Dynamic-Sized Secure Hash Algorithm with Memory-Hard Capabilities for Enhanced Cryptographic Security

A Project Report Submitted in Partial Fulfilment of the Requirements for the

7th Semester B.Tech Project, 2024

Submitted by

Tolemy Kashyap, 2112130
Jugya Kamal Gogoi, 2112098
Kumar Ashish Ranjan, 2112144
Arnav Raj, 2112116

Under the guidance of

Dr. Ripon Patgiri
Assistant Professor
Department of Computer Science & Engineering
National Institute of Technology, Silchar

Department of Computer Science & Engineering
NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR
Assam

DECLARATION

"Dynamic-Sized Secure Hash Algorithm with Memory-Hard Capabilities for Enhanced Cryptographic Security"

We declare that the art on display is mostly comprised of our own ideas and work, expressed in our own words. Where other people's thoughts or words were used, we properly cited and noted them in the reference materials. We have followed all academic honesty and integrity principles.

Tolemy Kashyap, 2112130
Jugya Kamal Gogoi, 2112098
Kumar Ashish Ranjan, 2112144
Arnav Raj, 2112116

Department of Computer Science and Engineering
National Institute of Technology Silchar, Assam

ACKNOWLEDGEMENT

We would like to thank our supervisor, Dr. Ripon Patgiri, Assistant Professor, CSE, NIT Silchar, for his invaluable direction, encouragement, and assistance during this project. His helpful suggestions for this entire effort and cooperation are gratefully thanked, as they enabled us to conduct extensive investigation and learn about many new subjects.

Our sincere gratitude to Dr. Ripon Patgiri for his unwavering support and patience, without which this project would not have been completed properly and on schedule. This project would not have been feasible without him, from answering any doubts we had to helping us with ideas whenever we were stuck.

Tolemy Kashyap, 2112130
Jugya Kamal Gogoi, 2112098
Kumar Ashish Ranjan, 2112144
Arnav Raj, 2112116

Department of Computer Science and Engineering
National Institute of Technology Silchar, Assam

Certificate of Supervision

It is certified that the work contained in this thesis entitled

"Dynamic-Sized Secure Hash Algorithm with Memory-Hard Capabilities for Enhanced Cryptographic Security"

submitted by

Tolemy Kashyap, 2112130
Jugya Kamal Gogoi, 2112098
Kumar Ashish Ranjan, 2112144
Arnav Raj, 2112116

for the B.Tech. 7th Semester Project December, 2024 is absolutely based on their own work carried out under my supervision.

Dr. Ripon Patgiri
Department of Computer Science and Engineering
National Institute of Technology Silchar, Assam

ABSTRACT

We introduce a randomized, variable-sized secure hash algorithm. We present three distinct variants of this algorithm, each capable of generating randomized secure hash values. The core mechanism of our approach involves generating a pool of pseudo-random bits using foundational hash functions and selecting a subset of these bits to construct the final randomized hash value. Each output of the underlying hash function contributes a single bit (either 0 or 1) to this pool. By randomizing the bit string produced, we ensure the generation of a secure, randomized hash value.

A distinctive feature of the algorithm is its ability to accept variable output sizes for the final hash value, allowing flexibility in generating hashes of different lengths based on the application's requirements. This adaptability enhances its utility across diverse cryptographic scenarios. Additionally, the algorithm incorporates a memory-hardness feature, which limits parallel processing capabilities, ensuring that legitimate users require minimal memory while attackers face significantly higher memory demands. Furthermore, we illustrate how the algorithm effectively mitigates the threat of Rainbow Table as a Service (RTaaS) attacks.

Contents

Chapter 1

Introduction

Secure hash algorithms (SHAs) are foundational components in cryptography, integral to applications such as digital signatures, message integrity checks, and data authentication. Among these, Keccak, the algorithm behind the SHA-3 standard selected by NIST, introduced the sponge construction, incorporating absorb and squeeze phases to process input data. This innovation enhanced secure hashing, offering improved resistance to certain cryptanalytic attacks. However, while Keccak aims to randomize output bits, it lacks a formal guarantee of true randomness. Moreover, it is inherently designed to generate fixed-sized outputs, making it unsuitable for scenarios requiring variable-sized hashes or for use as a random number generator.

Conventional SHAs face significant limitations due to their fixed-size output, which can be predictable and prone to attacks like rainbow tables and brute-force attacks. Fixed-length outputs also fail to meet the evolving needs of modern cryptographic applications, which increasingly demand adaptable and flexible hashing mechanisms. Additionally, traditional hash functions often do not incorporate adequate defenses against parallelization, leaving them vulnerable to attacks that exploit computational power, such as distributed brute-force or cryptanalytic methods.

Chapter 2

Literature Survey

2.1 Introduction

Secure hash functions are foundational to cryptographic systems, offering a means of ensuring data integrity, authentication, and secure communications. Over the years, various advancements in the field have aimed to address challenges such as brute-force attacks, memory-hardness, and the need for variable-sized outputs. This section delves into the research contributions that have shaped the development of dynamic-sized secure hash functions with memory-hard capabilities.

