
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI - 600036

# FUNCTIONAL VERIFICATION OF SECURE HASH ALGORITHM

*A Project Report*

*Submitted by*

**SUDHA JAIN**

**EE19M065**

*In the partial fulfilment of requirements*

*For the award of the degree*

*Of*

**MASTER OF TECHNOLOGY**

July 2021

# CERTIFICATE

This is to undertake that the Project report titled **FUNCTIONAL VERIFICATION OF SECURE HASH ALGORITHM**, submitted by me to the Indian Institute of Technology Madras, for the award of M.Tech, is a bona fide record of the research work done by me under the supervision of **Prof. Veezhinathan Kamakoti**. The contents of this Project report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Place: Chennai 600 036**
**Date: 18th June 2021**

**SUDHA JAIN**
**EE19M065**

**Prof. Veezhinathan Kamakoti**
Research Guide

**Dr. Boby George**
Research Co-Guide

# ACKNOWLEDGEMENTS

I express my sincere thanks to **Prof. Veezhinathan Kamakoti** and **Dr. Boby George** for their guidance and constant encouragement throughout the project work. I am grateful to them for providing me their valuable time through various sessions to discuss the project work which enabled me to take this project to fruitful completion.

Special thanks to Prof. Veezhinathan Kamakoti for giving the opportunity to work in the Shakti Processor Project, helped in understanding Digital Design and Implementation. I am greatly indebted to him for the knowledge on VLSI design that I gained during the course of my project work.

My sincere gratitude to Prof. Boby George for allowing me to do the project under Computer Science and Engineering department

I would also like to thank my Mentor, **Lavanya Jagan** for explaining me the of Verification plan and for helping me in learning the verification tools CoCoTb Verilator.

My sincere thanks to all my friends at IIT Madras, who supported me during my stay in the campus that made it really enjoyable and memorable.

Finally, I thank my family for their support and constant encouragement.

# ABSTRACT

Data probity assuredness and data validation are critical security features among many areas. Secure algorithms for data security are provided by cryptographic hash functions described by the NIST. These functions are used for digesting data and generate a hash message which is a 1-way function that is very secure and hard to reverse.

The CoCoTb (Coroutine cosimulation testbench): a Python-based digital logic verification framework is used to verify one such hash method in this research. The **Secure Hash Algorithm 2 (SHA-2)** is a collection of cryptographic functions described by the National Security Agency (NSA) of the United States and originally published in 2001. They are formed with the Merkle–Damgard construction, which starts with a 1-way compression function designed with the Devies–Meyer structure by a particular block cypher.

**Cocotb verilator** can be used to do functional verification of the design, such as the Secure Hash Algorithm. The test bench must be written in such a way that the design module can be checked in all potential scenarios.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABBREVIATIONS

| | |
|---|---|
| SHA | Secure Hash Algorithm |
| AES | Advanced Encryption Standard |
| MD | Message Digest |
| NIST | National institute of Science and Technology |
| NSA | National Security Agency |
| ASIC | Application-Specific Integrated Circuit |
| HDL | Hardware Description Language |
| UVM | Universal Verification Methodology |
| DUT | Design Under Test |
| VPI | Verilog Procedural Interface |
| FPGA | Field Programmable Gate Array |
| SoC | System on Chip |
| EDA | Electronic Design Automation |
| ISA | instruction Set Architecture |
| OOP | Object-Oriented Programming |
| VCS | Verilog Compiler Simulator |
| TLS | Transport Layer Security |
| SSL | Secure Sockets Layer |
| PGP | Pretty Good Privacy |
| SSH | Secure Shell |
| ICT | International Criminal Tribunal |
| IPsec | Internet Protocol Security |
| IITM | Indian Institute of Technology Madras |
| HDVL | Hardware Description and Verification Language |
| FIPS | Federal Information Processing Standards |

# CHAPTER 1

# INTRODUCTION

Cryptography is difficult to master, and the only way to know if anything was done correctly is to analyse it. This is a solid argument in favour of open source encryption methods... However, just exposing the code does not guarantee that it will be examined for security issues.

[1999] Bruce Schneier

Commercial encryption software should be regarded with caution... [due to] back doors. ... " Whenever possible, employ public-domain encryption that is interoperable with other implementations.

[2013] Bruce Schneier

The goal of ***System Verilog*** development was to create a uniform language that would meet the needs of users in terms of design and verification. It was first advocated by the Accellera system initiative, and it later became an IEEE standard for Hardware Description and Verification Language after numerous changes (HDVL).

To meet a variety of verification and modelling needs, System Verilog supports **object-oriented programming** (OOP) ideas. Despite its extensive capabilities, System Verilog has unable to gain general acceptance. The adoption of SystemVerilog was hampered by a slew of cultural and practical issues, but it prepared the way for additional libraries, toolkits, and methodology guidelines.

As a result, the **Universal Verification Methodology** (UVM) was created, which is a library that combines SystemVerilog and OOP ideas. UVM, on the other hand, has comparable concerns with complexity. UVM has approximately 300 classes to choose from. Again, it's quite strong, but it's also really tough to get started.

As a result, in our SHA2 verification, we used **Python**, and **CoCoTb** provided an interface between the simulator and Python.

In this project, such an environment is utilised to test the hash function algorithm. Hash functions convert any arbitrary data into fixed-size data, and the results they return are known as hashes or digests. They are a crucial and widely used encryption component.
The size of a hash value is determined by the algorithm that was used to generate it.

This project builds hash functions using the SHA-256 algorithm, which generates a **256-bit hash** (64 characters).

## 1.1 Research Goals

To construct a verification environment for the SHA-256 hash function. The following tasks are important for the project's success.

- Understand the SHA-256 hash functions' algorithms.
- Create a **CoCoTb testing environment** and establish a link between the test components and the RTL core of hash functions.
- Examine several test strategies to ensure IP's functionality.
- Examine assertion-driven methods that could be used.
- Examine the RTL core's functionality and timing relationships in contrast to a software model.
- Gather the results of the functionality and coverage tests.
- Run gate level simulations to ensure that the technology-dependent netlist is correct.
- Ensure that you have enough functional coverage to prevent corner case issues.

## 1.2 Thesis Structure

The following is how the rest of the thesis is organised:

- The basic introduction to Hash Functions is covered in **Chapter 2** with a literature review. The classification and applications of known hash functions are mentioned.
- The Secure Hash algorithm is explained in detail in **Chapter 3** so that the proposed technique can be understood.
- The verification framework utilised, CoCoTb, is introduced in **Chapter 4**. (Coroutine Co-simulation Testbench).
- **Chapter 5** explains the Testbench Architecture and description of testbench tools that have been used in developing the testbench.
- The simulations and results are presented in Python in **Chapter 6**.
- The conclusion and future work are included in **Chapter 7**.

# CHAPTER 2

# Hash Functions: An Overview

Secure Writing is the definition of cryptography. It determines the message's context from both the sender and the recipient. A branch of science that deals with the process of changing a readable message into an unreadable one and then returning the message to its original form is referred as **Cryptography**.

Electronic-mail, net banking, files transfer etc. are all examples of electronic processes used in today's environment. Cryptography has assumed a critical role in data conversion security. The message Hash or digest is a task that maps a message of irregular extent to a string of constant length. The SHA specification, released by the NIST includes these hash algorithms: **Secure Hash Algorithm-224, 256, 384, and 512**. Hash tasks are mostly used to protect the purity function. Siltu (2007)

## 2.1 PROPERTIES AND DEFINITIONS

### 2.1.1 Hash Function Definition

It is a process which takes an input and outputs a fixed-length string called **hash value**. Depending on algorithm, the input string can be of any length.The output is generally referred to as a **message digest** because it is a shorter version of the original message. Depending on the algorithm being utilised, the message digest's size is finalised.

This implies that all input streams provide the same length output for a given algorithm. Furthermore, even little input changes results in an completely distinct hash value. The **avalanche effect** is what this is called. In Figure 2.1, the hashing operation is depicted.

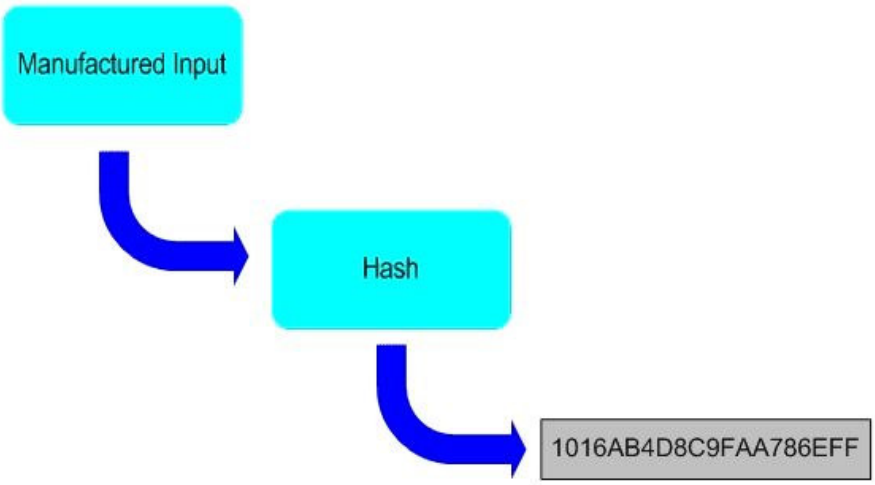


Fig. 2.1: Hashing Operation

The message digest length is directly proportional to the security of a hash function. First, pre-image resistance, secondly, second pre-image resistance, and the third collision resistance are all critical properties of any Hash Functions.

