluation was conducted. The integrity properties of the Merkle-tree construction were studied by estimating the probability that the Merkle root remains unchanged when a leaf is modified.

The CPU experiments showed that the proposed Merkle tree-based approach can effectively reduce the execution time of the hashing process by distributing the workload on multiple worker-threads. All the parallel implementations consistently outperformed their corresponding iterative baselines. However, the speed-ups measured in the experimental results were lower than the expected values predicted by the theoretical analysis. This difference may be due to thread management, memory handling, and synchronization operations that were not considered in the ideal model.

`parallel_sha512_it` achieved the best overall performance among the tested CPU implementations, having execution times consistently lower than the other variants and than the corresponding iterative baselines. This proved how distributing an iterative hashing workload among independent worker-threads that perform iterative hashing can effectively improve the performance. However, since the physical hardware on which the tests were conducted provided only six physical CPU cores, input configurations using eight worker threads where affected by additional scheduling overhead.

The GPU evaluation produced different results depending on the algorithm and on the workload. For the `parallel_sha512` algorithm, the CPU was faster for small and medium workloads because, even though the GPU implementation provided massive parallelism, the fixed overhead were too large to be amortized. This interpretation is supported by the comparison between the total GPU execution time and the kernel-only execution time. The two implementations reached comparable

execution times at approximately 1048576 leaves. For large workloads, the GPU became consistently faster. For instance, for 4194304 leaves the resulting speed-up was approximately 2.5×, showing that for a sufficiently large workload, the GPU version of `parallel_sha512` can exploit a greater degree of parallelism.

Considering the GPU implementations of `parallel_sha512_it` and `sha512_it` the results were different and they did not outperform their respective CPU counterpart. Both algorithms are indeed characterized by an iterative part that did not exploit the available GPU parallelism. `parallel_sha512_it` performs iterative hashing inside each work-item, while `sha512_it` launches a unique work-item performing iterative hashing. This resulted in very high GPU execution times for both algorithms. However, when considering the relative speed-ups between the iterative and parallel variants, the GPU execution times followed the theoretical expected values more closely than the CPU. This suggests that the GPU execution model reflects the algorithmic parallelism more effectively.

The hashcat integration and test of `parallel_sha512` and of `parallel_sha512_it` provided an additional evaluation from the password-cracking perspective. On the tested hardware, hashcat mode 1700, corresponding to raw SHA-512, processed approximately 13.58 million password candidates per second. The custom modes became progressively slower as the value of the algorithms' parameters increased.

For the most computationally expensive configurations, after normalization by the number of internal SHA-512 computations, `parallel_sha512_it` showed a relatively consistent slowdown of approximately 7.55× to 7.78×, whereas `parallel_sha512` showed a slowdown of approximately 14.8×. Generally, configurations requiring a similar number of SHA-512 computations produced similar slowdown values. The higher slowdown factor of `parallel_sha512` is very likely associated with leaf construction, data management and Merkle tree reduction operations.

From a security perspective, the proposed constructions use SHA-512 as their underlying cryptographic primitive and do not rely on new cryptographic assumptions. Therefore, their security depends in part on the preimage resistance, second-preimage resistance, and collision resistance expected from SHA-512.

The theoretical analysis conducted in this thesis focused on one specific integrity property of the Merkle-tree organization: the probability that the final root remains unchanged after the modification of a leaf. For all parameter configurations considered, the estimated probability was extremely small for both `parallel_sha512` and `parallel_sha512_it`. Therefore, the proposed Merkle tree organization and reduction does not introduce a significant probability that a modified leaf remains undetected. However, this result addresses only one security aspect of the proposed constructions and should not be interpreted as a complete cryptographic proof. The main practical contribution evaluated in this thesis is instead the increase in the computational effort required to process each password candidate.

Overall, the experimental results provide a partial support for the central research hypothesis: server-side parallelism can be exploited to increase the total hashing work performed for each password candidate without producing a proportional increase in verification latency, provided that the number of worker threads does not exceed the parallel processing capacity of the hardware adopted. However, the effectiveness of this approach strongly depends on various factors such as the selected algorithm, the workload size and distribution, and the execution architecture. `parallel_sha512_it` represents the most promising tested variant for CPU execution, whereas `parallel_sha512` benefits more clearly from GPU parallelism with sufficiently large workloads.

The results reported in this thesis are subject to several limitations. First of all, the crossover point between CPU and GPU execution and the measured speed-ups are specific to the tested hardware. Second, the GPU kernels that were implemented were not optimized for every device. Therefore, further improvements and optimizations are still possible leveraging different OpenCL characteristics such as vectorized implementations and memory layouts.

The theoretical performance evaluation assumes ideal parallelism and discards important factors related to how actual computers work. This explains the difference between the predicted and measured speed-ups.

The hashcat comparison also reflects the efficiency of the specific custom modules and kernels tested on the specific hardware. Therefore, the slowdown recorded may vary depending on the testing device. Furthermore, the slowdown represents an implementation-level metric and should not be interpreted as a measurement of cryptographic security.

Finally, to complete the security evaluation and to have a more detailed security proof, a more complete cryptographic assessment would also be needed.

Future work could focus on extending the experimental evaluation to a broader and more recent range of CPUs and GPUs. An additional and interesting experiment would be to test the proposed algorithms on Intel processors

supporting the newer SHA-512 instruction set in order to assess how hardware acceleration affects the performance of both iterative baseline and parallel constructions. Furthermore, GPU kernels could be further optimized by exploiting more advanced OpenCL features that could produce a performance improvement. Finally, the hashcat integration could also be extended and tested more in depth. Additional kernel optimizations could be introduced and different attack modes could be implemented and deeply tested. A broader cryptographic analysis would also be necessary before considering the proposed algorithms for use in real-world security applications.

Declaration on the Use of Artificial Intelligence (AI) Tools

I, the undersigned, Giovanni Ostanello, student identification number 892631, enrolled at Ca' Foscari University of Venice in the Master's Degree Programme in Computer Science and Information Technology, author of the Master's thesis entitled *A Multithreaded SHA-512-Based Hashing Algorithm Using Merkle Trees*, supervised by Flaminia Luccio and Riccardo Focardi, being aware of the penalties provided for under Article 76 of Presidential Decree No. 445 of 28 December 2000 and of the forfeiture of benefits provided for under Article 75 of the same Presidential Decree in the event of false or misleading declarations, under my personal responsibility pursuant to Articles 46 and 47 of Presidential Decree No. 445 of 28 December 2000, declare that I used generative artificial intelligence technologies to support various aspects of the research and writing of my Master's thesis, specifically as follows:

- ChatGPT-5.5 for the revision and optimization of texts to improve clarity and coherence.

I further declare that all AI-generated material was carefully verified, reviewed, and revised by the author, who assumes full responsibility for the quality, accuracy, and scientific integrity of the work.

Chapter A. Appendix

A.1 `parallel_sha512` code

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <thread>
#include <openssl/sha.h>
#include <cstring>
#include <stdlib.h>

using namespace std;
#define BYTES 4

void parallel_hash(const unsigned char* str, size_t str_len, unsigned int size, unsigned
    int start, unsigned int number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {

    vector<array<unsigned char, SHA512_DIGEST_LENGTH>> results(size);
    unsigned char bytes[BYTES];

    // Little-endian encoding (least significant byte LSB first)
    bytes[3] = start & 0xFF;                // 8 bits least significant
    bytes[2] = (start >> 8) & 0xFF;         // intermediate byte
    bytes[1] = (start >> 16) & 0xFF;        // intermediate byte
    bytes[0] = (start >> 24) & 0xFF;        // most significant byte

    SHA512_CTX ctx;
    for(size_t i = 0; i < size; ++i) {
        // Convert index to string without using std::string
        bytes[3]++;
        if(bytes[3] == 0) {             // first byte overflow
            bytes[2]++;
            if(bytes[2] == 0) {         // second byte overflow
                bytes[1]++;
                if(bytes[1] == 0)       // third byte overflow
                    bytes[0]++;
            }
        }
```

```
37
38          SHA512_Init(&ctx);
39          SHA512_Update(&ctx, str, str_len);
40          SHA512_Update(&ctx, bytes, BYTES);
41          SHA512_Final(results[i].data(), &ctx);
42      }
43
44      // Reduce the hashes to a single hash
45      size_t active_hashes = size;
46      while(active_hashes > 2) {
47          for(size_t i = 0; i < active_hashes / 2; ++i) {
48              SHA512_Init(&ctx);
49              SHA512_Update(&ctx, results[2 * i].data(), SHA512_DIGEST_LENGTH);
50              SHA512_Update(&ctx, results[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
51              SHA512_Final(results[i].data(), &ctx);
52          }
53          if(active_hashes % 2 != 0) {
54              results[active_hashes / 2] = results[active_hashes - 1];
55              active_hashes = active_hashes / 2 + 1;
56          } else {
57              active_hashes = active_hashes / 2;
58          }
59      }
60
61      SHA512_Init(&ctx);
62      SHA512_Update(&ctx, results[0].data(), SHA512_DIGEST_LENGTH);
63      SHA512_Update(&ctx, results[1].data(), SHA512_DIGEST_LENGTH);
64      SHA512_Final(hashes[number].data(), &ctx);
65  }
66
67  int main(int argc, char* argv[]) {
68      if(argc != 5) {
69          cerr << "Usage: " << argv[0] << " <pwd> <salt> <n_cores> <n_leaves>" << endl;
70          return 1;
71      }
72
73      const char* pwd = argv[1];
74      const char* salt = argv[2];
75      unsigned int n_cores = atoi(argv[3]);
76      unsigned int n_leaves = atoi(argv[4]);
77      vector<array<unsigned char, SHA512_DIGEST_LENGTH>> hashes(n_cores);
78      vector<thread> threads;
79      const size_t pwd_len = strlen(pwd), salt_len = strlen(salt);
80      vector<unsigned char> pwd_salt(pwd_len + salt_len);
81      memcpy(pwd_salt.data(), pwd, pwd_len);
82      memcpy(pwd_salt.data() + pwd_len, salt, salt_len);
83
84      for (size_t i = 0; i < n_cores; ++i) {
```

```
        threads.emplace_back(parallel_hash, pwd_salt.data(), pwd_len + salt_len,
        ↪  n_leaves, i * n_leaves, i, ref(hashes));
    }

    for (auto& thread : threads)
        thread.join();