2.2 Literature Surveys

2.2.1 Stronger Key Derivation via Sequential Memory-Hard Function (Colin Percival, 2009)

- The paper introduces the concept of sequential memory-hard functions, focusing on their resistance to brute-force attacks through significant memory resource requirements.
- It presents the `scrypt` key derivation function, a robust alternative to PBKDF2 and bcrypt, emphasizing its adaptability with tunable parameters for memory and CPU cost.
- The proposed approach limits hardware attack advantages, making it a reliable choice for securing cryptographic applications.
- Despite its strengths, the method can be computationally intensive for lightweight devices, posing challenges for real-time applications.

2.2.2 Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications (2016)

- This work introduces Argon2, a versatile memory-hard hash function, designed to mitigate attacks leveraging GPUs and ASICs.
- Two variants are proposed: Argon2d for cryptocurrency applications and Argon2i for password hashing, with a focus on resistance to side-channel attacks.

- The function demonstrates strong scalability and security, making it suitable for diverse use cases from server-side authentication to blockchain technology.
- However, its performance can be influenced by memory configurations, necessitating careful tuning for optimal results.

2.2.3 Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks (2016)

- This research presents the Balloon hashing algorithm, designed to offer robust resistance to sequential attacks using random sandwich graphs.
- It outperforms Argon2i and scrypt in specific scenarios, demonstrating strong theoretical guarantees of memory-hardness.
- The algorithm's application is versatile, extending from secure password storage to cryptographic key derivation.
- A limitation lies in its suboptimal performance against parallel attacks, which may require further refinement.

2.2.4 Towards Practical Attacks on Argon2i and Balloon Hashing (Joel Alwen and Jeremiah Blocki, 2017)

- The paper explores practical heuristics to extend attacks on memory-hard algorithms like Argon2i and Balloon Hashing.
- It conducts simulations under various memory configurations, demonstrating vulnerabilities in settings with insufficient memory passes.
- Findings suggest that Argon2i-B requires more than ten memory passes to ensure robustness against proposed attack models.
- While the research highlights weaknesses in current configurations, it also offers valuable insights for improving memory-hardness.
- Limitations include challenges in scaling the attacks to scenarios with higher memory parameters, leaving certain cases unexplored.

2.2.5 Cryptographic Hash Functions: Recent Design Trends and Security Notions (Saif Al-Kuwari et al., 2011)

- This work provides an extensive overview of cryptographic hash functions and their evolution in response to cryptanalytic advances.
- Key properties like collision resistance, pre-image resistance, and second pre-image resistance are thoroughly examined.
- The research emphasizes the importance of emerging designs such as sponge constructions and tree-based methods.

- Limitations of traditional approaches, including Merkle-Damgård constructions, are addressed with proposals for advanced frameworks.
- A key takeaway is the role of competitions like SHA-3 in driving innovation and standardization in cryptographic designs.

2.2.6 On the Indifferentiability of the Sponge Construction (Guido Bertoni et al., 2008)

- This foundational study formalizes the sponge construction, proving its indifferentiability from a random oracle under specific conditions.
- Two simulators are introduced for analyzing transformations and permutations, ensuring secure outputs for variable-length hash values.
- The simplicity of the sponge model, avoiding complex compression functions, is a notable advantage.
- Applications include hash functions, message authentication codes (MACs), and stream ciphers, demonstrating its versatility.
- While the framework excels in flexibility, ensuring collision resistance for high-capacity outputs remains a design focus.

2.2.7 Keccak: Advances in Cryptology (Guido Bertoni et al., 2013)

- Keccak, the winner of the SHA-3 competition, introduced a paradigm shift with its permutation-based design.
- Utilizing the sponge construction, Keccak enables efficient and secure hashing with variable-length outputs.
- Features like tunable widths (25 to 1600 bits) and multi-security levels cater to lightweight and high-security applications alike.
- Its broad applicability spans hash functions, MACs, stream ciphers, and authenticated encryption.
- Challenges include balancing security and performance, especially for resource-constrained environments.

2.2.8 The Making of Keccak (Guido Bertoni et al., 2014)

- This paper elaborates on the development of Keccak, focusing on its innovative use of sponge constructions.
- Detailed descriptions of non-linear, mixing, and dispersion operations showcase its robust design.
- Proven resistance to collision and side-channel attacks underlines its reliability.

- It emphasizes energy efficiency and high performance, making it suitable for hardware and software implementations.
- The focus on design transparency and scrutiny during the SHA-3 competition strengthened its adoption as a standard.

2.3 Literature Gap

Despite significant advancements in cryptographic hash functions, existing methods like `scrypt`, Argon2, and Balloon Hashing exhibit limitations in addressing evolving computational threats. Many algorithms are constrained by their inability to effectively handle variable-sized outputs or seamlessly integrate with random number generators, which limits their applicability in dynamic cryptographic systems. While solutions like the sponge construction enable flexible hashing frameworks, ensuring true randomness across all output sizes remains a challenge. Additionally, algorithms such as Argon2 and Balloon Hashing, although robust against sequential attacks, show vulnerabilities in scenarios involving parallel computational resources, necessitating further refinements in memory-hard designs.