### 2.1.2 Pre-image Resistance (Property.1)

**One-wayness** (pre-image resistance):
It is computationally very hard to get an input message with the specified hash value for all given hash values. Figure 2.2 illustrates this characteristic.

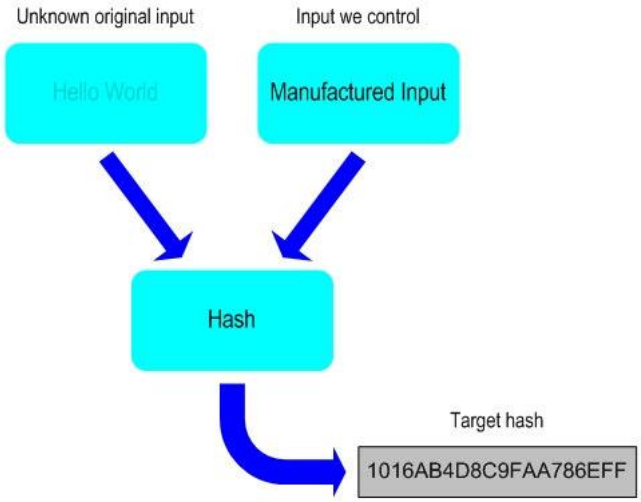


Fig. 2.2: Pre-image Resistance

### 2.1.3 Second Pre-image Resistance (Property.2)

Second pre-image resistance:

If we are provided with an input message say, m 1, finding another input message m2 in such a manner that **hash ( m 1 ) = hash ( m 2 )** is computationally very difficult.

Figure 2.3 illustrates this characteristic.

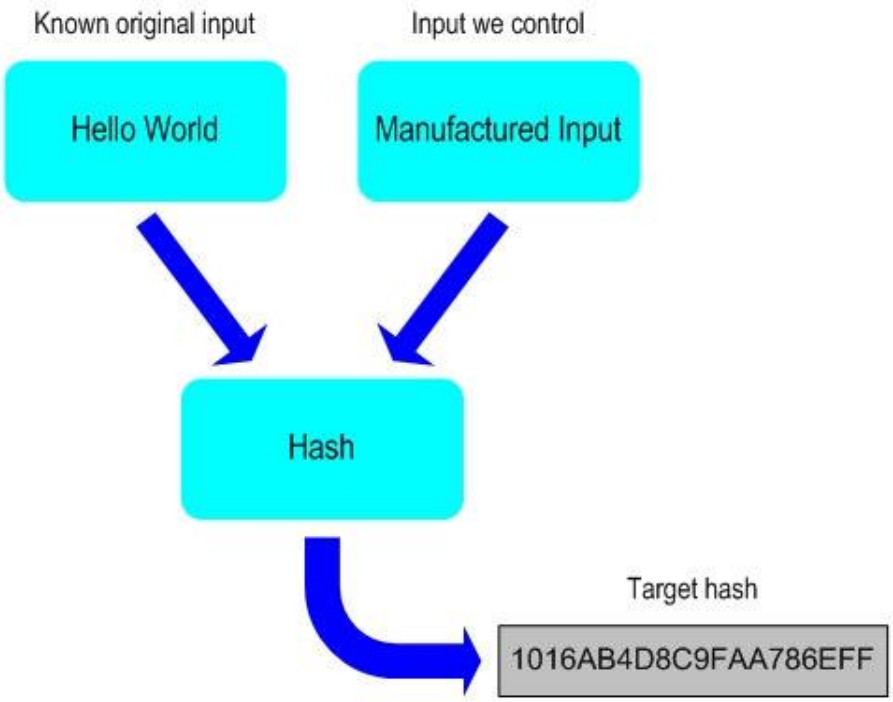


Fig. 2.3: Second Pre-image Resistance

### 2.1.4 Collision Resistance (Property.3)

Collision resistance:

Finding two dissimilar inputs with the identical hash value is computationally very difficult.

Figure 2.4 illustrates this characteristic.

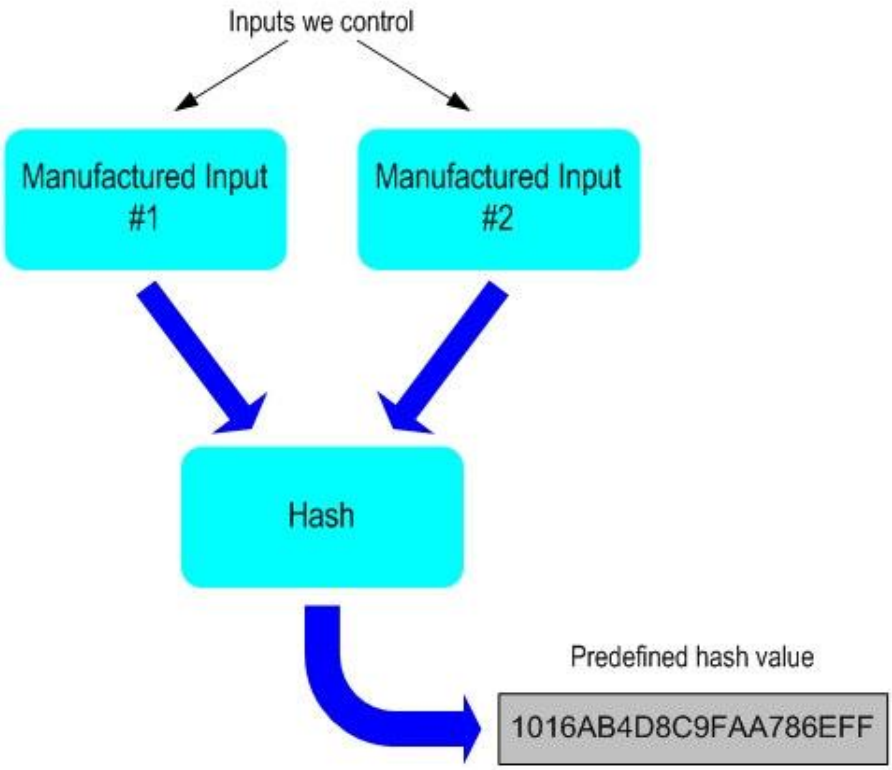


Fig. 2.4: Collision Resistance

Hash functions are classified as: **keyed and unkeyed hash functions**.

A private key is given as an another input parameter to keyed hash algorithms. In this situation, any value of the secret key satisfies the above-mentioned hash function features.

**Message Authentication Codes**, or MACs, are another name for keyed hash functions. We exclusively look at unkeyed hash functions in this paper.

## 2.2 Construction of Hash Functions

There are various ways to arrange hash functions, but the Merkle-Damgard model is the most popular and stable. It has been successfully used in functions like SHA-2. A message is padded and separated into uniform length chunks in this paradigm. The compression function F generates an intermediate hash by processing the blocks in a sequential order.

The hash message is output by the final compression procedure. The hash message's size is determined by the user's implementation. **Merkle-Damgard** is followed by SHA-2 and MD5.

For the compression purposes, they use operations like OR, XOR, and AND. Collision-pairs was created, exposing these techniques to collision attacks. Inspite of the fact, there was no confirmation that this algorithm had been successfully attacked. SHA-3 looks to be more secure than other SHA-2 variations, according to the NIST. Figure 2.5 illustrates Merkle-Damgard model.

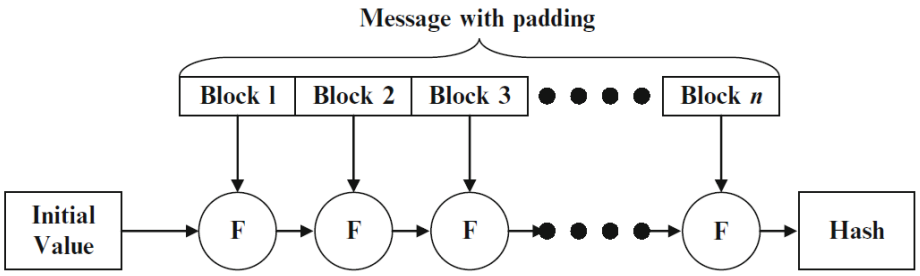


Fig. 2.5: Merkle-Damgard Model

SHA-2 can operate on four different modes, SHA-256, SHA-384, SHA-224,and SHA-512. This project uses SHA-256 and discussion is only based on SHA-256. SHA-256 also uses the Merkle-Damgard model and works similar to MD5. The message block size is 512 Bits like MD5, but the state variables are doubled to include 8 state variables. The operations performed on each stage are **AND, OR, XOR**.

All well-known hash functions, including MD4, MD5, SHA-1 and SHA-2 are built from block-cipher-like components designed for the purpose, with feedback to ensure that the resulting function is not invertible.
SHA-3 finalists included functions with block cipher-like components though the function finally selected, was built on a cryptographic sponge instead.

## 2.3 Applications

### 2.3.1 Check for Data Integrity

Data integrity is a crucial component of any safe system. By utilising a hash function to create the message digests for the files, any modifications to the files can be identified.

Digests are kept, and then, the digest is recalculated on the file in the and if the recent digest differs from the initial digest, shows that the original file is corrupted in some way. When it comes to protecting **crucial system binaries and sensitive datasets**, this is very crucial.

To ensure that the obtained file is same as the original file, The message digest of the received file is calculated. The original message digest provided by WEB site or FTP site is then compared to this calculated message digest.
Because it is arithmetically tough to discover 2 inputs with the identical hash value (a collision resistance property), if the calculated digest differs from the original, the received file varies from the transmitted file.
The use of a hash function to **verify data integrity** is shown in Figure 2.6.