    // final Merkle tree reduction
    size_t active_hashes = n_cores;
    SHA512_CTX ctx;
    while (active_hashes > 1) {
        for (size_t i = 0; i < active_hashes / 2; ++i) {
            SHA512_Init(&ctx);
            SHA512_Update(&ctx, hashes[2 * i].data(), SHA512_DIGEST_LENGTH);
            SHA512_Update(&ctx, hashes[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
            SHA512_Final(hashes[i].data(), &ctx);
        }
        if (active_hashes % 2 != 0) {
            hashes[active_hashes / 2] = hashes[active_hashes - 1];
            active_hashes = active_hashes / 2 + 1;
        } else {
            active_hashes = active_hashes / 2;
        }
    }

    // output printing
    cout << hex << setfill('0');
    for (size_t i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
        cout << setw(2) << static_cast<int>(hashes[0][i]);
    }
    cout << endl;
    return 0;
}
```

Listing A.22: `parallel_sha512` C++ implementation.

A.2 `parallel_sha512_it` code

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <thread>
#include <openssl/sha.h>
#include <cstring>

using namespace std;

```

```cpp
void parallel_hash(unsigned char* str, size_t str_len, unsigned int rounds, unsigned int
    ↪ number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {

    char bytes[4];
    int n = number + 1;
    // Little-endian encoding (least significant byte LSB first)
    bytes[3] = n & 0xFF;                // 8 bits least significant
    bytes[2] = (n >> 8) & 0xFF;         // intermediate byte
    bytes[1] = (n >> 16) & 0xFF;        // intermediate byte
    bytes[0] = (n >> 24) & 0xFF;        // most significant byte

    SHA512_CTX ctx;
    SHA512_Init(&ctx);
    SHA512_Update(&ctx, str, str_len);
    SHA512_Update(&ctx, bytes, 4);
    SHA512_Final(hashes[number].data(), &ctx);

    for (unsigned long i = 1; i < rounds; ++i) {
        SHA512_Init(&ctx);
        SHA512_Update(&ctx, hashes[number].data(), SHA512_DIGEST_LENGTH);
        SHA512_Final(hashes[number].data(), &ctx);
    }
}

int main(int argc, char* argv[]) {
    // checking for correct invocation
    if (argc != 5) {
        cerr << "Usage: " << argv[0] << " <pwd> <salt> <n_cores> <n_rounds>" << endl;
        return 1;
    }

    const char* pwd = argv[1];
    const char* salt = argv[2];
    char* endptr;

    unsigned int n_cores = strtoul(argv[3], &endptr, 10);
    if (*endptr != '\0' || n_cores == 0) {
        cerr << "Invalid number for n_cores: " << argv[3] << endl;
        return 1;
    }

    unsigned int n_rounds = strtoul(argv[4], &endptr, 10);
    if (*endptr != '\0' || n_rounds == 0) {
        cerr << "Invalid number for n_rounds: " << argv[4] << endl;
        return 1;
    }

    vector<array<unsigned char, SHA512_DIGEST_LENGTH>> hashes(n_cores);
```

```
    vector<thread> threads;
    const size_t pwd_len = strlen(pwd), salt_len = strlen(salt);
    vector<unsigned char> pwd_salt(pwd_len + salt_len);
    memcpy(pwd_salt.data(), pwd, pwd_len);
    memcpy(pwd_salt.data() + pwd_len, salt, salt_len);

    for (size_t i = 0; i < n_cores; ++i)
        threads.emplace_back(parallel_hash, pwd_salt.data(), pwd_len + salt_len,
        ↪ n_rounds, i, ref(hashes));

    for (auto& thread : threads)
        thread.join();

    size_t active_hashes = n_cores;
    SHA512_CTX ctx;
    while (active_hashes > 1) {
        for (size_t i = 0; i < active_hashes / 2; ++i) {
            SHA512_Init(&ctx);
            SHA512_Update(&ctx, hashes[2 * i].data(), SHA512_DIGEST_LENGTH);
            SHA512_Update(&ctx, hashes[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
            SHA512_Final(hashes[i].data(), &ctx);
        }
        if (active_hashes % 2 != 0) {
            hashes[active_hashes / 2] = hashes[active_hashes - 1];
            active_hashes = active_hashes / 2 + 1;
        } else {
            active_hashes = active_hashes / 2;
        }
    }

    // output printing
    cout << hex << setfill('0');
    for (size_t i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
        cout << setw(2) << static_cast<int>(hashes[0][i]);
    }
    cout << endl;
    return 0;
}
```

Listing A.23: `parallel_sha512_it` C++ implementation.

A.3 `parallel_sha512_it_crypt` code

```
void parallel_hash(const char* pwd_str, const char* salt_str, unsigned int rounds,
↪ unsigned int number,
        vector<array<unsigned char, SHA512_DIGEST_LENGTH>>& hashes) {

    string salt = "$6$rounds=" + to_string(rounds) + "$" + salt_str;
```

```
5
6       crypt_data data {};
7       const char* hash = crypt_r(pwd_str, salt.c_str(), &data);
8       if (hash == nullptr) {
9           perror("crypt_r");
10          return;
11      }
12
13      SHA512(reinterpret_cast<const unsigned char*>(hash), strlen(hash),
            ↪ hashes[number].data());
14  }
```

Listing A.24: `parallel_sha512_it_crypt` `parallel_hash` function implementation.

A.4 `parallel_sha512` OpenCL kernel code

```
1   #define IS_OPENCL
2   #define KERNEL_STATIC
3   #define BYTES 4
4   #define SHA512_DIGEST_LENGTH 64
5   #define MAX_LEAVES_PER_SUBTREE 1024
6
7   #include "inc_vendor.h"
8   #include "inc_types.h"
9   #include "inc_platform.cl"
10  #include "inc_common.cl"
11  #include "inc_hash_sha512.cl"
12  #include "inc_utilities_cl.h"
13
14  /**
15      Assuming that all subtrees have the same number of leaves, then the initial leaf
            ↪ counter value
16      can be computed using the following formula: gid * size, where size is the number of
            ↪ leaves that
17      each subtree has. One parameter less!
18   */
19  KERNEL_FQ void psha512(GLOBAL_AS const u32 *pwd_salt, const u32 pwd_salt_len, const u32
        ↪ size, GLOBAL_AS u8 *hashes) {
20      // retrieve thread unique id to access to the correct output buffer location
21      const u64 gid = get_global_id(0);
22      // max leaves check
23      if(size > MAX_LEAVES_PER_SUBTREE) return;
24
25      const u32 start = (u32)gid * size;
26      u32 results[MAX_LEAVES_PER_SUBTREE][SHA512_BLOCK_WORDS];
27      u32 bytes_block[SHA512_BLOCK_WORDS];
28      PRIVATE_AS u8 *bytes = (PRIVATE_AS u8 *)bytes_block;
29
```

```
    // setting the bytes_block to all 0s
    // performed through loop-unrolling: GPU optimization
    ZERO_BLOCK(bytes_block);

    // start leaf number encoding
    bytes[3] = (u8) (start & 0xff);             // 8 bits least significant
    bytes[2] = (u8) ((start >> 8) & 0xff);      // intermediate byte
    bytes[1] = (u8) ((start >> 16) & 0xff);     // intermediate byte
    bytes[0] = (u8) ((start >> 24) & 0xff);     // most significant byte

        // leaf hashing with added leaf number using byte block (as in CPU code)
    sha512_ctx_t ctx;
    for(u32 i = 0; i < size; i++) {
        bytes[3]++;
        if(bytes[3] == 0) {                     // first byte overflow
            bytes[2]++;
            if(bytes[2] == 0) {                 // second byte overflow
                bytes[1]++;
                if(bytes[1] == 0) {             // third byte overflow
                    bytes[0]++;
                }
            }
        }
        sha512_init(&ctx);
        sha512_update_global_swap(&ctx, pwd_salt, pwd_salt_len);
        sha512_update_swap(&ctx, bytes_block, BYTES);
        sha512_final(&ctx);
        // saving the hashing results into the appropriate results buffer
        // performed through loop-unrolling: GPU optimization
        STORE_DIGEST_BLOCK(results[i], ctx.h);
    }

        // Merkle tree reduction steps (same as CPU code)
    u32 active_hashes = size;
    while(active_hashes > 2) {
                for(u32 i = 0; i < active_hashes / 2; i++) {
                        sha512_init(&ctx);
                        sha512_update_swap(&ctx, results[2 * i], SHA512_DIGEST_LENGTH);
                        sha512_update_swap(&ctx, results[2 * i + 1],
                         ↪  SHA512_DIGEST_LENGTH);
                        sha512_final(&ctx);

            // saving the hashing results into the appropriate results buffer
            // performed through loop-unrolling: GPU optimization
            STORE_DIGEST_BLOCK(results[i], ctx.h);
                }
                if((active_hashes & 1) != 0) {
            // copying the block from different positions of the results array
            // performed through loop-unrolling: GPU optimization
```

```
            COPY_BLOCK(results[active_hashes / 2], results[active_hashes - 1])
                    active_hashes = active_hashes / 2 + 1;
                } else {
                    active_hashes = active_hashes / 2;
                }
    }
        // Final Merkle root hash
    sha512_init(&ctx);
    sha512_update_swap(&ctx, results[0], SHA512_DIGEST_LENGTH);
    sha512_update_swap(&ctx, results[1], SHA512_DIGEST_LENGTH);
    sha512_final(&ctx);

        // Producing the output and saving it into the output array "hashes"
    const u32 base = (u32)gid * SHA512_DIGEST_LENGTH;
    for(u32 i = 0; i < 8; i++) {
                const u64 word = ctx.h[i];
                const u32 out = base + i * 8;
                hashes[out + 0] = (u8) ((word >> 56) & 0xff);
                hashes[out + 1] = (u8) ((word >> 48) & 0xff);
                hashes[out + 2] = (u8) ((word >> 40) & 0xff);
                hashes[out + 3] = (u8) ((word >> 32) & 0xff);
                hashes[out + 4] = (u8) ((word >> 24) & 0xff);
                hashes[out + 5] = (u8) ((word >> 16) & 0xff);
                hashes[out + 6] = (u8) ((word >>  8) & 0xff);
                hashes[out + 7] = (u8) ((word >>  0) & 0xff);
    }
}
```

Listing A.25: `parallel_sha512` OpenCL kernel implementation.