Another pressing gap lies in balancing security and performance. Existing methods often require significant computational resources, making them less feasible for lightweight or resource-constrained devices. For instance, while Keccak and Argon2 provide high security levels, they may not achieve optimal efficiency in real-time applications or on constrained hardware. Furthermore, advancements in cryptanalysis continue to expose potential weaknesses in established hash designs, emphasizing the need for adaptive and forward-compatible mechanisms. Addressing these gaps will be crucial to designing cryptographic solutions capable of withstanding the demands of modern and future computing environments.

2.4 Conclusion

The reviewed literature highlights the evolution of memory-hard and secure hashing mechanisms, emphasizing their importance in cryptographic security. Despite significant progress, challenges persist, particularly in balancing scalability, resistance to parallel attacks, and adaptability to modern computational environments. Addressing these gaps will pave the way for the development of robust, future-ready cryptographic solutions.

Chapter 3

Proposed Methodology

3.1 Overview

The goal is to develop a randomized, variable-sized secure hash algorithm. The proposed approach addresses challenges faced by secure hash algorithms from various attacks. The conventional secure hash algorithm has a fixed-sized hash value which becomes easy to attack by adversaries. It becomes difficult for adversaries when the hash value size is variable, and the adversary does not have any clue about the size. Therefore, rainbow table attacks or other similar kinds of attacks become computationally infeasible when the hash value size is variable.

3.2 Important Notations

Notation	Description
$PH()$	Primary hash function
h_v	Hash value generated by $PH()$
β	Bit size of hash value h_v
ω	Input string of arbitrary size
S	Seed value ≥ 32 bits
ζ	Final secure hash value
η	Bit size of the final hash value
$SV()$	Function to get the seed value

Table 3.1: Important Notations used in the report

3.3 Proposed Methodology

3.3.1 Introduction

We introduce a variant of the secure hash algorithm which is randomized and variable-sized. This algorithm is built on a random number generator to produce randomized hash values, and it can also function as a random number generator if needed. It generates random bits based on an input string ω and also accepts a seed value S as a second input. Initially, the seed value can be public or private, depending on the design requirements, but it is later transformed into a private value during the hashing process. Additionally, the algorithm takes a target bit size η for the output ζ, where $\zeta = \{0,1\}^\eta$ represents the final randomized secure hash value.

To achieve this, the algorithm relies on a core hash function, denoted as $PH()$. Alternatively, the algorithm can be customized by introducing a new hash function based on the properties of the primary hash functions. The primary hash function generates a hash value $h_v = \{0,1\}^\beta$ of β-bits, and for each iteration, DSSHA selects one bit from h_v to form the randomized secure hash output. The process continues iteratively to generate the full randomized hash value. Importantly, the primary hash function can be set to produce hash outputs of at least 32 bits (i.e., $\beta \geq 32$-bit). This ensures flexibility in both the size and security level of the generated hash.

3.3.2 Variants

Our algorithm is divided into three categories i.e. 1D, 2D and 3D variants. The 1D variant produces secure hash values faster than the others, but it is not a fully randomized one. It prefers performance over security. 2D and 3D variants are slower than 1D variant but are fully randomized and designed to be more secure.