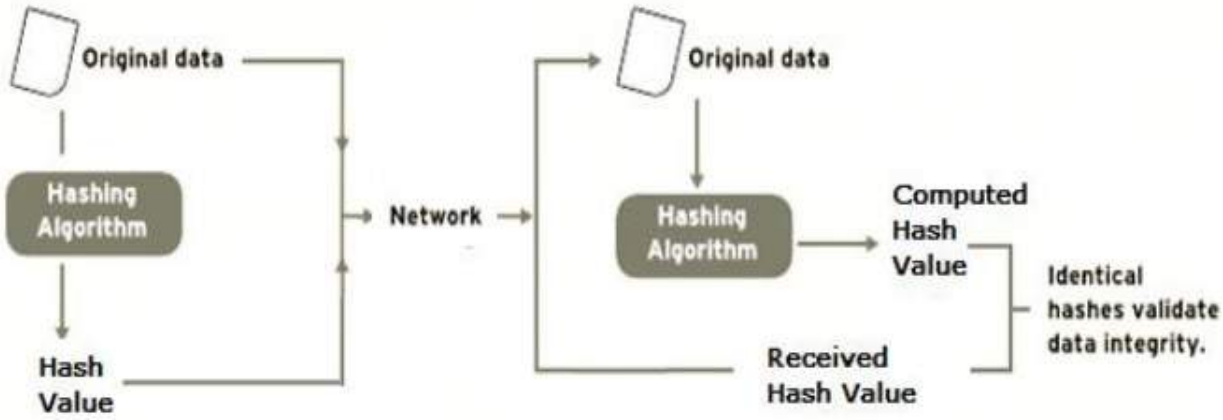


Fig. 2.6: Verifying Data Integrity

### 2.3.2 Authentication with a Password

Another field where hash functions are employed is password authentication. Clear-text password storage is insecure for computer systems. Someone could gain access to all

of the credentials, compromising the entire user password database. Because of these factors, storing password hashes rather than plain text passwords is a more safe option.

When a user registers in, the submitted password's hash value is determined and analogized to the value saved in the password database. User gets authenticated if the observed hash value matches the one recorded; else, the user is denied.
This scenario is depicted in the Figure 2.7 below.

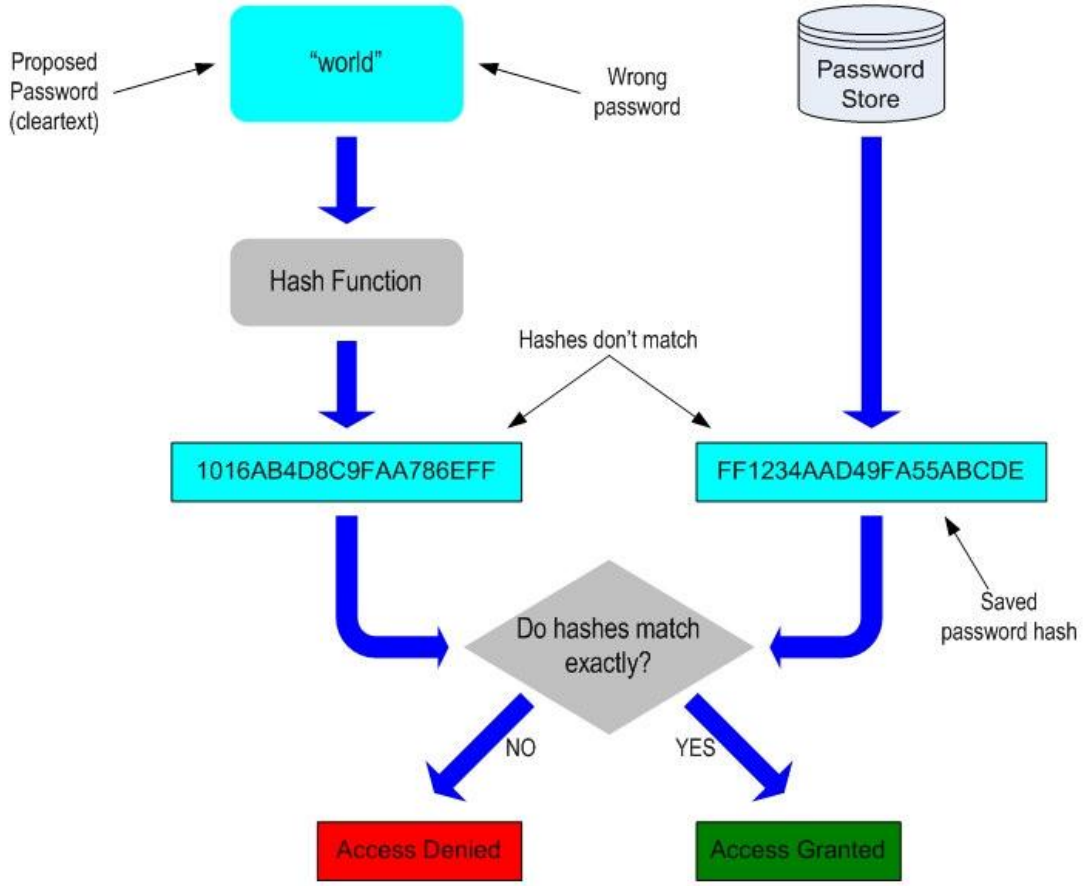


Fig. 2.7: Authentication with a Password

**User privacy is preserved, despite the password database is exposed**, because obtaining the original passwords from the hash values is computationally exceedingly challenging.

### 2.3.3 Digital Signatures with Encryption

Digital signatures are one of the most common uses of hash functions. A digital signature is a type of electronic signature. It is an algorithm widely accustomed to ensure **the authenticity and validity of a file**.
Many applications emerge from the combination of a public-key method with hash functions, one of which is digital signature.

A e-signature can be generated using a combination of a hash message and private-key encryption. It is possible to use the Generated Text as a signature. It could be validated by decrypting it with a public key and equating it to the hash message.

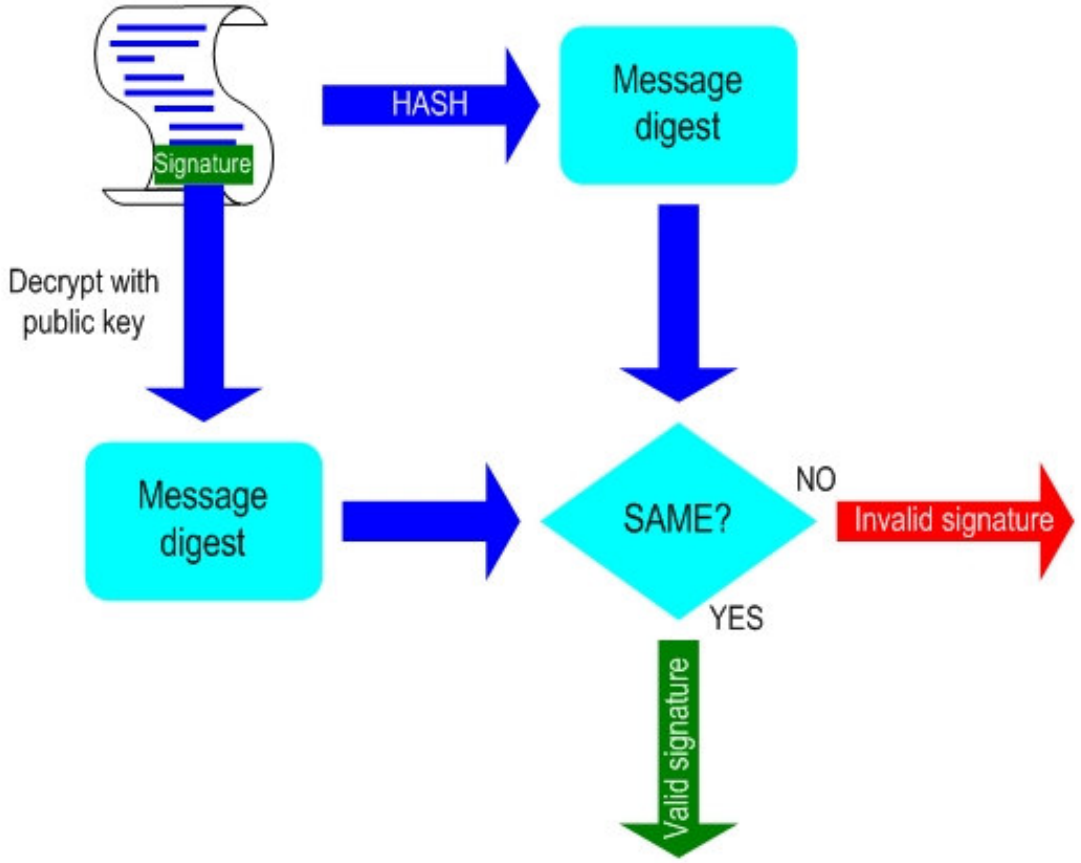


Fig. 2.8: Verification of a Digital Signature

The user could not be sure that the data integrity is secured if a hash function was not utilised. **Any change to the document will result in a change to the signature, which will not be validated because hash functions are one-way functions** .

Therefore, when the signature is verified, the user verifies that the file has not been tampered with. Additional benefit of digital signatures is that **they allow the source of messages to be verified**. A valid signature verifies that the message was transmitted by the sender who owns the private key used in the encryption procedure.