A.5 `parallel_sha512` C++ main code

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <openssl/sha.h>
#include <cstring>
#include <chrono>

#include "platform_model.h"
#include "device_model.h"
#include "utilities.h"

#define MAX_LEAVES 1024
#define SHA512_BLOCK_LENGTH 128

using namespace std;
```

```cpp

int main(int argc, char** argv) {
    if(argc < 5) {
        cerr << "Usage: " << argv[0] << " <pwd> <salt> <n_cores> <n_leaves>" << endl;
        return 1;
    }

    const char* pwd = argv[1];
    const char* salt = argv[2];
    const size_t n_cores = stoul(argv[3]);
    const cl_uint n_leaves = static_cast<cl_uint>(stoul(argv[4]));

    if(n_cores == 0) {
        cerr << "[ERROR] n_cores must be greater than 0.\n";
        return 1;
    }

    if(n_leaves < 2) {
        cerr << "[ERROR] n_leaves must be at least 2.\n";
        return 1;
    }

    if(n_leaves > MAX_LEAVES) {
        cerr << "[ERROR] n_leaves exceeds MAX_LEAVES (" << MAX_LEAVES << ").\n";
        return 1;
    }

    cl_int err = CL_SUCCESS;

    //cout << "[INFO] Detecting OpenCL platform..." << endl;
    cl_platform_id platform = getPlatform();
    //cout << "[INFO] Platform: " << getPlatformName(platform) << endl;

    //cout << "[INFO] Selecting OpenCL device..." << endl;
    cl_device_id device = getDevice(platform);
    //cout << "[INFO] Device: " << getDeviceName(device) << endl;

    //cout << "[INFO] Creating OpenCL context and command queue..." << endl;
    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &err);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clCreateContext failed: " << err << "\n";
        return 1;
    }

    #pragma GCC diagnostic push
    #pragma GCC diagnostic ignored "-Wdeprecated-declarations"
    cl_command_queue queue = clCreateCommandQueue(context, device, 0, &err);
    #pragma GCC diagnostic pop
```

```
66
67      if(err != CL_SUCCESS) {
68          cerr << "[ERROR] clCreateCommandQueue failed: " << err << "\n";
69          clReleaseContext(context);
70          return 1;
71      }
72
73      vector<array<unsigned char, SHA512_DIGEST_LENGTH>> hashes(n_cores);
74
75      const size_t pwd_len = strlen(pwd);
76      const size_t salt_len = strlen(salt);
77      const cl_uint tot_len = static_cast<cl_uint>(pwd_len + salt_len);
78      const size_t padded_len = ((static_cast<size_t>(tot_len) + SHA512_BLOCK_LENGTH - 1) /
        ↪ SHA512_BLOCK_LENGTH) * SHA512_BLOCK_LENGTH;
79
80      vector<unsigned char> pwd_salt(padded_len, 0);
81      memcpy(pwd_salt.data(), pwd, pwd_len);
82      memcpy(pwd_salt.data() + pwd_len, salt, salt_len);
83
84      cl_mem pwd_salt_buf = clCreateBuffer(context, CL_MEM_READ_ONLY |
        ↪ CL_MEM_COPY_HOST_PTR,
85                                           pwd_salt.size(), pwd_salt.data(), &err);
86
87      if(err != CL_SUCCESS) {
88          cerr << "[ERROR] clCreateBuffer(pwd_salt_buf) failed: " << err << "\n";
89          clReleaseCommandQueue(queue);
90          clReleaseContext(context);
91          return 1;
92      }
93
94      cl_mem hashes_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
95                                         SHA512_DIGEST_LENGTH * n_cores, nullptr, &err);
96
97      if(err != CL_SUCCESS) {
98          cerr << "[ERROR] clCreateBuffer(hashes_buf) failed: " << err << "\n";
99          clReleaseMemObject(pwd_salt_buf);
100         clReleaseCommandQueue(queue);
101         clReleaseContext(context);
102         return 1;
103     }
104
105     const string project_root = find_project_root();
106     const string kernel_path = project_root + "/kernels/psha512.cl";
107     const string include_dir = project_root + "/include";
108     const string kernel_dir = project_root + "/kernels";
109
110     //cout << "[INFO] Loading and compiling kernel source from: " << kernel_path << endl;
111     string kernelSource = load_kernel(kernel_path);
112     const char* src = kernelSource.c_str();
```

```
    cl_program program = clCreateProgramWithSource(context, 1, &src, nullptr, &err);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clCreateProgramWithSource failed: " << err << "\n";
        clReleaseMemObject(hashes_buf);
        clReleaseMemObject(pwd_salt_buf);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        return 1;
    }

    const string build_options = string("-I ") + include_dir + " -I " + kernel_dir +
                                 " -D DGST_R0=14"
                                 " -D DGST_R1=15"
                                 " -D DGST_R2=6"
                                 " -D DGST_R3=7"
                                 " -D DGST_ELEM=16";
    err = clBuildProgram(program, 1, &device, build_options.c_str(), nullptr, nullptr);
    if(err != CL_SUCCESS) {
        cerr << "[ERROR] Kernel build failed.\n";
        size_t log_size = 0;
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, nullptr,
        ↪  &log_size);
        vector<char> log(log_size);
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size,
        ↪  log.data(), nullptr);
        cerr << "Build log:\n" << log.data() << "\n";

        clReleaseProgram(program);
        clReleaseMemObject(hashes_buf);
        clReleaseMemObject(pwd_salt_buf);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        return 1;
    }

    //cout << "[INFO] Kernel compiled successfully. Preparing for execution...\n";
    cl_kernel kernel = clCreateKernel(program, "psha512", &err);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clCreateKernel failed: " << err << "\n";
        clReleaseProgram(program);
        clReleaseMemObject(hashes_buf);
        clReleaseMemObject(pwd_salt_buf);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        return 1;
    }

```

```
160     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &pwd_salt_buf);
161     err |= clSetKernelArg(kernel, 1, sizeof(cl_uint), &tot_len);
162     err |= clSetKernelArg(kernel, 2, sizeof(cl_uint), &n_leaves);
163     err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &hashes_buf);
164 
165     if(err != CL_SUCCESS) {
166         cerr << "[ERROR] clSetKernelArg failed: " << err << "\n";
167         clReleaseKernel(kernel);
168         clReleaseProgram(program);
169         clReleaseMemObject(hashes_buf);
170         clReleaseMemObject(pwd_salt_buf);
171         clReleaseCommandQueue(queue);
172         clReleaseContext(context);
173         return 1;
174     }
175 
176     //cout << "[INFO] Launching kernel...\n";
177 
178     auto t0 = chrono::high_resolution_clock::now();
179     err = clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &n_cores, nullptr, 0,
        ↪  nullptr, nullptr);
180 
181     if(err != CL_SUCCESS) {
182         cerr << "[ERROR] clEnqueueNDRangeKernel failed: " << err << "\n";
183         clReleaseKernel(kernel);
184         clReleaseProgram(program);
185         clReleaseMemObject(hashes_buf);
186         clReleaseMemObject(pwd_salt_buf);
187         clReleaseCommandQueue(queue);
188         clReleaseContext(context);
189         return 1;
190     }
191 
192     clFinish(queue);
193 
194     auto t1 = chrono::high_resolution_clock::now();
195 
196     double elapsed_ns = chrono::duration<double>(t1 - t0).count();
197     cout << "[TIME] GPU phase: " << fixed << setprecision(10) << elapsed_ns << " s\n";
198 
199 
200     //cout << "[INFO] Reading back results from device...\n";
201     err = clEnqueueReadBuffer(queue, hashes_buf, CL_TRUE, 0,
202                               SHA512_DIGEST_LENGTH * n_cores, hashes.data(), 0, nullptr,
                                  ↪  nullptr);
203 
204     if(err != CL_SUCCESS) {
205         cerr << "[ERROR] clEnqueueReadBuffer failed: " << err << "\n";
206         clReleaseKernel(kernel);
```

```
207         clReleaseProgram(program);
208         clReleaseMemObject(hashes_buf);
209         clReleaseMemObject(pwd_salt_buf);
210         clReleaseCommandQueue(queue);
211         clReleaseContext(context);
212         return 1;
213     }
214 
215     //cout << "[INFO] Cleaning up OpenCL resources...\n";
216     clReleaseMemObject(pwd_salt_buf);
217     clReleaseMemObject(hashes_buf);
218     clReleaseKernel(kernel);
219     clReleaseProgram(program);
220     clReleaseCommandQueue(queue);
221     clReleaseContext(context);
222 
223     size_t active_hashes = n_cores;
224     SHA512_CTX ctx;
225     while(active_hashes > 1) {
226         for(size_t i = 0; i < active_hashes / 2; ++i) {
227             SHA512_Init(&ctx);
228             SHA512_Update(&ctx, hashes[2 * i].data(), SHA512_DIGEST_LENGTH);
229             SHA512_Update(&ctx, hashes[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
230             SHA512_Final(hashes[i].data(), &ctx);
231         }
232 
233         if(active_hashes % 2 != 0) {
234             hashes[active_hashes / 2] = hashes[active_hashes - 1];
235             active_hashes = active_hashes / 2 + 1;
236         } else {
237             active_hashes = active_hashes / 2;
238         }
239     }
240 
241     //cout << "$psha512$n_subtrees=" << n_cores << "$n_leaves=" << n_leaves << "$" <<
      ↪  salt << "$";
242     cout << hex << setfill('0');
243     for(size_t i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
244         cout << setw(2) << static_cast<int>(hashes[0][i]);
245     }
246     cout << endl;
247     return 0;
248 }
```

Listing A.26: `parallel_sha512` C++ main implementation.

A.6 `parallel_sha512_it` OpenCL kernel code

```
#define IS_OPENCL
#define KERNEL_STATIC
#define BYTES 4
#define SHA512_DIGEST_LENGTH 64

#include "inc_vendor.h"
#include "inc_types.h"
#include "inc_platform.cl"
#include "inc_common.cl"
#include "inc_hash_sha512.cl"
#include "inc_utilities_cl.h"

KERNEL_FQ void psha512_it_sha(GLOBAL_AS const u32 *pwd_salt, const u32 pwd_salt_len,
 ↪  const u32 rounds, GLOBAL_AS u8 *hashes) {
    // the gid + 1 is the leaf number to concatenate to pwd_salt
    const u64 gid = get_global_id(0);
    const u32 start = (u32)gid + 1;
    u32 results[SHA512_BLOCK_WORDS];
    u32 bytes_block[SHA512_BLOCK_WORDS];
    PRIVATE_AS u8 *bytes = (PRIVATE_AS u8 *)bytes_block;

    // setting the bytes_block to all 0s
    // performed through loop-unrolling: GPU optimization
    ZERO_BLOCK(bytes_block);