3.3.3 One-dimensional randomized, variable-sized SHA

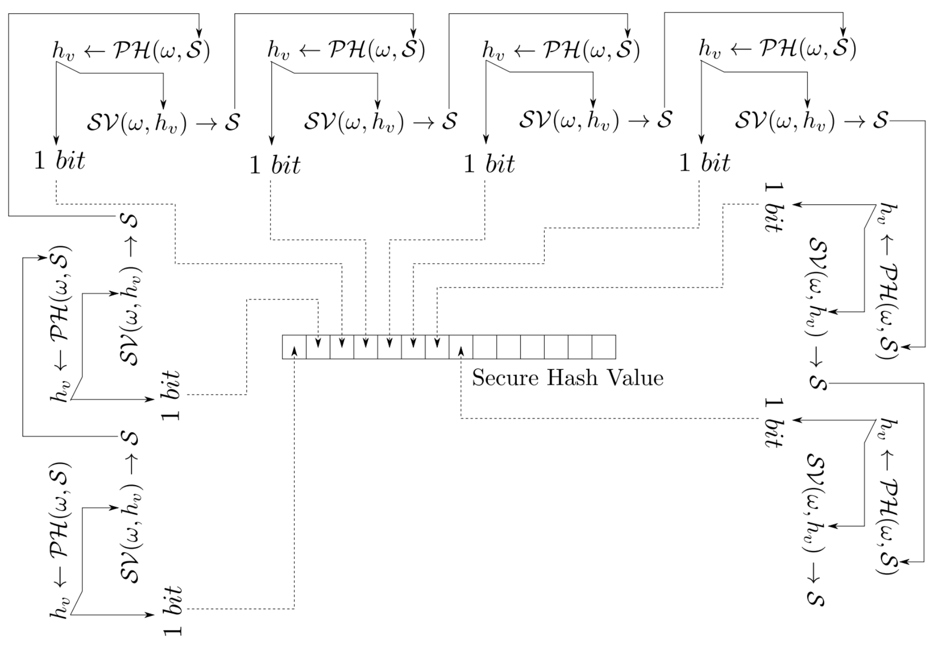

Figure 3.1: Architecture for 1D variant for generating 8-bit secure hash value

1D-variant provides a streamlined architecture, as illustrated in Figure 3.1, for generating a randomized secure hash. The input string is represented by ω, while the seed value is denoted as S. The primary hash function is referred to as $PH()$, and the output of this hash function is called h_v. Figure 3.1 shows the process where the primary hash function is invoked, and a seed value is computed using the $SV()$ function to produce a single bit for the randomized secure hash.

In this design, the input string ω remains constant, while the seed value S is updated with each bit of the hash generated. The example in the figure illustrates the generation of an 8-bit secure hash value, although users can define the desired output length according to their needs. It imposes no limitations on the size of the final hash, allowing it to be flexible—whether it's 1024 bits or any other size, depending on the application. In contrast, traditional secure hash algorithms, such as SHA-256, generate fixed-length hash values, with SHA-256 limited to producing a 256-bit output.

Algorithm 1 Computing seed value.

procedure GETSEEDVALUE($\omega, \mathcal{L}, S$)
    for $i \leftarrow 1$ **to** τ **do** $\triangleright$ τ is a constant and $8 \leq \tau \leq 64$.
        $S \leftarrow \text{PrimaryHash}(\omega, \mathcal{L}, S)$
    return S

We introduce a function for generating the seed value, as detailed in Algorithm 1. This algorithm modifies the publicly available seed into a private seed using between 8 and 64 primary hash functions ($8 \leq \tau \leq 64$). The purpose of this process is to safeguard the publicly available seed, converting it into a private value by incorporating the input string. The core concept behind Algorithm 1 is to protect the seed value from being easily computed, ensuring stronger security for the seed throughout the process.

Algorithm 2 1D randomized, variable-sized SHA.

procedure GENDSSHA-1D(ω, S, η)
    $\mathcal{L} \leftarrow \text{stringLength}(\omega)$
    $S \leftarrow \text{getSeedValue}(\omega, \mathcal{L}, S)$
    $S \leftarrow S \oplus \eta$
    for $i \leftarrow 1$ **to** η **do**
        $h_v \leftarrow \text{PrimaryHash}(\omega, \mathcal{L}, S)$
        $\rho \leftarrow h_v \bmod \varrho$ $\triangleright$ The $\varrho = (\beta - c)$ is a prime number.
        $\text{bit} \leftarrow (h_v \wedge (1 \ll \rho)) \gg \rho$
        $\text{hash\_bits}[i] \leftarrow \text{bit}$
        $S \leftarrow h_v$
        $S \leftarrow \text{PrimaryHash}(\omega, \mathcal{L}, S)$
    $\zeta \leftarrow \text{convertIntoHex}(\text{hash\_bits}, \eta)$
    return ζ

Algorithm 2 describes 1D-variant, optimized for high performance. It requires an additional space complexity of $O(1)$, excluding the memory needed for storing the hash output in ζ. Initially, the seed value is public, but it is transformed into a private value by repeatedly applying the primary hash function $8 \leq \tau \leq 64$ times, such that $S = \text{PrimaryHash}(\omega, L, S)$. The algorithm computes a bit position $\rho = h_v \% \varrho$, where $\varrho = (\beta - c)$ is a prime number. For example, if the primary hash function generates a 32-bit hash value, $c = 1$, and $\varrho = 31$, a prime number, so $\rho = h_v \% 31$, resulting in a single-bit index of h_v. The ρ-th bit of h_v is then extracted to form part of the secure hash, either as a 0 or 1. For larger bit sizes, ϱ would be 61 for 64-bit or 127 for 128-bit outputs.

The core idea is to use the primary hash functions, such as Murmur2, to generate bits for the final secure hash value. The primary hash function first produces a β-bit non-secure hash value, and DSSHA-1D extracts a single bit from this β-bit output to contribute to the secure hash. The next step involves using a second hash function to generate a new seed value, which is then used to compute the next hash value hv. This process is repeated η times to produce an η-bit secure hash, which is ultimately converted to hexadecimal and stored in ζ.

However, this algorithm is not entirely a randomized secure hash algorithm, as it lacks true randomness in generating the secure hash value. Additionally, it does not possess memory-hardness properties, which means it is not designed to resist attacks that exploit parallel processing and high memory bandwidth.