## 2.4 KNOWN HASH FUNCTIONS

There is several hash functions developed up to now and among these hash functions MD5, SHA-1, and SHA-256 are most popular.

### 2.4.1 Summary of the standard hash functions

| Algorithm | Output size | Block size | Word size | Rounds xSteps | Year of the standard |
|---|---|---|---|---|---|
| MD4 | 128 | 512 | 32 | 16x3 | 1990 |
| MD5 | 128 | 512 | 32 | 16x4 | 1991 |
| RIPEMD | 128 | 512 | 32 | 16x3 (x2 parallel) | 1992 |
| RIPEMD-128 | 128 | 512 | 32 | 16x4 (x2 parallel) | 1996 |
| RIPEMD-160 | 160 | 512 | 32 | 16x5 (x2 parallel) | 1996 |
| SHA-0 | 160 | 512 | 32 | 80 | 1993 |
| SHA-1 | 160 | 512 | 32 | 80 | 1995 |
| SHA-256 | 256 | 512 | 32 | 64 | 2002 |
| SHA-224 | 224 | 512 | 32 | 64 | 2004 |
| SHA-384 | 384 | 1024 | 64 | 80 | 2002 |
| SHA-512 | 512 | 1024 | 64 | 80 | 2002 |

Fig. 2.9: Summary of the standard hash functions

Ron Rivest proposed MD4 in 1990, and it was developed for greater-speed software implementations on 32-bit processors employing 32-bit operations. MD stands for **message digest**. However, because to a collision problem, **MD4 was reorganised to MD5 in 1991**, with advancements like compression rounds were increased from three to four. The MD5 compresses data into 512-bit blocks, which are then subdivided into 16 32-bit subblocks. The word has a 32-bit size.

The number of rounds is an important parameter for compression functions. MD5 provides a **64-round compression mechanism**. MD5 is a widely used hash algorithm in a variety of applications, including **IPsec**.
However, it was pointed out that utilising MD5's compression function, collisions can be formed. It was projected that by constructing dedicated technology at a cost of 10 million dollars, two messages with the identical hash value can be identified in around 25 days.

National Institute of Standards and Technology (NIST) demarked **the SHA (Secure Hash Method) 160-bit function to use with the DSS (Digital Signature Standard) digital signature algorithm in 1993.** Soon after, a method for causing collisions inside the function of compression was discovered by studying the message extension function, which was made up entirely of XOR (exclusive OR).

To change that, the message expansion function was changed to Secure Hash Algorithm-1 by inserting a 1-bit rotation. Because **Secure Hash Algorithm-1** is meant to meet the level of security of the block cypher , which utilises an **80-bits hidden key**.

Secure Hash Algorithm-1 is a 512-bit block algorithm with five 32-bit chaining variables that is modelled after MD5. The output is 160 bits long. SHA-1 contains more rounds (80 instead of 64) than MD5, despite the fact that the round functions are less diversified and simpler. For extracting 32-bits smaller sub blocks from a 512-bit message, SHA-1 employs a more complicated process. More than half of the sub blocks are modified if one bit of the message is reversed, whereas this amount is only four for MD5.

These two designs are strikingly similar to SHA-1, however these are far nearer to one another than to mutual forerunner. **SHA-384** is a simple alteration to SHA-512 that involves reducing the output to 384 bits and modifying the chaining variable's start value.

The technique for obtaining 32-bit sub blocks from a single message block is the most significant distinction between the 3 recent hash functions and SHA-1. The collisions have been reported for already known hash functions and it has been hypothesised that SHA-1 could be broken. As a result, the transition to more secure hash algorithms should be sped up.
As a starting point, the Secure hash algorithm-1 and Secure hash algorithm-256 are used in this research. The reason for this choice is because Secure hash algorithm-1 is among the widely used hash ..functions, while **SHA-256 was evolved after SHA-1 and provides higher degrees of security.**

Both of these functions, as previously stated, work with 512-bit message blocks and have the same word size of 32,.. bits. Despite these are identical generally, they differ in terms of the no. of chaining variables, production of 32-bitsmaller sub blocks from 512-bit message blocks.

### 2.4.2 Limitations and Algorithms

- SHA-1: It necessitates a significant amount of computational energy and resources.
- SHA-2: Because SHA256 and SHA512 have **higher collision resistance**, they create larger outputs (i.e. 256 bits and 512 bits, respectively) than SHA1 (160 bits). Those who defend the usage of SHA2 point to the larger output size as a rationale for attack resistance.
- SHA-3: SHA-3 was developed to be an efficient hash function, not a good password-hashing-scheme (PHS), while on the contrary bcrypt was created to be a PHS and was also evaluated in that direction.
- MD5: It is not a great idea to work with salted MD5 for passwords. MD5 is not popular of its **cryptographic flaws**, but because it is quick. This means that on a single GPU, an attacker could test millions of user passwords each second.

# CHAPTER 3

# Secure Hash Algorithm SHA-256

## 3.1 SHA-2

SHA-2 (Secure Hash Algorithm 2) Bashkaran (2019) is a collection of cryptographic hash functions developed by the NSA of the United States and originally released in 2001. They are constructed with the **Merkle–Damgard model**, which starts with a **1-way compression function** designed with the Davies–Meyer structure from a particular block cypher.

SHA-2 is disimilar from its antecedent, SHA-1, in a number of manners. **SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256** are the six hash functions of the SHA-2 family, having digests (hash values) of 224, 256, 384, or 512 bits. SHA-256 and SHA-512 are two new hash algorithms that use eight 32-bit and 64-bit words, respectively, to construct their hashes.

They employ **distinct shift amounts and cumulative constants**, but their structures are ver much similar, with the number of rounds being the sole difference.
SHA-224 and SHA-384 are reduced variants of SHA-256 and SHA-512, calculated by differing beginning values, respectively. Starting values for SHA-512/224 and SHA-512/256 are created following the procedure defined in **Federal Information Processing Standards (FIPS)** PUB 180-4.

NIST initially issued SHA-2 as a government standard in the United States (FIPS).

The patent was distributed under a **royalty-free licence by the United States**. The best public attacks now defeat preimage resistance in 52 of 64 rounds of SHA-256 and 57 of 80 rounds of SHA-512, as well as collision resistance in 46 of 64 rounds of SHA-256.

## 3.2 Hash Standard

NIST issued 3 additional hash functions to the SHA family with the release of FIPS PUB 180-2. SHA-2 refers to a group of algorithms that are called by the digest lengths (in bits): **SHA-256, SHA-384, and SHA-512**.

FIPS PUB 180-1, that was issued in April 1995, was replaced by the new Secure Hash Standard i.e. FIPS PUB 180-2 that was published in August 2002.. New standard enclosed the original SHA-1 algorithm, along with modified technical notation that was compatible with the SHA-2 family's inner workings.

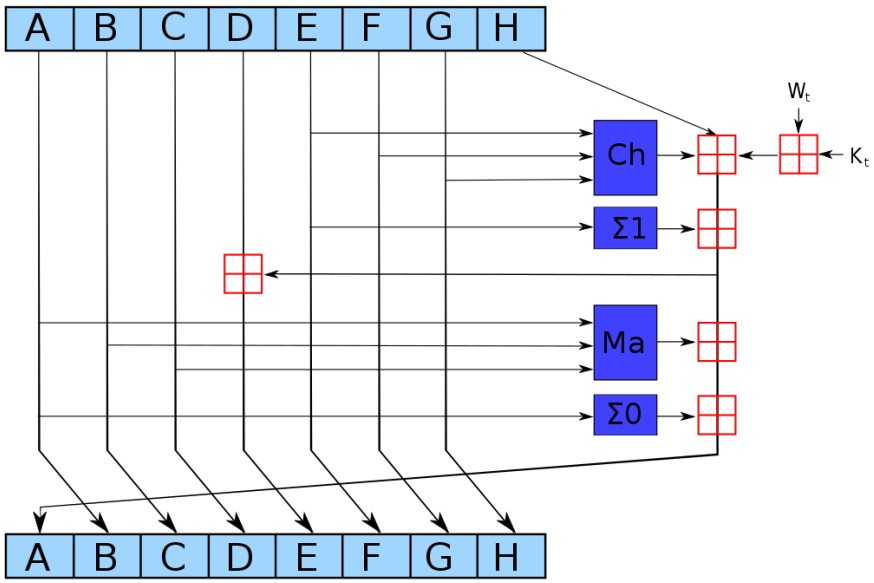


Fig. 3.1: Hash Standard

The bitwise rotation uses different constants for SHA-512. The given numbers are for SHA-256. The red is addition modulo 232 for SHA-256, or 264 for SHA-512.

A modification notice for FIPS PUB 180-2 was issued in February 2004, describing a second variation, SHA-224, which was described to meet the key length of 2-key Triple DES. This was revised in FIPS PUB 180-3 in October 2008.

Dislocating security data concerning hash algorithms and suggestions to use them to Special Publications 800-107 and 800-57 was the major impetus for changing the standard. In addition, **elaborated test data and sample message digests** were separated from the standard and made available like independent papers.

The criterion was revised in March 2012 with FIPS PUB 180-4, which added the hash functions SHA-512/224 and SHA-512/256 as well as a technique for creating start values for cutshort versions of Secure hash algorithm-512.

The limit on elaborating input information before computation had been also eliminated, and thus, permitting hash data for evaluating concurrently for matter synthesis, like for a real-time video or audio stream. Prior to hash output, **padding the last final data block** is however required.