    // start leaf number encoding
    bytes[3] = (u8) (start & 0xff);             // 8 bits least significant
    bytes[2] = (u8) ((start >> 8) & 0xff);      // intermediate byte
    bytes[1] = (u8) ((start >> 16) & 0xff);     // intermediate byte
    bytes[0] = (u8) ((start >> 24) & 0xff);     // most significant byte

    // first hash to the pwd_salt with the leaf number concatenated
    sha512_ctx_t ctx;
    sha512_init(&ctx);
    sha512_update_global_swap(&ctx, pwd_salt, pwd_salt_len);
    sha512_update_swap(&ctx, bytes_block, BYTES);
    sha512_final(&ctx);

    // saving the hashing results into the appropriate results buffer
    // performed through loop-unrolling: GPU optimization
    STORE_DIGEST_BLOCK(results, ctx.h);

    // rounds loop
    for(u64 i = 1; i < rounds; ++i) {
        sha512_init(&ctx);
        sha512_update_swap(&ctx, results, SHA512_DIGEST_LENGTH);
        sha512_final(&ctx);
```

```
47          // saving the hashing results into the appropriate results buffer
48          // performed through loop-unrolling: GPU optimization
49          STORE_DIGEST_BLOCK(results, ctx.h);
50      }
51
52      // Producing the output and saving it into the output array "hashes"
53      const u32 base = (u32)gid * SHA512_DIGEST_LENGTH;
54      for(u32 i = 0; i < 8; i++) {
55                  const u64 word = ctx.h[i];
56                  const u32 out = base + i * 8;
57                  hashes[out + 0] = (u8) ((word >> 56) & 0xff);
58                  hashes[out + 1] = (u8) ((word >> 48) & 0xff);
59                  hashes[out + 2] = (u8) ((word >> 40) & 0xff);
60                  hashes[out + 3] = (u8) ((word >> 32) & 0xff);
61                  hashes[out + 4] = (u8) ((word >> 24) & 0xff);
62                  hashes[out + 5] = (u8) ((word >> 16) & 0xff);
63                  hashes[out + 6] = (u8) ((word >>  8) & 0xff);
64                  hashes[out + 7] = (u8) ((word >>  0) & 0xff);
65      }
66  }
```

Listing A.27: `parallel_sha512_it` OpenCL kernel implementation.

A.7 `parallel_sha512_it` C++ main code

```
1   #include <iostream>
2   #include <iomanip>
3   #include <vector>
4   #include <array>
5   #include <openssl/sha.h>
6   #include <cstring>
7   #include <chrono>
8
9   #include "platform_model.h"
10  #include "device_model.h"
11  #include "utilities.h"
12
13  #define SHA512_BLOCK_LENGTH 128
14
15  using namespace std;
16
17  int main(int argc, char** argv) {
18      if(argc < 5) {
19          cerr << "Usage: " << argv[0] << " <pwd> <salt> <n_cores> <n_rounds>" << endl;
20          return 1;
21      }
22
23      const char* pwd = argv[1];
```

```
    const char* salt = argv[2];
    const size_t n_cores = stoul(argv[3]);
    const cl_uint n_rounds = static_cast<cl_uint>(stoul(argv[4]));

    if(n_cores == 0) {
        cerr << "[ERROR] n_cores must be greater than 0.\n";
        return 1;
    }

    cl_int err = CL_SUCCESS;

    //cout << "[INFO] Detecting OpenCL platform..." << endl;
    cl_platform_id platform = getPlatform();
    //cout << "[INFO] Platform: " << getPlatformName(platform) << endl;

    //cout << "[INFO] Selecting OpenCL device..." << endl;
    cl_device_id device = getDevice(platform);
    //cout << "[INFO] Device: " << getDeviceName(device) << endl;

    //cout << "[INFO] Creating OpenCL context and command queue..." << endl;
    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &err);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clCreateContext failed: " << err << "\n";
        return 1;
    }

    #pragma GCC diagnostic push
    #pragma GCC diagnostic ignored "-Wdeprecated-declarations"
    cl_command_queue queue = clCreateCommandQueue(context, device, 0, &err);
    #pragma GCC diagnostic pop

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clCreateCommandQueue failed: " << err << "\n";
        clReleaseContext(context);
        return 1;
    }

    vector<array<unsigned char, SHA512_DIGEST_LENGTH>> hashes(n_cores);

    const size_t pwd_len = strlen(pwd);
    const size_t salt_len = strlen(salt);
    const cl_uint tot_len = static_cast<cl_uint>(pwd_len + salt_len);
    const size_t padded_len = ((static_cast<size_t>(tot_len) + SHA512_BLOCK_LENGTH - 1) /
     ↪  SHA512_BLOCK_LENGTH) * SHA512_BLOCK_LENGTH;

    vector<unsigned char> pwd_salt(padded_len, 0);
    memcpy(pwd_salt.data(), pwd, pwd_len);
    memcpy(pwd_salt.data() + pwd_len, salt, salt_len);
```

```
72
73      cl_mem pwd_salt_buf = clCreateBuffer(context, CL_MEM_READ_ONLY |
        ↪  CL_MEM_COPY_HOST_PTR,
74                                  pwd_salt.size(), pwd_salt.data(), &err);
75
76      if(err != CL_SUCCESS) {
77          cerr << "[ERROR] clCreateBuffer(pwd_salt_buf) failed: " << err << "\n";
78          clReleaseCommandQueue(queue);
79          clReleaseContext(context);
80          return 1;
81      }
82
83      cl_mem hashes_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
84                                  SHA512_DIGEST_LENGTH * n_cores, nullptr, &err);
85
86      if(err != CL_SUCCESS) {
87          cerr << "[ERROR] clCreateBuffer(hashes_buf) failed: " << err << "\n";
88          clReleaseMemObject(pwd_salt_buf);
89          clReleaseCommandQueue(queue);
90          clReleaseContext(context);
91          return 1;
92      }
93
94      const string project_root = find_project_root();
95      const string kernel_path = project_root + "/kernels/psha512_it_sha.cl";
96      const string include_dir = project_root + "/include";
97      const string kernel_dir = project_root + "/kernels";
98
99      //cout << "[INFO] Loading and compiling kernel source from: " << kernel_path << endl;
100     string kernelSource = load_kernel(kernel_path);
101     const char* src = kernelSource.c_str();
102     cl_program program = clCreateProgramWithSource(context, 1, &src, nullptr, &err);
103
104     if(err != CL_SUCCESS) {
105         cerr << "[ERROR] clCreateProgramWithSource failed: " << err << "\n";
106         clReleaseMemObject(hashes_buf);
107         clReleaseMemObject(pwd_salt_buf);
108         clReleaseCommandQueue(queue);
109         clReleaseContext(context);
110         return 1;
111     }
112
113     const string build_options = string("-I ") + include_dir + " -I " + kernel_dir +
114                                  " -D DGST_R0=14"
115                                  " -D DGST_R1=15"
116                                  " -D DGST_R2=6"
117                                  " -D DGST_R3=7"
118                                  " -D DGST_ELEM=16";
119     err = clBuildProgram(program, 1, &device, build_options.c_str(), nullptr, nullptr);
```

```
120     if(err != CL_SUCCESS) {
121         cerr << "[ERROR] Kernel build failed.\n";
122         size_t log_size = 0;
123         clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, nullptr,
            ↪  &log_size);
124         vector<char> log(log_size);
125         clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size,
            ↪  log.data(), nullptr);
126         cerr << "Build log:\n" << log.data() << "\n";
127 
128         clReleaseProgram(program);
129         clReleaseMemObject(hashes_buf);
130         clReleaseMemObject(pwd_salt_buf);
131         clReleaseCommandQueue(queue);
132         clReleaseContext(context);
133         return 1;
134     }
135 
136     //cout << "[INFO] Kernel compiled successfully. Preparing for execution...\n";
137     cl_kernel kernel = clCreateKernel(program, "psha512_it_sha", &err);
138 
139     if(err != CL_SUCCESS) {
140         cerr << "[ERROR] clCreateKernel failed: " << err << "\n";
141         clReleaseProgram(program);
142         clReleaseMemObject(hashes_buf);
143         clReleaseMemObject(pwd_salt_buf);
144         clReleaseCommandQueue(queue);
145         clReleaseContext(context);
146         return 1;
147     }
148 
149     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &pwd_salt_buf);
150     err |= clSetKernelArg(kernel, 1, sizeof(cl_uint), &tot_len);
151     err |= clSetKernelArg(kernel, 2, sizeof(cl_uint), &n_rounds);
152     err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &hashes_buf);
153 
154     if(err != CL_SUCCESS) {
155         cerr << "[ERROR] clSetKernelArg failed: " << err << "\n";
156         clReleaseKernel(kernel);
157         clReleaseProgram(program);
158         clReleaseMemObject(hashes_buf);
159         clReleaseMemObject(pwd_salt_buf);
160         clReleaseCommandQueue(queue);
161         clReleaseContext(context);
162         return 1;
163     }
164 
165     //cout << "[INFO] Launching kernel...\n";
166 
```

```
    auto t0 = chrono::high_resolution_clock::now();
    err = clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &n_cores, nullptr, 0,
     ↪  nullptr, nullptr);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clEnqueueNDRangeKernel failed: " << err << "\n";
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseMemObject(hashes_buf);
        clReleaseMemObject(pwd_salt_buf);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        return 1;
    }

    clFinish(queue);

    auto t1 = chrono::high_resolution_clock::now();

    double elapsed_ns = chrono::duration<double>(t1 - t0).count();
    cout << "[TIME] GPU phase: " << fixed << setprecision(10) << elapsed_ns << " s\n";


    //cout << "[INFO] Reading back results from device...\n";
    err = clEnqueueReadBuffer(queue, hashes_buf, CL_TRUE, 0,
                              SHA512_DIGEST_LENGTH * n_cores, hashes.data(), 0, nullptr,
                               ↪  nullptr);

    if(err != CL_SUCCESS) {
        cerr << "[ERROR] clEnqueueReadBuffer failed: " << err << "\n";
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseMemObject(hashes_buf);
        clReleaseMemObject(pwd_salt_buf);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        return 1;
    }

    //cout << "[INFO] Cleaning up OpenCL resources...\n";
    clReleaseMemObject(pwd_salt_buf);
    clReleaseMemObject(hashes_buf);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    size_t active_hashes = n_cores;
    SHA512_CTX ctx;
```