3.3.4 Two-dimensional randomized, variable-sized SHA

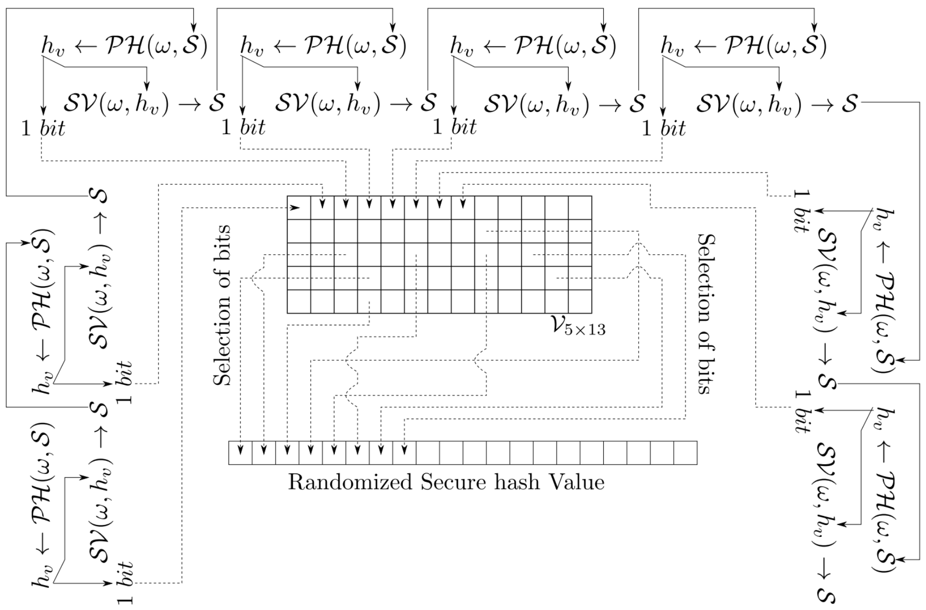

Figure 3.2: Architecture for 2D variant for generating 8-bit secure hash value

Figure 3.2 illustrates 2D randomized, variable-sized SHA, a process designed to generate a fully randomized secure hash value ζ. This figure corresponds to Algorithm 4, which enhances the security of the generated hash value by balancing medium security with medium performance. Here, we introduce a two-dimensional vector, $V_{X\times Y}$, where X and Y are prime numbers. This vector is used to create a pool of pseudo-random bits, consisting of 0s and 1s, from which η bits are selected to form the final randomized secure hash.

The key idea of using this vector is to populate it with random bits and then choose specific bits to generate a secure hash. The selection of these bits is determined by two primary hash functions. The 2D variant algorithm is carried out in three main stages: (a) determining the dimensions, (b) filling the vector, and (c) retrieving the bits. In the first stage, the dimensions X and Y of the vector are computed. In the second stage, the vector is filled with pseudo-random bits. Finally, in the third stage, the algorithm retrieves η-bits from the vector to construct the randomized secure hash value.

This structure ensures a more randomized and secure approach to hash generation by utilizing the vector of random bits and the dual primary hash functions.

Algorithm 3 Generation of dimension for vector $\mathcal{V}$.

---

procedure GENDIM(S, θ)
    $d \leftarrow S\%\theta$ ▷ Extracting the digits.
    $dim \leftarrow \sqrt{d}$
    $dimension \leftarrow (dim < 16)?(dim + 16) : dim$
    return $dimension$

Algorithm 4 2D randomized, variable-sized SHA.

---

procedure GENDSSHA-2D(ω, S, η)
    $\mathcal{L} \leftarrow$ stringLength(ω) ▷ Stage: Dimension - Starts
    $S \leftarrow$ getSeedValue$(\omega, \mathcal{L}, S)$
    Initialize θ ▷ θ is used to extract a number from the computed seed value S for the calculation of a dimension.
    $r \leftarrow$ genDim(S, θ)
    $S \leftarrow$ getSeedValue$(\omega, \mathcal{L}, S)$
    $c \leftarrow$ genDim(S, θ)
    $X \leftarrow$ prime$[r]$, $Y \leftarrow$ prime$[c]$ ▷ $X \neq Y$
    $S \leftarrow S \oplus \eta$ ▷ Stage: Dimension - Ends, Stage: Filling - Starts
    for $i \leftarrow 1$ **to** X **do**
        for $j \leftarrow 1$ **to** Y **do**
            $S \leftarrow$ getSeedValue$(\omega, \mathcal{L}, S)$
            $h_v \leftarrow$ PrimaryHash$(\omega, \mathcal{L}, S)$
            $\rho \leftarrow h_v \bmod \varrho$ ▷ The $\varrho = (\beta - c)$ is a prime number.
            bit $\leftarrow (h_v \wedge (1 \ll \rho)) \gg \rho$
            $\mathcal{V}[i][j] \leftarrow$ bit
            $S \leftarrow h_v$
▷ Stage: Filling - Ends, Stage: Retrieving - Starts
    for $k \leftarrow 1$ **to** η **do**
        $S \leftarrow$ getSeedValue$(\omega, \mathcal{L}, S)$
        $h_v \leftarrow$ PrimaryHash$(\omega, \mathcal{L}, S)$
        $i \leftarrow (h_v \bmod X) + 1$
        $S \leftarrow h_v$
        $S \leftarrow$ getSeedValue$(\omega, \mathcal{L}, S)$
        $h_v \leftarrow$ PrimaryHash$(\omega, \mathcal{L}, S)$
        $j \leftarrow (h_v \bmod Y) + 1$
        $S \leftarrow h_v$
        hash_bits$[k] \leftarrow \mathcal{V}[i][j]$
▷ Stage: Retrieving - Ends
    $\zeta \leftarrow$ convertIntoHex(hash_bits, η)
    return ζ