## 3.3 SHA-256 Pseudo Code

Note 1: Variables are all of unsigned 32-bit integers, and summation is carried out using modulo 232. Note 2: There is one message schedule array item w[i], 0 to 63, and one round constant k[i] for each round. Note 3: There are eight working variables, a to h in the compression function. Note 4: When describing constants in the pseudo code and converting message block data from bytes to words, big-endian convention is utilised.

The initial step is to **set up the values of hash**: the initial 32 bits of the non-integral real portions are the square roots of the first 8 primes 2.....19

Step 2: **Create a circular constants array**: The initial 32 bits of the non-integral real portions of the cube roots of the first 64 primes 2....311 are the initial 32 bits of the fractional portions of the cube roots of the first 64 primes.

**Pre-processing (Padding)** is the third step. add the '1' bit to the message Add n bits of '0', where n is the smallest number greater than or equal to 0, results in a string length of 448 bits.

Adjoin the message's length in bits such as a 64-bit big-endian integer and this would generate the whole post-processed length a multiple of 512 bits.

Fourth step : **Break up the message into 512-bit chunks** Make a 64-entry message scheduling array w[0..63] containing 32-bit words for each chunk.

Step 5: Federal Information Processing Standards schedule array into the remaining 48 words w[16..63]: for I between 16 and 63 s0 : = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3) s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightrotate 20) xor (w[i-2] rightrotate 21) xor (w[i-2] rightrotate 22) (w[i-2] rightshift 10) w[i] = w[i-16] + s0 + w[i-7] + s1

Step 6: **Initialize working variables to current hash value** Compression function main loop:

for i from 0 to 63 S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
ch := (e and f) xor ((not e) and g)
temp1 := h + S1 + ch + k[i] + w[i]
S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
maj := (a and b) xor (a and c) xor (b and c)
temp2 := S0 + maj
h := g
g := f
f := e
e := d + temp1
d := c
c := b
b := a
a := temp1 + temp2
7th Step : **Compress the chunk** and add it to the recent hash value.
8th Step: **Generate the final hash value** in big-endian format: assemble : = hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7 append

h8 append h9 append h10 append h11 append h12 append h13 append h14 append h15 append h16 append h17 append The only differences between SHA-224 and SHA-256 are:

- the starting hash values i.e. from h0 to h7 are distinct
- Also, the output is produced by deleting h7.

Wikipedia contributors (2021)

## 3.4 Applications

**IPsec ,TLS and SSL, and SSH** are just a few of the **security applications** and protocols that employ the SHA-2 hash algorithm. SHA-256 is employed for verifying Debian software packages whereas Secure Hash Algorithm-512 is used **to verify archived film from the ICT for Rwanda's genocide**. DNSSEC proposes the usage of SHA-256 and SHA-512. For safe password hashing, Unix and Linux providers are switching to SHA-2 256 bits and 512-bits.

SHA-256 is employed by several cryptocurrencies, including Bitcoin, to verify transactions deals and calculate evidence of work.

Scrypt-based proof-work systems have become popular as ASIC SHA-2 accelerator processors have become more common. **The Hash Algorithms (SHA-1 and SHA-2) are needed by law to use in specific applications of US government**, which includes usage inside other algorithms of cryptography and protocols and thus, for securing sensitive data that is not classified. FIPS PUB 180-1 have also urged private and commercial enterprises to adopt and utilise SHA-1. For most government applications, SHA-1 is being phased out.Selvakumar and Ganadhas (2009)

According to the US National Institute of Standards and Technology, "**Agencies should definitely utilise the SHA-2 family of hash functions** for purposes that need collsion ressitance after 2010." . The mandate from the National Institute of Standards and Technology (NIST) that all US federal entities must discontinue using SHA-1 later than 2010 was intended for hastening the transition inspite from SHA-1.

# CHAPTER 4

# Verification Framework : CoCoTb

## 4.1 Verification

### 4.1.1 Basics of Verification

Verification demonstrates functional correctness of design and the Correctness is described by the specification.

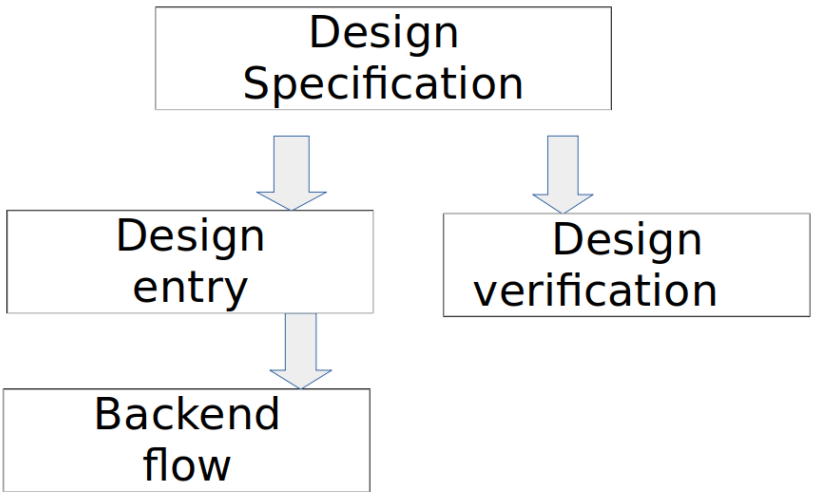


Fig. 4.1: Verification of Basics

The most significant part in the product development operation is the **Design Verification**, accounting for up to 80 percent of the overall time spent on the project. The goal is to make that the design fulfils all of the system's needs and standards.

(1) The simulation within which detailed functionality and timing of the design can be checked is referred to as **Logic simulation** and circuit simulation;

(2) The verification inside which functional models explaining the functionality of the design are created to check the behavioural specification of the design without detailed timing checks is referred to as **functional verification** and

(3) **formal verification**, which involves comparing functionality to a "golden" model.

**Model checking**, within which the design's properties are checked against some assumed "properties" specified in the functional or behavioural model and **equivalence checking**, in which the functionality is checked against a "golden" model, are also part of formal verification.
Although equivalence examining can be employed to evaluate synthesis outputs at lower levels of the **EDA cycle, property checking** is required for the initial design capture.

### 4.1.2 Abstraction Levels of Verification

• The processor's interface with other components is emphasised at the **SoC level**.
• At the core, ISA compliance and micro-architecture are prioritised.
• Level of the unit The various pipeline phases have traditionally had considerable control.

### 4.1.3 Methodology of Verification

• Development of a test plan
• Development of a test bench
• Simulation/Formal/FPGA based verification
• Analysis and review of coverage metrics
• Verification sign-off

### 4.1.4 Test Strategy

• Documenting how the design will be validated
• What design features should be tested under what circumstances?
• What method will be used to verify each feature?
• What is the stimulus, reference model, and process for checking?
• Assigning owners, deadlines, and prioritisation
In the design of integrated circuits, simulation is very significant. A designer can use simulation to verify a design's usefulness and performance before embarking on the costly and time-consuming process of manufacturing it.

**Verification plan for SHA-256** is shown in the table .

| TestPlan id | Features | Sub Feature | TestPlan | Design Parameter |
|---|---|---|---|---|
| sha_1 | Input to engine (Pre-hash) | 256 bits | Contribute 8 initial hash values in the algorithm | input_engine_pre_hash |
| sha_2 | Input message value | 512 bits | Contribute input message value | input_engine_input_value |
| sha_3 | Enable to start execution | Should be 1 to start execution | To enable Input port of the engine | EN_input_engine |
| sha_4 | Output check | 1 bit, It will be set when the output is available. | Output to be checked if ready signal is set. | RDY_output_engine_get |
| sha_5 | Input engine ready | 1 bit, Should always be 1 | Check whether the input engine is ready when the signal is high | RDY_input_engine |
| sha_6 | Reference Model Output | 256bits | Monitor to ensure that the Reference Model output generates | expected_output |
| sha_7 | DUT Output | 256 bits | Monitors to ensure that DUT output generates | output_mon |
| | | | Compare whether the Reference model output and DUT output are equal | Scoreboard compare |

Fig. 4.2: Verification plan for SHA-256

### 4.1.5 Verification Types: Simulation

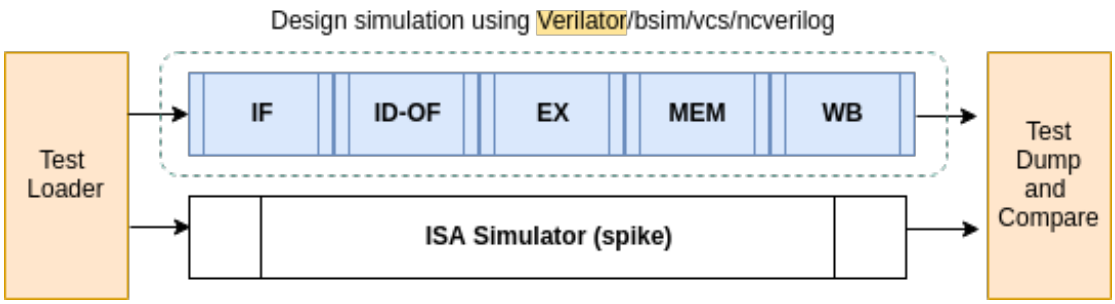


Fig. 4.3: Simulation

What to compare ? Processor state

Spike compares at the instruction level, micro-arch states are excluded.

Simulation plays an important role in the design of integrated circuits. Using simulation, a designer can determine both the functionality and the performance of a design before the expensive and time-consuming step of manufacture.

### 4.1.6 Verilator

• Verilator translates **synthesizable Verilog code into verilated files**, which are C++ executables.

• The user generates a C++ wrapper file that implements the design.