```
    while(active_hashes > 1) {
        for(size_t i = 0; i < active_hashes / 2; ++i) {
            SHA512_Init(&ctx);
            SHA512_Update(&ctx, hashes[2 * i].data(), SHA512_DIGEST_LENGTH);
            SHA512_Update(&ctx, hashes[2 * i + 1].data(), SHA512_DIGEST_LENGTH);
            SHA512_Final(hashes[i].data(), &ctx);
        }

        if(active_hashes % 2 != 0) {
            hashes[active_hashes / 2] = hashes[active_hashes - 1];
            active_hashes = active_hashes / 2 + 1;
        } else {
            active_hashes = active_hashes / 2;
        }
    }

    //cout << "$psha512_it$n_subtrees=" << n_cores << "$rounds=" << n_rounds << "$" <<
    ↪  salt << "$";
    cout << hex << setfill('0');
    for(size_t i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
        cout << setw(2) << static_cast<int>(hashes[0][i]);
    }
    cout << endl;
    return 0;
}
```

Listing A.28: `parallel_sha512_it` C++ main implementation.

A.8 SHA-512 constants

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

Table A.1: SHA-512 round constants

A.9 Hashcat module 92000 C implementation

```
#include "common.h"
#include "types.h"
#include "modules.h"
#include "bitops.h"
#include "convert.h"
#include "shared.h"

static const u32   ATTACK_EXEC    = ATTACK_EXEC_INSIDE_KERNEL;
static const u32   DGST_POS0      = 14;
static const u32   DGST_POS1      = 15;
static const u32   DGST_POS2      = 6;
static const u32   DGST_POS3      = 7;
static const u32   DGST_SIZE      = DGST_SIZE_8_8;
static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH_SALTED;
static const char *HASH_NAME      = "psha512";
static const u64   KERN_TYPE      = 92000;
static const u32   OPTI_TYPE      = OPTI_TYPE_USES_BITS_64
                                  | OPTI_TYPE_RAW_HASH;
static const u64   OPTS_TYPE      = OPTS_TYPE_STOCK_MODULE
```

```
                                    | OPTS_TYPE_PT_GENERATE_BE;
static const u32   SALT_TYPE      = SALT_TYPE_GENERIC;
static const char *ST_PASS        = "hashcat";
static const char *ST_HASH        =
 ↪  "$psha512$n_subtrees=32$n_leaves=256$12345678$3a0a4d35c6629bfded
6dc320fbc92425b9c2aa7f13868c92e1629ca60eada672ed034c49a
04981aef853ee5fa44b8a00ca5dfa79bfe0d7add08a0f1dfba496a3";

#define MAX_SUBTREES 128
#define MAX_LEAVES_PER_SUBTREE 1024

typedef struct psha512{
        u32 n_subtrees;
        u32 n_leaves;
} psha512_t;

u64         module_esalt_size     (MAYBE_UNUSED const hashconfig_t *hashconfig,
 ↪  MAYBE_UNUSED const user_options_t *user_options, MAYBE_UNUSED const
 ↪  user_options_extra_t *user_options_extra) { return sizeof (psha512_t); }
// All other predefined functions are identical to those used in Hashcat module 1700.

char *module_jit_build_options(MAYBE_UNUSED const hashconfig_t *hashconfig,
        MAYBE_UNUSED const user_options_t *user_options,
        MAYBE_UNUSED const user_options_extra_t *user_options_extra,
        MAYBE_UNUSED const hashes_t *hashes,
        MAYBE_UNUSED const hc_device_param_t *device_param) {

        char *jit_build_options = NULL;
        // Extra treatment for Apple systems
        if(device_param->opencl_platform_vendor_id == VENDOR_ID_APPLE) {
                // Metal
                if(device_param->is_metal == true) {
                        hc_asprintf(&jit_build_options, "-D _unroll");
                }
                return jit_build_options;
        }
        // HIP
        if(device_param->opencl_device_vendor_id == VENDOR_ID_AMD_USE_HIP) {
                hc_asprintf(&jit_build_options, "-D _unroll");
        }
        // ROCM
        if((device_param->opencl_device_vendor_id == VENDOR_ID_AMD) &&
         ↪  (device_param->has_vperm == true)) {
                hc_asprintf(&jit_build_options, "-D _unroll");
        }
        return jit_build_options;
}

int module_hash_decode(MAYBE_UNUSED const hashconfig_t *hashconfig,
```

```
65          MAYBE_UNUSED void *digest_buf,
66          MAYBE_UNUSED salt_t *salt,
67          MAYBE_UNUSED void *esalt_buf,
68          MAYBE_UNUSED void *hook_salt_buf,
69          MAYBE_UNUSED hashinfo_t *hash_info,
70          const char *line_buf,
71          MAYBE_UNUSED const int line_len) {
72
73          u64 *digest = (u64 *)digest_buf;
74          hc_token_t token;
75          memset (&token, 0, sizeof(hc_token_t));
76
77          token.token_cnt  = 5;
78          token.signatures_cnt    = 1;
79          token.signatures_buf[0] = "$psha512$";
80          token.len[0]     = 9;         // strlen("$psha512$")
81          token.attr[0]    = TOKEN_ATTR_FIXED_LENGTH
82                                       | TOKEN_ATTR_VERIFY_SIGNATURE;
83          token.sep[1]     = '$';
84          token.len_min[1] = 12;        // "n_subtrees=1"
85          token.len_max[1] = 20;
86          token.attr[1]    = TOKEN_ATTR_VERIFY_LENGTH;
87          token.sep[2]     = '$';
88          token.len_min[2] = 8;         // "n_leaves=1"
89          token.len_max[2] = 20;
90          token.attr[2]    = TOKEN_ATTR_VERIFY_LENGTH;
91          token.sep[3]     = '$';
92          token.len_min[3] = SALT_MIN;
93          token.len_max[3] = SALT_MAX;
94          token.attr[3]    = TOKEN_ATTR_VERIFY_LENGTH;
95          token.len[4]     = 128;
96          token.attr[4]    = TOKEN_ATTR_FIXED_LENGTH
97                                       | TOKEN_ATTR_VERIFY_HEX;
98
99          const int rc_tokenizer = input_tokenizer((const u8 *) line_buf, line_len,
            ↪  &token);
100         if(rc_tokenizer != PARSER_OK) return (rc_tokenizer);
101         psha512_t *psha512 = (psha512_t *)esalt_buf;
102         const char *subtree_prefix = "n_subtrees=";
103         const char *leaves_prefix  = "n_leaves=";
104
105         if(memcmp(token.buf[1], subtree_prefix, strlen(subtree_prefix)) != 0) return
            ↪  PARSER_SIGNATURE_UNMATCHED;
106         if(memcmp(token.buf[2], leaves_prefix,  strlen(leaves_prefix))  != 0) return
            ↪  PARSER_SIGNATURE_UNMATCHED;
107
108         char *endptr = NULL;
109         psha512->n_subtrees = hc_strtoul((const char *)token.buf[1] +
            ↪  strlen(subtree_prefix), &endptr, 10);
```

```
110         if(endptr != (const char *) token.buf[1] + token.len[1]) return
            ↪ PARSER_SALT_VALUE;
111
112         psha512->n_leaves = hc_strtoul((const char *)token.buf[2] +
            ↪ strlen(leaves_prefix), &endptr, 10);
113         if(endptr != (const char *) token.buf[2] + token.len[2]) return
            ↪ PARSER_SALT_VALUE;
114
115         if(psha512->n_subtrees < 2) return PARSER_SALT_VALUE;
116         if(psha512->n_subtrees > MAX_SUBTREES) return PARSER_SALT_VALUE;
117         if(psha512->n_leaves < 2) return PARSER_SALT_VALUE;
118         if(psha512->n_leaves > MAX_LEAVES_PER_SUBTREE) return PARSER_SALT_VALUE;
119
120         const u8 *hash_pos = token.buf[4];
121         digest[0] = hex_to_u64(hash_pos +   0);
122         digest[1] = hex_to_u64(hash_pos +  16);
123         digest[2] = hex_to_u64(hash_pos +  32);
124         digest[3] = hex_to_u64(hash_pos +  48);
125         digest[4] = hex_to_u64(hash_pos +  64);
126         digest[5] = hex_to_u64(hash_pos +  80);
127         digest[6] = hex_to_u64(hash_pos +  96);
128         digest[7] = hex_to_u64(hash_pos + 112);
129         digest[0] = byte_swap_64(digest[0]);
130         digest[1] = byte_swap_64(digest[1]);
131         digest[2] = byte_swap_64(digest[2]);
132         digest[3] = byte_swap_64(digest[3]);
133         digest[4] = byte_swap_64(digest[4]);
134         digest[5] = byte_swap_64(digest[5]);
135         digest[6] = byte_swap_64(digest[6]);
136         digest[7] = byte_swap_64(digest[7]);
137
138         const u8 *salt_pos = token.buf[3];
139         const int salt_len = token.len[3];
140         const bool parse_rc = generic_salt_decode(hashconfig, salt_pos, salt_len, (u8
            ↪ *)salt->salt_buf, (int *)&salt->salt_len);
141         if(parse_rc == false) return (PARSER_SALT_LENGTH);
142         return (PARSER_OK);
143 }
144
145 int module_hash_encode(MAYBE_UNUSED const hashconfig_t *hashconfig,
146         MAYBE_UNUSED const void *digest_buf,
147         MAYBE_UNUSED const salt_t *salt,
148         MAYBE_UNUSED const void *esalt_buf,
149         MAYBE_UNUSED const void *hook_salt_buf,
150         MAYBE_UNUSED const hashinfo_t *hash_info,
151         char *line_buf,
152         MAYBE_UNUSED const int line_size) {
153
154         const u64 *digest = (const u64 *)digest_buf;
```