Dimension: To define the dimension $V_{X \times Y}$, two prime numbers X and Y are required. These are derived from a seed value by hashing it repeatedly τ times, where $8 \leq \tau \leq 64$. The seed value S is updated as $S = \text{PrimaryHash}(\omega, \mathcal{L}, S)$ during each iteration. This repeated hashing process converts the public seed value into a private one using the input string, enhancing security. Algorithm 3 calculates the dimensions based on the seed value. The maximum size of the dimension depends on the digit extracted from S using a tunable constant θ (e.g., $\theta = 12967$, preferably a prime). A digit is computed as $d = S\%\theta$, and the dimension is derived as $\sqrt{d}$. For two-dimensional vectors, Algorithm 3 is invoked twice to calculate $r = \text{genDim}(S, \theta)$ and $c = \text{genDim}(S, \theta)$, corresponding to $X = \text{prime}[r]$ and $Y = \text{prime}[c]$, where prime[] is an array of pre-computed prime numbers. Ensuring $X \neq Y$ adds complexity and security. This approach conceals the dimensions X and Y from adversaries, making the 2D-variant a memory-hard secure hash algorithm. The unknown dimensions increase the difficulty for adversaries to compute the hash function.

Filling: To populate the $V_{X \times Y}$ vector, the algorithm generates $X \times Y$ pseudo-random bits (0s and 1s). Each bit is derived using the 'getSeedValue()' function, which produces a hash value h_v via the primary hash function. A single bit is extracted from h_v to fill one cell of the vector. This process requires computing $X \times Y \times \tau$ seed values to fully populate the vector.

Retrieving: After filling the vector, the algorithm selects specific bits from it to generate a randomized secure hash value. Each cell in the vector is accessed using two indices, i and j, which are computed using the primary hash function. Specifically, $i = \text{PrimaryHash}(\omega, \mathcal{L}, S)\%X$ and $j = \text{PrimaryHash}(\omega, \mathcal{L}, S)\%Y$. The seed value is updated through 'getSeedValue()' before each computation. This process is repeated η times to extract η-bits from the vector, ensuring fair and even distribution of slots if X and Y are prime numbers. In total, $2\eta\tau$ seed values are computed during retrieval.

3.3.5 Three-dimensional randomized, variable-sized SHA

Similar to the 2D variant algorithm, the 3D randomized, variable-sized SHA algorithm is divided into three stages: a) dimensions, b) filling, and c) retrieving. Algorithm 5 computes three dimensions X, Y and Z such that $X \neq Y \neq Z$ in the dimension stage, following the same approach as in the 2D algorithm. The filling and retrieving processes are performed similarly, with the only difference being the addition of the third dimension.

Algorithm 5 3D randomized, variable-sized SHA.

procedure genDSSHA-3D(ω, S, η)
  $\mathcal{L} \leftarrow$ StringLength(ω) ▷ Stage: Dimension - Starts
  $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
  Initialize $\theta \leftarrow 0$ ▷ The θ is used to extract a number from the computed seed value S for the calculation of a dimension.
  $r \leftarrow$ GenDim(S, θ) ▷ The 3 is dimension, i.e., 3D.
  $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
  $c \leftarrow$ GenDim(S, θ)
  $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
  $w \leftarrow$ GenDim(S, θ)
  $X \leftarrow$ Prime[r], $Y \leftarrow$ Prime[c], $Z \leftarrow$ Prime[w] ▷ $X \neq Y \neq Z$; Stage: Dimensions - Ends
  $S \leftarrow S \oplus \eta$ ▷ Stage: Filling - Starts
  for $i \leftarrow 1$ to X **do**
    for $j \leftarrow 1$ to Y **do**
      for $k \leftarrow 1$ to Z **do**
        $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
        $h_v \leftarrow$ PrimaryHash($\omega, \mathcal{L}, S$)
        $\rho \leftarrow h_v \% Z$ ▷ The $\rho = (\beta - c)$ is a prime number, for instance, 31 or 61.
        $bit \leftarrow (h_v \& (1 \ll \rho)) \gg \rho$
        $\mathcal{V}[i][j][k] \leftarrow bit$
        $S \leftarrow h_v$
  ▷ Stage: Filling - Ends
  for $k \leftarrow 1$ to η **do** ▷ Stage: Retrieving - Starts
    $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
    $h_v \leftarrow$ PrimaryHash($\omega, \mathcal{L}, S$)
    $i \leftarrow (h_v \% X) + 1, S \leftarrow h_v$
    $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
    $h_v \leftarrow$ PrimaryHash($\omega, \mathcal{L}, S$)
    $j \leftarrow (h_v \% Y) + 1, S \leftarrow h_v$
    $S \leftarrow$ GetSeedValue($\omega, \mathcal{L}, S$)
    $h_v \leftarrow$ PrimaryHash($\omega, \mathcal{L}, S$)
    $k \leftarrow (h_v \% Z) + 1, S \leftarrow h_v$
    $hash\_bits[k] \leftarrow \mathcal{V}[i][j][k]$
  ▷ Stage: Retrieving - Ends
  $\zeta \leftarrow$ ConvertIntoHex($hash\_bits, \eta$)
  return ζ

Chapter 4

Experimental Results and Discussion

Overview

Below, the results and their implications are analyzed.