• The **executable** that runs the simulation is made up of the wrapper file that has been created and linked with the verilated files.

### 4.1.7 Environment for Verification

• CoCoTB is used to create the test bench in Python.

• Verilator is used for Design Simulation.

• The Verilator tool was used to collect **code coverage, and cocotb-coverage** was used to specify **functional coverage**. Constraint random generation is also included in this package.

## 4.2 CoCoTb

### 4.2.1 What is CoCoTb and how does it work?

• Cocotb is a **COroutine-based Cosimulation TestBench** environment for Python-based verification of SystemVerilog RTL and VHDL.
Cocotb is fully free, hosted on **GitHub and is open source**(under the BSD License).
• Cocotb needs a simulator to replicate the HDL design and have been tested on Linux, Windows, and macOS with a number of simulators.
• On EDA Playground, a live version of CoCoTb could be utilised inside a web browser.
• Cocotb is a Python library for **verifying digital logic**.
• Provides a Python interface for controlling common **RTL simulators** (such as Cadence, Questa, and VCS).
• Provides a verification alternative to the Verilog/SystemVerilog/VHDL framework.

### 4.2.2 Verification Methodologies

For verification, why not one should use Verilog or VHDL?

• **Hardware description languages (HDLs)** are useful for creating hardware and firmware.
• However, hardware design and verification are two distinct issues.
• It's possible that using the same language for both isn't the best option.
• Software, not hardware, is used to **create verification testbenches**.
• When developing sophisticated testbenches, **higher-level language** ideas (such as OOPs) are useful.

• Adding higher-level programming tools to a hardware description language is one method to improve the problem.
• For verification, use an existing general-purpose language.
• The first method is SystemVerilog, which has **simulation-only OOP language** capabilities.
• UVM libraries built in SystemVerilog (Universal Verification Methodology).

### 4.2.3 What makes CoCoTb unique?

• CoCoTb promotes the **same design reuse and arbitrary testing methodology** as Universal Verification Methodology, the difference is that it is **written in Python**.
• VHDL or SystemVerilog are typically used just for the design, not the testbench, with CoCoTb.
• Through standardised, CoCoTb provides in-built abet for interfacing with uninterrupted integration platforms like GitLab, and others.
• CoCoTb was created with the goal of reducing the time it takes to create a test.
• CoCoTb **detects tests automatically**, eliminating the need for a second step when adding a test to a regression.

Python is used for all verification instead of SystemVerilog or VHDL, because it offers **several benefits over for verification**:
• Federal Information Processing Standards
• The tests could be **modified and re-run without the need of recompiling** the design or exiting the simulator GUI.

### 4.2.4 CoCoTb: Basic Architecture

**What is CoCoTb and how does it work?**
• A conventional simulator is used to execute the design under test (DUT).
• CoCoTb acts as a bridge between the simulator and Python.
• Makes use of the Verilog Procedural Interface (VPI) or the **VHDL Procedural Interface (VHDL Procedural Interface)** .

• Python testbench code can:
1. Change values in the **DUT hierarchy**.
2. Wait for the simulation to finish.

3. Watch for a **signal's rising or dropping edge**.

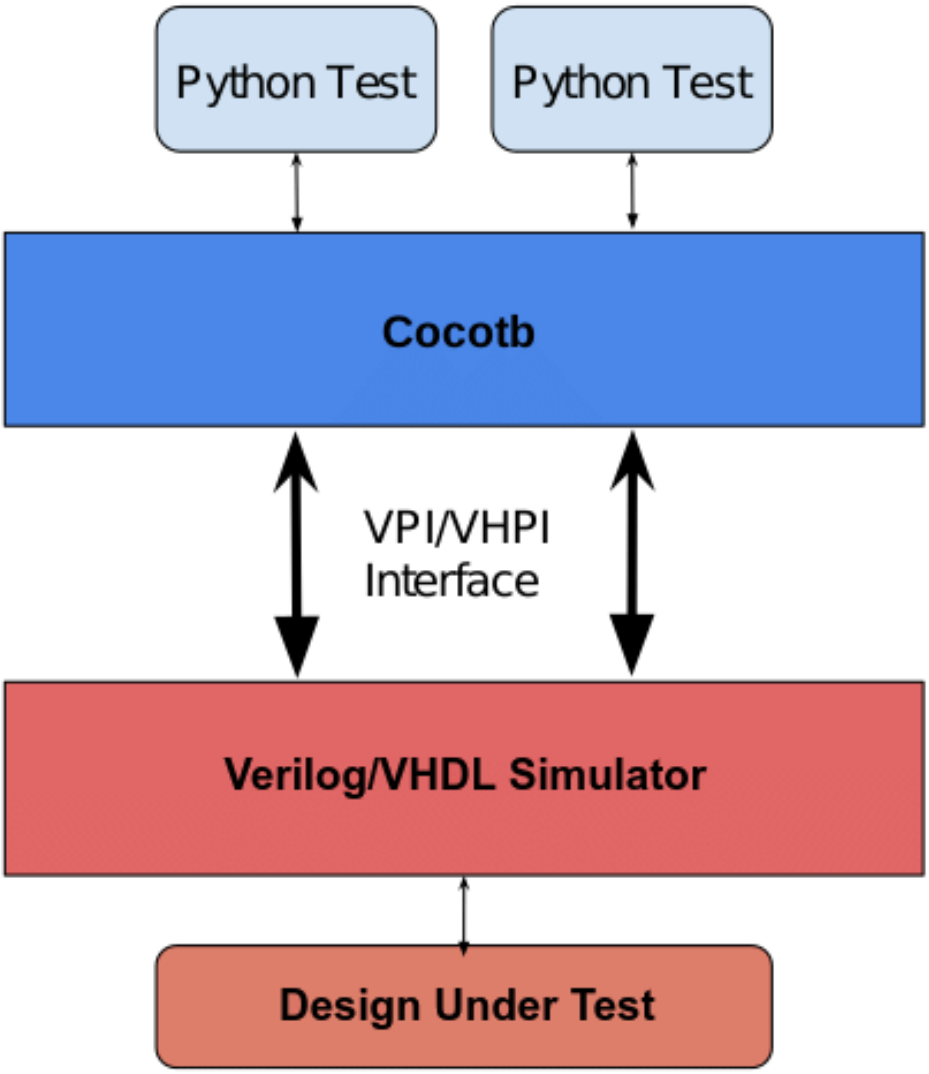


Fig. 4.4: CoCoTb Architecture

No additional RTL code is required for a normal cocotb testbench. Without any **wrapper code**, the DUT is initialised as the most upper level in the simulator.
Cocotb applies stimuli to the DUT's (or any higher-level) inputs and checks the outputs simply from Python. It's worth noting that the DUT should be completed definitely and as CoCoTb cannot generate HDL blocks;.

A testbench is nothing more than a **Python function**. Either the simulator or the Python code is running out of resources at any given time.

When passing control of operation back to the simulator, the **await keyword** is utilised. Multiple coroutines can be spawned by a test, allowing for independent execution flows.

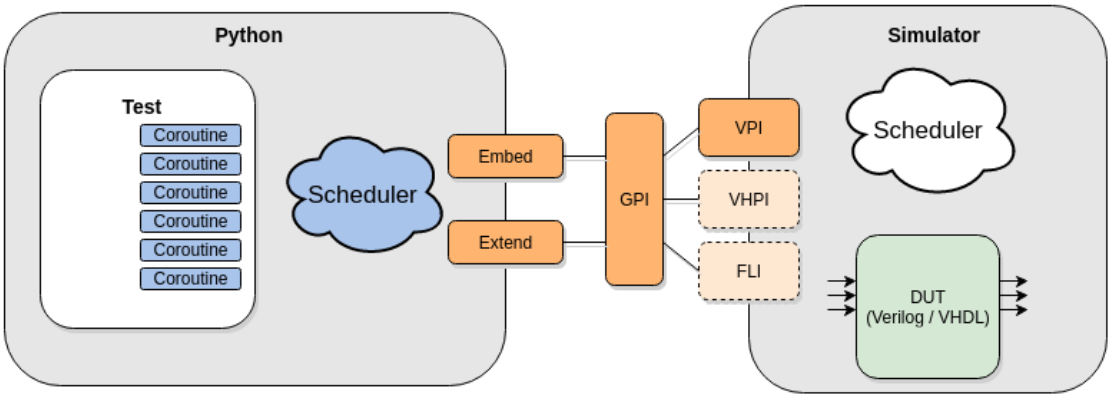


Fig. 4.5: How does CoCoTb work?

Cocotb is a powerful high-level programming language verification tool:

• **Far more powerful** than a standard Verilog testbench.

• More user-friendly than a SystemVerilog or UVM testbench.

Because of the **cosimulation methodology**, the RTL simulator is still used behind the scenes:

• Testbenches can contain a mix of Python and RTL.

• Cocotb testbenches can also be utilised **for simulations after synthesis**. Rosser (2018)

## 4.3 How to Use CoCoTb?

No additional RTL code is required for a normal cocotb testbench. Without any **wrapper code**, the DUT is initialised as the most upperlevel in the simulator. Cocotb applies stimuli to the DUT's inputs and monitors the outputs using Python.

### 4.3.1 How to Make a Makefile?

A Makefile is often required to create a Cocotb test. **Cocotb has a set of guidelines** that form it simple to learn from starting. We just tell CoCoTb about the original root

files that required to be compiled, the toplevel object that needs to be created, and the Python test script that needs to be loaded. Then we would make a file named test my design.py that contained all of our tests.