```
155         const psha512_t *psha512 = (const psha512_t *)esalt_buf;
156         u64 tmp[8];
157         tmp[0] = digest[0];
158         tmp[1] = digest[1];
159         tmp[2] = digest[2];
160         tmp[3] = digest[3];
161         tmp[4] = digest[4];
162         tmp[5] = digest[5];
163         tmp[6] = digest[6];
164         tmp[7] = digest[7];
165
166         if(hashconfig->opti_type & OPTI_TYPE_OPTIMIZED_KERNEL) {
167                 tmp[0] += SHA512M_A;
168                 tmp[1] += SHA512M_B;
169                 tmp[2] += SHA512M_C;
170                 tmp[3] += SHA512M_D;
171                 tmp[4] += SHA512M_E;
172                 tmp[5] += SHA512M_F;
173                 tmp[6] += SHA512M_G;
174                 tmp[7] += SHA512M_H;
175         }
176
177         tmp[0] = byte_swap_64(tmp[0]);
178         tmp[1] = byte_swap_64(tmp[1]);
179         tmp[2] = byte_swap_64(tmp[2]);
180         tmp[3] = byte_swap_64(tmp[3]);
181         tmp[4] = byte_swap_64(tmp[4]);
182         tmp[5] = byte_swap_64(tmp[5]);
183         tmp[6] = byte_swap_64(tmp[6]);
184         tmp[7] = byte_swap_64(tmp[7]);
185
186         u8 hash_buf[129] = { 0 };
187         int hash_len = 0;
188         u64_to_hex(tmp[0], hash_buf + hash_len); hash_len += 16;
189         u64_to_hex(tmp[1], hash_buf + hash_len); hash_len += 16;
190         u64_to_hex(tmp[2], hash_buf + hash_len); hash_len += 16;
191         u64_to_hex(tmp[3], hash_buf + hash_len); hash_len += 16;
192         u64_to_hex(tmp[4], hash_buf + hash_len); hash_len += 16;
193         u64_to_hex(tmp[5], hash_buf + hash_len); hash_len += 16;
194         u64_to_hex(tmp[6], hash_buf + hash_len); hash_len += 16;
195         u64_to_hex(tmp[7], hash_buf + hash_len); hash_len += 16;
196
197         hash_buf[hash_len] = 0;
198         u8 salt_buf[SALT_MAX * 2 + 1] = { 0 };
199         const int salt_len = generic_salt_encode(hashconfig, (const u8 *)salt->salt_buf,
          ↪  (const int)salt->salt_len, salt_buf);
200         salt_buf[salt_len] = 0;
201         const int out_len = snprintf(
202                 line_buf,
```

```
                line_size,
                "$psha512$n_subtrees=%u$n_leaves=%u$%s$%s",
                psha512->n_subtrees,
                psha512->n_leaves,
                salt_buf,
                hash_buf
        );
        return out_len;
}

void module_init(module_ctx_t *module_ctx) {
        module_ctx->module_context_size             = MODULE_CONTEXT_SIZE_CURRENT;
        module_ctx->module_interface_version        = MODULE_INTERFACE_VERSION_CURRENT;
        ...
        // All other assignments are identical to those used in Hashcat module 1700.
        module_ctx->module_esalt_size               = module_esalt_size;
        ...
}
```

Listing A.29: Hashcat module 92000 implementation.

A.10 Hashcat module 92001 C implementation

```
// same includes as module 92000

static const u32   ATTACK_EXEC    = ATTACK_EXEC_INSIDE_KERNEL;
static const u32   DGST_POS0      = 14;
static const u32   DGST_POS1      = 15;
static const u32   DGST_POS2      = 6;
static const u32   DGST_POS3      = 7;
static const u32   DGST_SIZE      = DGST_SIZE_8_8;
static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH_SALTED;
static const char *HASH_NAME      = "psha512_it";
static const u64   KERN_TYPE      = 92001;
static const u32   OPTI_TYPE      = OPTI_TYPE_USES_BITS_64
                                  | OPTI_TYPE_RAW_HASH;
static const u64   OPTS_TYPE      = OPTS_TYPE_STOCK_MODULE
                                  | OPTS_TYPE_PT_GENERATE_BE;
static const u32   SALT_TYPE      = SALT_TYPE_GENERIC;
static const char *ST_PASS        = "hashcat";
static const char *ST_HASH        =
 ↪  "$psha512_it$n_subtrees=4$rounds=16383$12345678$6f12b22ab7c7916bd8a3
881bdebf61ad9ecd641938698d1576cd2e7c6e103e9199bac9189
d666e7d7df972221d2fee911cbb787205c2610e09777886f1237fe0";

typedef struct psha512_it{
        u32 n_subtrees;
        u32 rounds;
```

```
} psha512_it_t;

// same functions as declared in module 92000

char *module_jit_build_options(MAYBE_UNUSED const hashconfig_t *hashconfig,
        // same implementation as module 92000
}

int module_hash_decode(MAYBE_UNUSED const hashconfig_t *hashconfig,
        MAYBE_UNUSED void *digest_buf,
        MAYBE_UNUSED salt_t *salt,
        MAYBE_UNUSED void *esalt_buf,
        MAYBE_UNUSED void *hook_salt_buf,
        MAYBE_UNUSED hashinfo_t *hash_info,
        const char *line_buf,
        MAYBE_UNUSED const int line_len) {

        u64 *digest = (u64 *)digest_buf;
        hc_token_t token;
        memset (&token, 0, sizeof(hc_token_t));
        token.token_cnt  = 5;
        token.signatures_cnt    = 1;
        token.signatures_buf[0] = "$psha512_it$";
        token.len[0]     = 12;        // strlen("$psha512_it$")
        token.attr[0]    = TOKEN_ATTR_FIXED_LENGTH
                                    | TOKEN_ATTR_VERIFY_SIGNATURE;
        token.sep[1]     = '$';
        token.len_min[1] = 12;        // "n_subtrees=1"
        token.len_max[1] = 20;
        token.attr[1]    = TOKEN_ATTR_VERIFY_LENGTH;
        token.sep[2]     = '$';
        token.len_min[2] = 8;         // "rounds=1"
        token.len_max[2] = 20;
        token.attr[2]    = TOKEN_ATTR_VERIFY_LENGTH;
        token.sep[3]     = '$';
        token.len_min[3] = SALT_MIN;
        token.len_max[3] = SALT_MAX;
        token.attr[3]    = TOKEN_ATTR_VERIFY_LENGTH;
        token.len[4]     = 128;
        token.attr[4]    = TOKEN_ATTR_FIXED_LENGTH
                                    | TOKEN_ATTR_VERIFY_HEX;

        const int rc_tokenizer = input_tokenizer((const u8 *) line_buf, line_len,
         ↪  &token);
        if(rc_tokenizer != PARSER_OK) return (rc_tokenizer);
        psha512_it_t *psha512_it = (psha512_it_t *)esalt_buf;
        const char *subtree_prefix = "n_subtrees=";
        const char *rounds_prefix  = "rounds=";

```

```
73          if(memcmp(token.buf[1], subtree_prefix, strlen(subtree_prefix)) != 0) return
            ↪  PARSER_SIGNATURE_UNMATCHED;
74          if(memcmp(token.buf[2], rounds_prefix,  strlen(rounds_prefix))  != 0) return
            ↪  PARSER_SIGNATURE_UNMATCHED;
75          char *endptr = NULL;
76          psha512_it->n_subtrees = hc_strtoul((const char *)token.buf[1] +
            ↪  strlen(subtree_prefix), &endptr, 10);
77          if(endptr != (const char *) token.buf[1] + token.len[1]) return
            ↪  PARSER_SALT_VALUE;
78          psha512_it->rounds = hc_strtoul((const char *)token.buf[2] +
            ↪  strlen(rounds_prefix), &endptr, 10);
79          if(endptr != (const char *) token.buf[2] + token.len[2]) return
            ↪  PARSER_SALT_VALUE;
80          if(psha512_it->n_subtrees < 2) return PARSER_SALT_VALUE;
81          if(psha512_it->n_subtrees > 32) return PARSER_SALT_VALUE;
82          if(psha512_it->rounds < 1) return PARSER_SALT_VALUE;
83
84          // same implementation as module 92000
85  }
86
87  int module_hash_encode(MAYBE_UNUSED const hashconfig_t *hashconfig,
88          MAYBE_UNUSED const void *digest_buf,
89          MAYBE_UNUSED const salt_t *salt,
90          MAYBE_UNUSED const void *esalt_buf,
91          MAYBE_UNUSED const void *hook_salt_buf,
92          MAYBE_UNUSED const hashinfo_t *hash_info,
93          char *line_buf,
94          MAYBE_UNUSED const int line_size) {
95
96          const u64 *digest = (const u64 *)digest_buf;
97          const psha512_it_t *psha512_it = (const psha512_it_t *)esalt_buf;
98
99      // same implementation as module 92000
100
101         const int out_len = snprintf(
102                 line_buf,
103                 line_size,
104                 "$psha512_it$n_subtrees=%u$rounds=%u$%s$%s",
105                 psha512_it->n_subtrees,
106                 psha512_it->rounds,
107                 salt_buf,
108                 hash_buf
109         );
110
111         return out_len;
112 }
113
114 void module_init(module_ctx_t *module_ctx) {
115         // same implementation as module 92000
```

```
}
```

Listing A.30: Hashcat module 92001 implementation.

A.11 Hashcat mode 92000 OpenCL kernel code

```
#ifdef KERNEL_STATIC
#include M2S(INCLUDE_PATH/inc_vendor.h)
#include M2S(INCLUDE_PATH/inc_types.h)
#include M2S(INCLUDE_PATH/inc_platform.cl)
#include M2S(INCLUDE_PATH/inc_common.cl)
#include M2S(INCLUDE_PATH/inc_rp.h)
#include M2S(INCLUDE_PATH/inc_rp.cl)
#include M2S(INCLUDE_PATH/inc_scalar.cl)
#include M2S(INCLUDE_PATH/inc_hash_sha512.cl)
#endif

#define SHA512_BLOCK_WORDS 32
#define SHA512_DIGEST_LENGTH 64
#define MAX_SUBTREES 128
#define MAX_LEAVES_PER_SUBTREE 1024

// Unrolled function to store the digest from the sha512 context structure to the
// result array (not the final output array)
#define STORE_DIGEST_BLOCK(dst, h)                          \
    dst[0]  = hc_swap32_S((u32) (h[0] >> 32));              \
    dst[1]  = hc_swap32_S((u32) (h[0] >>  0));              \
    dst[2]  = hc_swap32_S((u32) (h[1] >> 32));              \
    dst[3]  = hc_swap32_S((u32) (h[1] >>  0));              \
    dst[4]  = hc_swap32_S((u32) (h[2] >> 32));              \
    dst[5]  = hc_swap32_S((u32) (h[2] >>  0));              \
    dst[6]  = hc_swap32_S((u32) (h[3] >> 32));              \
    dst[7]  = hc_swap32_S((u32) (h[3] >>  0));              \
    dst[8]  = hc_swap32_S((u32) (h[4] >> 32));              \
    dst[9]  = hc_swap32_S((u32) (h[4] >>  0));              \
    dst[10] = hc_swap32_S((u32) (h[5] >> 32));              \
    dst[11] = hc_swap32_S((u32) (h[5] >>  0));              \
    dst[12] = hc_swap32_S((u32) (h[6] >> 32));              \
    dst[13] = hc_swap32_S((u32) (h[6] >>  0));              \
    dst[14] = hc_swap32_S((u32) (h[7] >> 32));              \
    dst[15] = hc_swap32_S((u32) (h[7] >>  0));              \
    dst[16] = 0;                                            \
    dst[17] = 0;                                            \
    dst[18] = 0;                                            \
    dst[19] = 0;                                            \
    dst[20] = 0;                                            \
    dst[21] = 0;                                            \
    dst[22] = 0;                                            \
```