4.1 Results

Algorithm	**Output**	**Collision**	**Preimage**	**Second Preimage**
Random oracle	η	$2^{\eta/2}$	2^{η}	2^{η}
SHA3-224	224	2^{112}	2^{224}	2^{224}
SHA3-256	256	2^{128}	2^{256}	2^{256}
SHA3-384	384	2^{192}	2^{384}	2^{384}
SHA3-512	512	2^{256}	2^{512}	2^{512}
SHAKE128	η	$2^{\min(\eta/2,128)}$	$\geq 2^{\min(\eta,128)}$	$2^{\min(\eta,128)}$
SHAKE256	η	$2^{\min(\eta/2,256)}$	$\geq 2^{\min(\eta,256)}$	$2^{\min(\eta,256)}$
DSSHA	η (fixed)	$2^{\eta/2}$	2^{η}	2^{η}
DSSHA	$\eta \in [\mu, \lambda]$	$\sum_{\eta=\mu}^{\lambda} 2^{(\eta)/2}$	$\sum_{\eta=\mu}^{\lambda} 2^{\eta}$	$\sum_{\eta=\mu}^{\lambda} 2^{\eta}$

Table 4.1: Comparison of Various Hash Algorithms

The table above presents a comparison of various hash algorithms, focusing on the **output size**, **collision resistance**, **preimage resistance**, and **second preimage resistance**. It includes both standard cryptographic hash functions (such as SHA3 and SHAKE) and the DSSHA family of algorithms, which are based on randomized secure hashing techniques.

4.1.1 Key Observations

- **Output Size and Security Strength:** The output size of the hash function directly influences its security strength. Algorithms with larger output sizes, such as SHA3-512 (512 bits) or SHA3-384 (384 bits), provide stronger security compared to those with smaller output sizes, like SHA3-224 (224 bits).

  The DSSHA family is flexible with its output size, denoted by η, allowing it to scale according to specific security needs. This flexibility makes it adaptable for different applications that may require varying security levels.

- **Collision Resistance: Collision resistance** refers to the difficulty of finding two distinct inputs that hash to the same output. As expected, algorithms with larger output sizes (e.g., SHA3-512) have higher collision resistance values, represented by $2^{\eta/2}$.

  The DSSHA algorithms have more collision resistance as compared to the standard hash functions as the output size is not known.

- **Preimage Resistance: Preimage resistance** is the computational difficulty of finding an input that maps to a given hash value. This resistance is also influenced by the output size of the hash function. Larger output sizes, such as those in SHA3-512 or SHA3-384, provide higher preimage resistance values (e.g., 2^{η}).

  DSSHA algorithms also show high preimage resistance, scaling with the output size η. For the same output size, DSSHA provides similar preimage resistance to that of the established SHA3 and SHAKE algorithms.

- **Second Preimage Resistance: Second preimage resistance** is the difficulty in finding a second distinct input that hashes to the same value as a given input. Like collision resistance, this metric increases with larger output sizes. For instance, SHA3-512 offers 2^{512} resistance, which is significantly higher than SHA3-224's 2^{224}.

  The DSSHA family exhibits similar second preimage resistance, where the value is 2^{η}, consistent with the general trend that larger output sizes provide better security.

- **DSSHA Variants:** The DSSHA algorithms (1D, 2D, 3D) comes in different bit vector sizes. As expected, the larger-bit versions provide stronger security (higher collision, preimage, and second preimage resistance), similar to the standard SHA3 algorithms.

  The generalized **DSSHA** algorithm, which allows for a range of output sizes from μ to λ, provides flexible security depending on the specific value of η. The resistance values are expressed as sums over the chosen range of output sizes.

4.2 Discussion and Analysis

4.2.1 Analysis of Trade-offs

Trade-off 1 *Performance vs. Security in Hash Functions*

There is a significant trade-off between the performance and security of the hash function used in our algorithm. High-bit-sized primary hash functions (e.g., 64-bit versions) provide better security compared to low-bit-sized hash functions (e.g., 32-bit versions). For example, the probability of generating a correct hash value without knowing the input string is $\frac{1}{2^{32}}$ for a 32-bit hash function and $\frac{1}{2^{64}}$ for a 64-bit hash function. This makes 64-bit hashes more secure due to their lower probability of collisions. Using high-bit hash functions also allows for more secure bit selection. For instance, selecting a bit among 61 bits (using a modulus operation with a prime like 61) offers better security than selecting from 31 bits. However, this enhanced security comes at the cost of slower performance, as high-bit-sized hash functions require more computational resources. While stronger cryptographic functions like MD5, SHA1, or SHA2 could be used as primary hash functions, they are slower compared to simpler hash functions such as Murmur2, XXHash, or FastHash. Additionally, increasing the number of primary hash function invocations further impacts performance. Hence, our algorithm must balance the need for higher security with the practical requirement for faster computation.