### 4.3.2 Putting together a test

Python is used to write the test. **Cocotb wraps the handle you pass** it around your top level. That handle is dut in this project, but you can use your own chosen name instead. In all Python files that reference your RTL project, the handle is used.

### 4.3.3 Obtaining access to the design

When cocotb starts up, it searches the simulator for the **upper-level initialisation and produces a handle named DUT**. The "dot" notation, which is used in Python to access object attributes, may be employed to be able to use the top-level signals. Signals within the design can be accessed using the same approach.

### 4.3.4 Assigning Signal Values

The value attribute of a handle object or direct during traversing the hierarchy can both be used to assign values to signals.

### 4.3.5 Obtaining data from signals

A BinaryValue object is returned when the value property of a handle object is accessed. Non-resolved bits are saved and can be retrieved with the **binstr attribute, and a resolved integer** value with the integer attribute.

# CHAPTER 5

# Testbench Architecture

## 5.1 Writing Testbench

### 5.1.1 Obtaining access to the design

Cocotb identifies the toplevel initialisation in the simulator and produces a handle named dut when it initialises. The **"dot" notation**, which is used in Python to **access object attributes**, may be employed to access toplevel signals. Signals within the design can be accessed using the same approach.

### 5.1.2 Assigning signals with values

The value attribute of a handle object during traversing the hierarchy can both be used to assign values to signals.

### 5.1.3 Values that are signed and unsigned

A Python int can be used to assign both signed and unsigned values to signals.Cocotb makes **no assumptions about the signal's signedness**.
It simply considers the width of the signal, hence it will accept values in the range of a signed number's minimum negative value to an unsigned number's maximum positive value:
**value = 2**Nbits - 1 -2**(Nbits - 1)**
Note that assigning values that are out of range will result in an OverflowError.

To assign a value to signals with greater fine-grained control, a BinaryValue object can be used instead of a Python int (e.g. signed values only).

### 5.1.4 Reading values from signals

The value property of a handle object can be used to access values in the DUT.
The HDL type of a handle determines the Python type of a value:
BinaryValue is the type for logic arrays and subtypes (sfixed, unsigned, etc.).
Integer nets and constants (integer, natural, and so on) always return int.
Constants (real) and floating point nets both return float.
The result of boolean nets and constants (boolean) is bool.
Any unresolved bits in a BinaryValue object are kept and could be retrieved with the binstr attribute.

### 5.1.5 Execution in parallel and in sequence

An await will start an async coroutine and then wait for it to finish. The current coroutine is "blocked" by the calling coroutine. Wrapping the call with fork() causes the coroutine to execute in the background, allowing the current coroutine to continue to execute.
You can wait for the outcome of the forked coroutine at any moment, which will block until the forked coroutine completes.

## 5.2 Coroutines

Coroutines are used in cocotb testbenches. The simulation is suspended while the coroutine is running. The await keyword is used by the coroutine to block on the execution of another coroutine or to return control of operations to the simulator, allowing simulation time to progress.

Mostly, coroutines await the arrival of a Trigger object, that informs the simulator of an event that would wake the coroutine when it occurs. Coroutines may potentially be waiting for other coroutines to complete.
• Coroutines have the ability to return a value, allowing them to be used by other coroutines.
• For concurrent operation, coroutines can be employed with fork().

• An await statement can be used with forked coroutines to block until the forked coroutine completes.

• Forked coroutines can be killed before they finish, pushing them to finish sooner than they would otherwise.

For async def coroutines, the cocotb.coroutine decorator is no longer required. Wherever decorated coroutines are accepted, including yield statements and cocotb.fork, async def coroutines can be used without the @cocotb.coroutine decorator.

## 5.3 Triggers

Triggers are used to tell the cocotb scheduler when coroutine execution should resume. A coroutine should be waiting for a trigger to be used. The current coroutine will be paused as a result of this. When the trigger fires, the paused coroutine will resume execution.

## 5.4 Testbench Structure

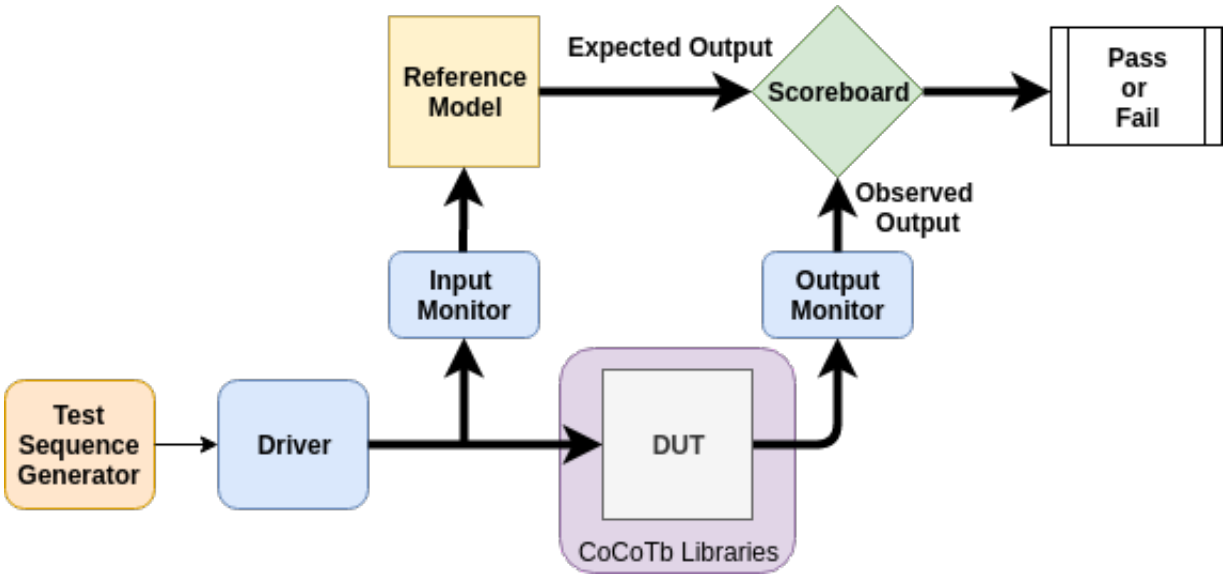


Fig. 5.1: Testbench Architecture

- UVM technique is used.
- To obtain the expected output, generated test sequences are sent to both the DUT and the model.

- In the Scoreboard, the observed output from the DUT is compared to the expected output from the model.

### 5.4.1 Logging

Cocotb makes use of the built-in logging library, with additional settings provided in Logging to provide some appropriate defaults. A logging is kept for each Design under test, scoreboard, driver, and monitor (and for any other function that uses the coroutine decorator).

Each logger has its logging level that could be customised. Individual logging levels can be defined for each hierarchical entity within a DUT. When logging is done for HDL objects, one should keep in mind that the preferred method is "log".
This reduces the chance of name clashes when using the Python logging feature with an HDL log component. Levels of log printing can also be configured per-object.

### 5.4.2 Bus

Buses are described as collection of signals. The Bus class will automatically bundle any group of signals together that are named similar to dut.<bus name><separator><signal name>.
A list of signal names, or a dictionary mapping attribute names to signal names is also passed into the Bus class. Buses can have values driven onto them, be captured (returning a dictionary), or sampled and stored into a similar object.

### 5.4.3 Driver

Examples and specific bus implementation bus drivers (AMBA, Avalon, XGMII, and others) exist in the Driver class enabling a test to append transactions to perform the serialization of transactions onto a physical interface.
classcocotbbus.drivers.Driver
Class defining the standard interface for a driver within a testbench.
The driver is responsible for serializing transactions onto the physical pins of the inter-

face. This may consume simulation time.

Constructor for a driver instance.

async driver send(transaction: Any, sync: bool = True, **kwargs: Any) → None

Actual implementation of the send.

Sub-classes should override this method to implement the actual send() routine.

Parameters

transaction – The transaction to be sent.

sync – Synchronize the transfer by waiting for a rising edge.

**kwargs – Additional arguments if required for protocol implemented in a sub-class.

async send(transaction: Any, callback: Callable[[Any], Any], event:

cocotb.triggers.Event, sync: bool = True, **kwargs) → None

Send coroutine

Parameters

transaction – The transaction to be sent.

callback – Optional function to be called when the transaction has been sent.

event – event to be set when the transaction has been sent.

sync – Synchronize the transfer by waiting for a rising edge.

**kwargs – Any additional arguments used in child class' driver send method.

### 5.4.4 Monitor

For our testbenches to actually be useful, we have to monitor some of these buses, and not just drive them. That's where the Monitor class comes in, with pre-built monitors for Avalon and XGMII buses. The Monitor class is a base class which you are expected to derive for your particular purpose.

You must create a "monitor recv()" function which is responsible for determining 1) at what points in simulation to call the recv() function, and 2) what transaction values to pass to be stored in the monitors receiving queue. Monitors are good for both outputs of the DUT for verification, and for the inputs of the DUT, to drive a test model of the DUT to be compared to the actual DUT.

For this purpose, input monitors will often have a callback function passed that is a model. This model will often generate expected transactions, which are then compared

using the Scoreboard class.
classcocotbbus.monitors.Monitor(callback=None, event=None)
Base class for Monitor objects.

Monitors are passive ‘listening’ objects that monitor pins going in or out of a DUT. This class should not be used directly, but should be sub-classed and the internal monitorrecv() method should be overridden. This monitorrecv() method should capture some behavior of the pins, form a transaction, and pass this transaction to the internal recv() method. The monitorrecv() method is added to the cocotb scheduler during the ...init.. phase, so it should not be awaited anywhere.
The primary use of a Monitor is as an interface for a Scoreboard.