```
    dst[23] = 0;                                    \
    dst[24] = 0;                                    \
    dst[25] = 0;                                    \
    dst[26] = 0;                                    \
    dst[27] = 0;                                    \
    dst[28] = 0;                                    \
    dst[29] = 0;                                    \
    dst[30] = 0;                                    \
    dst[31] = 0

// Unrolled function to copy a sha512 block (src) into another block (dst)
#define COPY_BLOCK(dst, src)                        \
    dst[0]  = src[0];                               \
    dst[1]  = src[1];                               \
    dst[2]  = src[2];                               \
    dst[3]  = src[3];                               \
    dst[4]  = src[4];                               \
    dst[5]  = src[5];                               \
    dst[6]  = src[6];                               \
    dst[7]  = src[7];                               \
    dst[8]  = src[8];                               \
    dst[9]  = src[9];                               \
    dst[10] = src[10];                              \
    dst[11] = src[11];                              \
    dst[12] = src[12];                              \
    dst[13] = src[13];                              \
    dst[14] = src[14];                              \
    dst[15] = src[15];                              \
    dst[16] = src[16];                              \
    dst[17] = src[17];                              \
    dst[18] = src[18];                              \
    dst[19] = src[19];                              \
    dst[20] = src[20];                              \
    dst[21] = src[21];                              \
    dst[22] = src[22];                              \
    dst[23] = src[23];                              \
    dst[24] = src[24];                              \
    dst[25] = src[25];                              \
    dst[26] = src[26];                              \
    dst[27] = src[27];                              \
    dst[28] = src[28];                              \
    dst[29] = src[29];                              \
    dst[30] = src[30];                              \
    dst[31] = src[31]

typedef struct psha512{
    u32 n_subtrees;
    u32 n_leaves;
} psha512_t;
```

```
92
93  #define PSHA512_BODY(compare_macro)                          \
94      const u64 gid = get_global_id(0);                        \
95      if(gid >= GID_CNT) return;          \
96      GLOBAL_AS const psha512_t *psha512 = (GLOBAL_AS const psha512_t *)esalt_bufs;      \
97      const u32 n_subtrees = psha512[DIGESTS_OFFSET_HOST].n_subtrees;       \
98      const u32 n_leaves   = psha512[DIGESTS_OFFSET_HOST].n_leaves;         \
99          \
100     COPY_PW(pws[gid]);          \
101     const u32 salt_len = salt_bufs[SALT_POS_HOST].salt_len;       \
102     u32 s[64] = { 0 };          \
103     for(u32 i = 0, idx = 0; i < salt_len; i += 4, idx += 1) {      \
104         s[idx] = hc_swap32_S(salt_bufs[SALT_POS_HOST].salt_buf[idx]);      \
105     }       \
106         \
107     for(u32 il_pos = 0; il_pos < IL_CNT; il_pos++) {         \
108         pw_t tmp = PASTE_PW;          \
109         tmp.pw_len = apply_rules(rules_buf[il_pos].cmds, tmp.i, tmp.pw_len);         \
110         sha512_ctx_t ctx;          \
111         u32 subtree_roots[MAX_SUBTREES][SHA512_BLOCK_WORDS];         \
112         u32 leaf_hashes[MAX_LEAVES_PER_SUBTREE][SHA512_BLOCK_WORDS];         \
113         \
114         for(u32 subtree = 0; subtree < n_subtrees; subtree++) {         \
115             u32 counter_block[SHA512_BLOCK_WORDS] = { 0 };         \
116             PRIVATE_AS u8 *counter_bytes = (PRIVATE_AS u8 *) counter_block;         \
117             const u32 start = subtree * n_leaves;         \
118             counter_bytes[3] = (u8) ((start >>  0) & 0xff);         \
119             counter_bytes[2] = (u8) ((start >>  8) & 0xff);         \
120             counter_bytes[1] = (u8) ((start >> 16) & 0xff);         \
121             counter_bytes[0] = (u8) ((start >> 24) & 0xff);         \
122         \
123             for(u32 leaf = 0; leaf < n_leaves; leaf++) {         \
124                 counter_bytes[3]++;         \
125                 if(counter_bytes[3] == 0) {         \
126                     counter_bytes[2]++;         \
127                     if(counter_bytes[2] == 0){         \
128                         counter_bytes[1]++;         \
129                         if(counter_bytes[1] == 0) {         \
130                             counter_bytes[0]++;         \
131                         }         \
132                     }         \
133                 }         \
134                 sha512_init(&ctx);         \
135                 sha512_update_swap(&ctx, tmp.i, tmp.pw_len);         \
136                 sha512_update(&ctx, s, salt_len);         \
137                 sha512_update_swap(&ctx, counter_block, 4);         \
138                 sha512_final(&ctx);         \
139                 STORE_DIGEST_BLOCK(leaf_hashes[leaf], ctx.h);         \
140             }         \
```

```
\
    u32 active_hashes = n_leaves;        \
    while(active_hashes > 2) {        \
        for(u32 i = 0; i < active_hashes / 2; i++) {        \
            sha512_init(&ctx);        \
            sha512_update_swap(&ctx, leaf_hashes[2 * i],
             ↪  SHA512_DIGEST_LENGTH);     \
            sha512_update_swap(&ctx, leaf_hashes[2 * i + 1],
             ↪  SHA512_DIGEST_LENGTH);     \
            sha512_final(&ctx);        \
            STORE_DIGEST_BLOCK(leaf_hashes[i], ctx.h);        \
        }       \
        if((active_hashes & 1) != 0) {        \
            COPY_BLOCK(leaf_hashes[active_hashes / 2], leaf_hashes[active_hashes
             ↪  - 1]); \
            active_hashes = active_hashes / 2 + 1;        \
        } else {        \
            active_hashes = active_hashes / 2;        \
        }       \
    }       \
    sha512_init(&ctx);       \
    sha512_update_swap(&ctx, leaf_hashes[0], SHA512_DIGEST_LENGTH);        \
    sha512_update_swap(&ctx, leaf_hashes[1], SHA512_DIGEST_LENGTH);        \
    sha512_final(&ctx);        \
    STORE_DIGEST_BLOCK(subtree_roots[subtree], ctx.h);        \
}       \
\
u32 active = n_subtrees;        \
while(active > 1) {        \
    for(u32 i = 0; i < active / 2; i++) {        \
        sha512_init(&ctx);        \
        sha512_update_swap(&ctx, subtree_roots[2 * i],     SHA512_DIGEST_LENGTH);
         ↪  \
        sha512_update_swap(&ctx, subtree_roots[2 * i + 1], SHA512_DIGEST_LENGTH);
         ↪  \
        sha512_final(&ctx);        \
        STORE_DIGEST_BLOCK(subtree_roots[i], ctx.h);        \
    }       \
    if((active & 1) != 0) {        \
        COPY_BLOCK(subtree_roots[active / 2], subtree_roots[active - 1]);
         ↪  \
        active = active / 2 + 1;        \
    } else {        \
        active = active / 2;        \
    }       \
}       \
\
const u32 r0 = l32_from_64_S(ctx.h[7]);        \
const u32 r1 = h32_from_64_S(ctx.h[7]);        \
```

```
        const u32 r2 = l32_from_64_S(ctx.h[3]);         \
        const u32 r3 = h32_from_64_S(ctx.h[3]);         \
        compare_macro(r0, r1, r2, r3);          \
    }

KERNEL_FQ KERNEL_FA void m92000_mxx(KERN_ATTR_RULES()) {
    PSHA512_BODY(COMPARE_M_SCALAR)
}

KERNEL_FQ KERNEL_FA void m92000_sxx(KERN_ATTR_RULES()) {
    const u32 search[4] = {
        digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R0],
        digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R1],
        digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R2],
        digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R3]
    };

    PSHA512_BODY(COMPARE_S_SCALAR)
}
```

Listing A.31: Hashcat mode 92000 OpenCL kernel implementation.

A.12 Hashcat mode 92001 OpenCL kernel code

```
#ifdef KERNEL_STATIC
#include M2S(INCLUDE_PATH/inc_vendor.h)
#include M2S(INCLUDE_PATH/inc_types.h)
#include M2S(INCLUDE_PATH/inc_platform.cl)
#include M2S(INCLUDE_PATH/inc_common.cl)
#include M2S(INCLUDE_PATH/inc_rp.h)
#include M2S(INCLUDE_PATH/inc_rp.cl)
#include M2S(INCLUDE_PATH/inc_scalar.cl)
#include M2S(INCLUDE_PATH/inc_hash_sha512.cl)
#endif

#define SHA512_BLOCK_WORDS 32
#define MAX_SUBTREES 32

typedef struct psha512_it{
        u32 n_subtrees;
        u32 rounds;
} psha512_it_t;

KERNEL_FQ KERNEL_FA void m92001_mxx(KERN_ATTR_RULES()) {
        const u64 lid = get_local_id(0);
        const u64 gid = get_global_id(0);
        GLOBAL_AS const psha512_it_t *psha512_it = (GLOBAL_AS const psha512_it_t
        ↪ *)esalt_bufs;
```