Trade-off 2 *Dimensions of the Bit Vector vs. Performance*

Increasing the dimensions of the bit vector enhances security but reduces performance. A larger bit vector provides a more extensive pool of pseudo-random bits, improving the randomness and security of the generated hash value. However, forming a large bit vector requires more invocations of the primary hash function, significantly increasing the time complexity of the algorithm. While a larger bit vector strengthens security, the additional computational overhead impacts performance, as it takes longer to process a larger pool of bits. Thus, our algorithm must balance the trade-off between achieving better security through larger bit vectors and maintaining efficient algorithm performance.

4.2.2 Parallelism in Generating Hash Values using DSSHA

Parallel processing is a technique that enables solving problems concurrently, significantly improving execution time. However, this can pose a challenge for secure hash algorithms. For instance, an algorithm with $O(n)$ time complexity can be reduced to $O(1)$ in parallel execution. If an adversary can solve a problem in $O(n)$ time on a sequential system, they can potentially solve the same problem in $O(1)$ time with parallel execution, which undermines the security of hash algorithms.

Consequently, generating a single hash value in parallel is not recommended. The DSSHA hash function generates hash bits sequentially, which limits its efficiency in parallel execution. This inefficiency arises because the output of one primary hash function serves as the input for the next, creating a strong dependency between the hash bits. In such cases, parallel processing becomes disadvantageous when there is a reliance on previous computations or a strong interdependence between the input and output.

Therefore, the generation of a single hash value using DSSHA is best performed sequentially, as parallel execution can lead to reduced security. However, multiple hash values can be generated in parallel through distributed computing frameworks, such as MapReduce. In this setup, multiple map tasks can generate different hash values, with a reduce task collecting the results. While this approach benefits from parallelism, generating multiple hash values in a MapReduce framework can still be inefficient. This inefficiency stems from the memory hardness of hash computations, which prevents MapReduce from fully leveraging parallelism.

Chapter 5

Conclusion and Future Work

5.1 Conclusion and Future Work

5.1.1 Conclusion

The proposed methodology for hash value generation using DSSHA offers a highly efficient and secure framework for generating randomized secure hash values. DSSHA effectively enhances the security of hash values while maintaining a balance between performance and time complexity. The experimental results demonstrate that DSSHA provides a computationally efficient approach for hash value generation, with scalability for various input sizes and scenarios.

However, despite its advantages, optimization is needed to reduce the time complexity and improve its applicability in real-time scenarios or distributed computing systems.

In conclusion, DSSHA shows great potential for secure hashing, particularly in contexts where high security is paramount. However, additional improvements are required to address performance bottlenecks and enable the algorithm's efficient use in more complex environments.

5.1.2 Future Work

Several areas for improvement and exploration have been identified to enhance the proposed methodology:

1. **Algorithm Implementation:** A full fledged implementation of the algorithm in a suitable programming language.

2. **Testing with different suites:** Testing with NIST SP 800-22 which is a statistical test suite that validates random and pseudo random number generators for cryptographic applications.

These enhancements aim to make the algorithm more versatile, accurate, and applicable across a broader range of scenarios.

Bibliography

[1] Colin Percival,"Stronger key derivation via sequential memory hard function". Published 2009

[2] A. Biryukov, D. Dinu and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," 2016 IEEE European Symposium on Security and Privacy (EuroS&P), Saarbruecken, Germany, 2016, pp. 292-302, doi: 10.1109/EuroSP.2016.31.

[3] Boneh, D., Corrigan-Gibbs, H., Schechter, S. (2016). Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks. In: Cheon, J., Takagi, T. (eds) Advances in Cryptology – ASIACRYPT 2016. ASIACRYPT 2016. Lecture Notes in Computer Science(), vol 10031. Springer, Berlin, Heidelberg.

[4] J. Alwen and J. Blocki, "Towards Practical Attacks on Argon2i and Balloon Hashing," 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, 2017, pp. 142-157, doi: 10.1109/EuroSP.2017.47.

[5] Al-Kuwari, S., Davenport, J.H., Bradford, R.J.: Cryptographic hash functions: Recent design trends and security notions. Cryptology ePrint Archive, Paper 2011/565 (2011)

[6] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Advances in Cryptology– EUROCRYPT 2008, pp. 181–197. Springer, Berlin, Germany (2008)

[7] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Advances in Cryptology– EUROCRYPT 2013, pp. 313–314. Springer, Berlin, Germany

[8] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Making of KECCAK. Cryptologia 38(1), 26–60 (Jan 2014)