Parameters
callback (callable) – Callback to be called with each recovered transaction as the argument. If the callback isn’t used, received transactions will be placed on a queue and the event used to notify any consumers.
event (cocotb.triggers.Event) – Event that will be called when a transaction is received through the internal recv() method. Event.data is set to the received transaction.

### 5.4.5 Scoreboard

Against compare actual outputs to expected outputs, the Scoreboard class is utilised. Actual outputs are represented by monitors on the scoreboard, while expected outputs might be either a simple list or a function that performs a transaction.

Ability to use a common scoreboard.
classcocotb bus.scoreboard.
Scoreboard(dut, reorder depth=0, fail immediately=True) Scoreboard(dut, reorder depth=0, fail immediately=True) object’s foundations Class for scoreboards in general.
By providing a monitor and an expected output queue, we can add interfaces.

The expected output might be either a transaction-generating function or a plain list

of the expected result.

Handle to the DUT (parameters dut (SimHandle))
Consider up to reorder depth elements of the expected result list as passing matches (int, optional). The default value is 0, which means that given a passing match, just the first element in the expected result list is taken into account.

Raise Test if fail immediately (bool, optional) is true.
Instead of simply documenting an error, failure occurs immediately when something goes wrong. True is the default value.

### 5.4.6 Assignment Procedures

• classcocotb.handle.**Deposit(value)**
Action used for placing a value into a given handle.
• classcocotb.handle.**Force(value)**
Action used to force a handle to a given value until a release is applied.
• classcocotb.handle.**Freeze**
Action used to make a handle keep its current value until a release is used.
• classcocotb.handle.**Release**
Action used to stop the effects of a previously applied force/freeze action.

cocotb contributors

# CHAPTER 6

# Results

## 6.1 Simulation Results

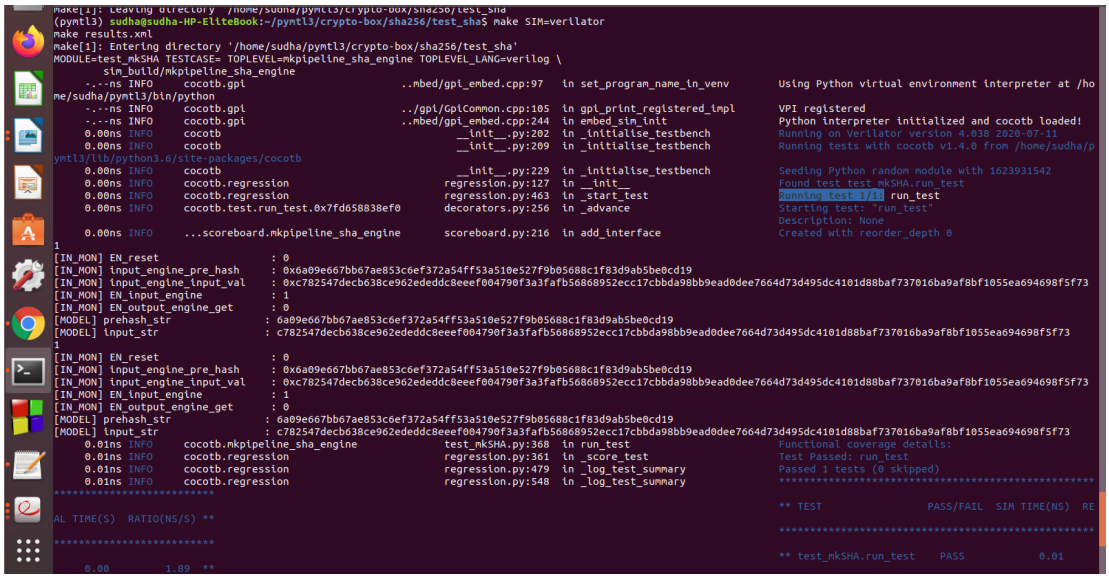


Fig. 6.1: Simulation Result

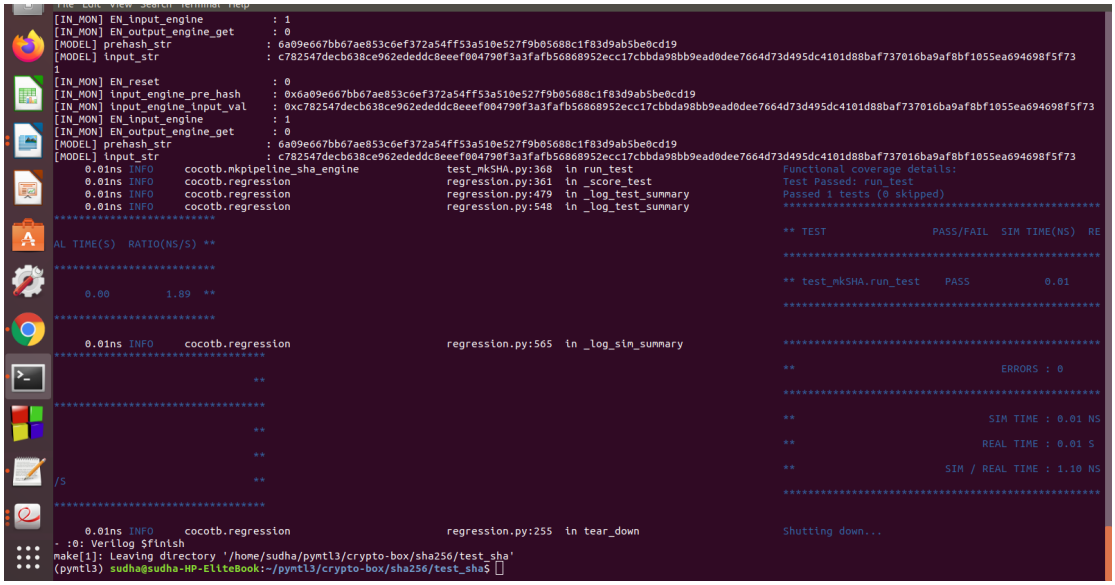


Fig. 6.2: Simulation Result

The python testbench for testing SHA-256 design has been verified using Verilator in the CoCoTb verification Environment. The pass results are shown in Figure 6.1 and Figure 6.2.

## 6.2 Code coverage

Code coverage is a metric used in computer science to characterise how much of a programme source code is executed while a test suite is run. When compared to a programme with low test coverage, a programme with high test coverage has had more of its source code executed during testing, implying that it has a lesser risk of holding undetected software problems. Test coverage can be calculated using a variety of metrics, the most basic of which are the percentage of programme subroutines and programme statements called during the execution of the test suite.

Code coverage helps in analysing how comprehensively a software is verified. It helps in measurement of efficiency of test implementation. It allows quantitative measurement. It also defines the degree to which the code has been tested.

**Verilator version v4.038** had been used to run simulations. **Cocotb version 1.4.0** had been used to write the testbench. Coverage results have been generated for 1 run to 1000000 runs. Generally, it is observed that until 100000 runs approximately, an increase in code coverage percentage is seen and after 100000 runs, the coverage percentage curve appears to flatten.

Test coverage results can be used by software authors to create additional tests and input or configuration sets to boost coverage of critical functions. Statement (or line) coverage and branch (or edge) coverage are two typical types of test coverage. Test coverage is one factor to consider while certifying avionics equipment for safety. The rules by which the Federal Aviation Administration (FAA) certifies avionics gear are documented in DO-178B and DO-178C.

# CHAPTER 7

# Conclusion

SHA-256 RTL cores were successfully verified using CoCoTb Verilator and the test-bench code was written in Python for the verification purpose. A reference model in C language was used to generate expected values and imported into the CoCoTb framework. DUT output data was compared with expected model values in the Scoreboard to verify the correctness of the each core. Assertions were used to verify the protocol over which each DUT communicates. This provided a level of confidence and ensured each RTL core functions properly. Random input generation was used to ensure high coverage.

## 7.1 Future Work

- We may determine that the code is not compact based on code coverage measures, thus, some RTL blocks can be modified to make the code more compact.
- Processing time for SHA-256 could be reduced if the RTL was enhanced.
- At any given time, SHA-256 can only handle one input. Pipelining could be implemented, allowing cores to accept streams of data as input.
- Because the timing on a few paths are not very great, certain nets can be re-designed.
- Functional coverage could be done more efficiently.
- Instead of using Randomizing without any constraints that might end up not hitting several possible regions. To resolve this, the randomization can be divided so as to cover each region.

# REFERENCES


1. **Bashkaran, D. A.** (2019). *Verification of SHA-256 and MD5 Hash Functions Using UVM*. Doctoral thesis, Department of Electrical Engineering(KGCOE).

2. **cocotb contributors** (). Cocotb read the docs, library reference. URL `https://docs.cocotb.org/en/latest/library_reference.html`.

3. **Rosser, B. J.** (2018). *Cocotb: a Python-based digital logic verification framework*. Master's thesis, University of Pennsylvania (US).

4. **Selvakumar, A. L.** and **C. S. Ganadhas** (2009). The evaluation report of sha-256 crypt analysis hash function. *In 2009 International Conference on Communication Software and Networks*.

5. **Siltu, C. T.** (2007). *Design and FPGA implementation of hash processor*. Master's thesis, Middle East Technical University.

6. **Wikipedia contributors** (2021). Sha-2 — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=SHA-2&oldid=1024045149`. [Online; accessed 18-June-2021].