```
24          const u32 n_subtrees = psha512_it[DIGESTS_OFFSET_HOST].n_subtrees;
25          const u32 rounds     = psha512_it[DIGESTS_OFFSET_HOST].rounds;
26          if(gid >= GID_CNT) return;
27
28          /*  base    */
29          COPY_PW(pws[gid]);
30          const u32 salt_len = salt_bufs[SALT_POS_HOST].salt_len;
31          u32 s[64] = { 0 };
32          for(u32 i = 0, idx = 0; i < salt_len; i += 4, idx += 1) {
33                  s[idx] = hc_swap32_S(salt_bufs[SALT_POS_HOST].salt_buf[idx]);
34          }
35
36          /*  loop    */
37          for(u32 il_pos = 0; il_pos < IL_CNT; il_pos++) {
38                  pw_t tmp = PASTE_PW;
39                  tmp.pw_len = apply_rules(rules_buf[il_pos].cmds, tmp.i, tmp.pw_len);
40                  u32 results[MAX_SUBTREES][SHA512_BLOCK_WORDS];
41
42                  /*      Subtree hashes      */
43                  for(u32 subtree = 0; subtree < n_subtrees; subtree++) {
44                          const u32 leaf_id = subtree + 1;
45                          u32 leaf_block[SHA512_BLOCK_WORDS] = { 0 };
46                          PRIVATE_AS u8 *leaf_bytes = (PRIVATE_AS u8 *)leaf_block;
47                          leaf_bytes[0] = (u8) ((leaf_id >> 24) & 0xff);
48                          leaf_bytes[1] = (u8) ((leaf_id >> 16) & 0xff);
49                          leaf_bytes[2] = (u8) ((leaf_id >>  8) & 0xff);
50                          leaf_bytes[3] = (u8) ((leaf_id >>  0) & 0xff);
51
52                          sha512_ctx_t ctx;
53                          sha512_init(&ctx);
54                          sha512_update_swap(&ctx, tmp.i, tmp.pw_len);
55                          sha512_update(&ctx, s, salt_len);
56                          sha512_update_swap(&ctx, leaf_block, 4);
57                          sha512_final(&ctx);
58                          STORE_DIGEST_BLOCK(results[subtree], ctx.h);
59
60                          for(u32 r = 1; r < rounds; r++) {
61                                  sha512_init(&ctx);
62                                  sha512_update_swap(&ctx, results[subtree], 64);
63                                  sha512_final(&ctx);
64                                  STORE_DIGEST_BLOCK(results[subtree], ctx.h);
65                          }
66                  }
67
68                  /*      Merkle reduction        */
69                  sha512_ctx_t ctx;
70                  u32 active = n_subtrees;
71                  while(active > 1) {
72                          for(u32 i = 0; i < active / 2; i++) {
```

```
                                sha512_init (&ctx);
                                sha512_update_swap(&ctx, results[2 * i],     64);
                                sha512_update_swap(&ctx, results[2 * i + 1], 64);
                                sha512_final(&ctx);
                                STORE_DIGEST_BLOCK(results[i], ctx.h);
                        }
                        if((active & 1) != 0) {
                                COPY_BLOCK(results[active / 2], results[active - 1]);
                                active = active / 2 + 1;
                        } else {
                                active = active / 2;
                        }
                }

                const u32 r0 = l32_from_64_S(ctx.h[7]);
                const u32 r1 = h32_from_64_S(ctx.h[7]);
                const u32 r2 = l32_from_64_S(ctx.h[3]);
                const u32 r3 = h32_from_64_S(ctx.h[3]);
                COMPARE_M_SCALAR(r0, r1, r2, r3);
        }
}

KERNEL_FQ KERNEL_FA void m92001_sxx(KERN_ATTR_RULES()) {
        const u64 lid = get_local_id(0);
        const u64 gid = get_global_id(0);
        if(gid >= GID_CNT) return;

        /*  digest  */
        const u32 search[4] = {
                digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R0],
                digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R1],
                digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R2],
                digests_buf[DIGESTS_OFFSET_HOST].digest_buf[DGST_R3]
        };
        GLOBAL_AS const psha512_it_t *psha512_it = (GLOBAL_AS const psha512_it_t
         ↪  *)esalt_bufs;
        const u32 n_subtrees = psha512_it[DIGESTS_OFFSET_HOST].n_subtrees;
        const u32 rounds     = psha512_it[DIGESTS_OFFSET_HOST].rounds;

        /*  base    */
        COPY_PW(pws[gid]);
        const u32 salt_len = salt_bufs[SALT_POS_HOST].salt_len;
        u32 s[64] = { 0 };
        for(u32 i = 0, idx = 0; i < salt_len; i += 4, idx += 1) {
                s[idx] = hc_swap32_S(salt_bufs[SALT_POS_HOST].salt_buf[idx]);
        }

        /*  loop    */
        for(u32 il_pos = 0; il_pos < IL_CNT; il_pos++) {
```

```
121                 pw_t tmp = PASTE_PW;
122                 tmp.pw_len = apply_rules(rules_buf[il_pos].cmds, tmp.i, tmp.pw_len);
123                 u32 results[MAX_SUBTREES][SHA512_BLOCK_WORDS];
124
125                 /**     Subtree hashes      */
126                 for(u32 subtree = 0; subtree < n_subtrees; subtree++) {
127                         const u32 leaf_id = subtree + 1;
128                         u32 leaf_block[SHA512_BLOCK_WORDS] = { 0 };
129                         PRIVATE_AS u8 *leaf_bytes = (PRIVATE_AS u8 *)leaf_block;
130                         leaf_bytes[0] = (u8) ((leaf_id >> 24) & 0xff);
131                         leaf_bytes[1] = (u8) ((leaf_id >> 16) & 0xff);
132                         leaf_bytes[2] = (u8) ((leaf_id >>  8) & 0xff);
133                         leaf_bytes[3] = (u8) ((leaf_id >>  0) & 0xff);
134
135                         sha512_ctx_t ctx;
136                         sha512_init(&ctx);
137                         sha512_update_swap(&ctx, tmp.i, tmp.pw_len);
138                         sha512_update(&ctx, s, salt_len);
139                         sha512_update_swap(&ctx, leaf_block, 4);
140                         sha512_final(&ctx);
141                         STORE_DIGEST_BLOCK(results[subtree], ctx.h);
142
143                         for(u32 r = 1; r < rounds; r++) {
144                                 sha512_init(&ctx);
145                                 sha512_update_swap(&ctx, results[subtree], 64);
146                                 sha512_final(&ctx);
147                                 STORE_DIGEST_BLOCK(results[subtree], ctx.h);
148                         }
149                 }
150
151                 /**     Merkle reduction        */
152                 sha512_ctx_t ctx;
153                 u32 active = n_subtrees;
154                 while(active > 1) {
155                         for(u32 i = 0; i < active / 2; i++) {
156                                 sha512_init (&ctx);
157                                 sha512_update_swap(&ctx, results[2 * i],     64);
158                                 sha512_update_swap(&ctx, results[2 * i + 1], 64);
159                                 sha512_final(&ctx);
160                                 STORE_DIGEST_BLOCK(results[i], ctx.h);
161                         }
162                         if((active & 1) != 0) {
163                                 COPY_BLOCK(results[active / 2], results[active - 1]);
164                                 active = active / 2 + 1;
165                         } else {
166                                 active = active / 2;
167                         }
168                 }
169                 const u32 r0 = l32_from_64_S(ctx.h[7]);
```

```
170                 const u32 r1 = h32_from_64_S(ctx.h[7]);
171                 const u32 r2 = l32_from_64_S(ctx.h[3]);
172                 const u32 r3 = h32_from_64_S(ctx.h[3]);
173                 COMPARE_S_SCALAR(r0, r1, r2, r3);
174         }
175 }
```

Listing A.32: Hashcat mode 92001 OpenCL kernel implementation.

Bibliography

[1] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Dec. 16, 1996.

[2] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. FIPS PUB 180-4. U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[3] Christof Paar and Jan Pelzl. *Understanding Cryptography*. Springer, 2010.

[4] Ralph C. Merkle. "Secrecy, Authentication, and Public Key Systems". PhD thesis. Stanford University, 1979. URL: https://ralphmerkle.com/papers/Thesis1979.pdf.

[5] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3. URL: https://link.springer.com/chapter/10.1007/3-540-48184-2_32.

[6] Gabriele 'matrix' Gristina Jens 'atom' Steube. *Hashcat*. Version 7.1.2. URL: https://hashcat.net/hashcat/.

[7] Gabriele 'matrix' Gristina Jens 'atom' Steube. *Hashcat*. Version 7.1.2. URL: https://github.com/hashcat/hashcat.

[8] Ulrich Drepper. *The SHA-512 Crypt Algorithm*. 2007. URL: https://github.com/besser82/libxcrypt/blob/develop/lib/crypt-sha512.c.

[9] Oleksandr Kuznetsov et al. "Evaluating the Security of Merkle Trees: An Analysis of Data Falsification Probabilities". In: *Cryptography* 8.3 (2024), p. 33. DOI: 10.3390/cryptography8030033.

[10] Douglas R. Stinson and Maura B. Paterson. *Cryptography: Theory and Practise*. Chapman & Hall/CRC Press, July 24, 2018.

[11] Thorsten Kukuk, Zack Weinberg, and Björn Esser. *The C crypt implementation*. 2007. URL: https://github.com/besser82/libxcrypt/blob/develop/lib/crypt.c.

[12] Marco d'Itri. *mkpasswd*. Version 5.0.24. 2001-2008. URL: https://github.com/weppos/whois-debian/blob/master/src/mkpasswd.c.

[13] *OpenSSL: The Open Source Toolkit for SSL/TLS*. Version 3.6.1. The OpenSSL Project. 1998. URL: https://www.openssl.org.

[14] *The OpenSSL library*. Version 3.6.1. The OpenSSL Project. URL: https://github.com/openssl/openssl.

[15] *OpenCL: Open Computing Language*. Version 3.0.19. Khronos Group. 2008. URL: https://www.khronos.org/opencl/.

[16] *John the Ripper Password Cracker*. Openwall. URL: https://www.openwall.com/john/.

[17] Colin Percival and Simon Josefsson. *The scrypt Password-Based Key Derivation Function*. RFC 7914. Internet Engineering Task Force, 2016. DOI: 10.17487/RFC7914. URL: https://www.rfc-editor.org/rfc/rfc7914.

[18] Niels Provos and David Mazières. "A Future-Adaptable Password Scheme". In: *Proceedings of the 1999 USENIX Annual Technical Conference*. USENIX Association, 1999. URL: https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme.

[19] Alex Biryukov et al. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC 9106. Internet Research Task Force. DOI: 10.17487/RFC9106. URL: https://www.rfc-editor.org/rfc/rfc9106.

[20] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. Tech. rep. 8018. Internet Engineering Task Force, 2017. DOI: 10.17487/RFC8018. URL: https://doi.org/10.17487/RFC8018.

[21] Jack O'Connor et al. *BLAKE3: One Function, Fast Everywhere*. Real World Crypto Symposium. Accessed: 2026-06-24. 2020. URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf.

[22] John Kelsey, Shu-jen Chang, and Ray Perlner. *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash*. NIST Special Publication 800-185. National Institute of Standards and Technology, 2016. DOI: 10.6028/NIST.SP.800-185. URL: https://doi.org/10.6028/NIST.SP.800-185.