NTNU
Norwegian University of
Science and Technology

# Java Implementation and Performance Analysis of 14 SHA-3 Hash Functions on a Constrained Device

**Mats Knutsen**
**Kim-André Martinsen**

Master of Science in Communication Technology
Submission date: June 2010
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

# Problem Description

NIST is running a four year worldwide competition for the next SHA-3 cryptographic hash function. Out of the 64 initial submissions, now in the second round, there are 14 candidates left. As the constrained devices are becoming more popular, it is important to have good performance of hash functions on such platforms as well.
The candidates will get a constrained device and development tools, and will have to transfer the existing C code for all 14 candidates to a code (C or Java) suitable for the constrained device, and to perform performance analysis.

Assignment given: 15. January 2010
Supervisor: Danilo Gligoroski, ITEM

*Performance is your reality. Forget everything else.*

-Harold S. Geneen

# Abstract

Several of the widely used cryptographic hash functions in use today are under attack. With the need to maintain a certain level of security, NIST is in the process of selecting new cryptographic hash function(s). Through a public competition the candidates will be evaluated and analyzed by the public and the winner(s) become the new standard cryptographic hash algorithm(s). Cryptographic hash algorithms have a wide range of applications, and the winner(s) will have to perform well in various platforms and application areas.

The number of constrained devices surrounding us at a daily basis is rapidly increasing. As these devices are used for a great variety of applications, security issues arise. The winning algorithm(s) will not only have to prove a strong security, but also show good performance and capability to run on constrained devices.

In this thesis, we show the results of our implementation of the second round SHA-3 candidates in Java, and perform a cost and performance analysis of them on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM requirements. The analysis is conducted on the Sun SPOT platform, by Sun Microsystems, with a Squawk Virtual Machine.

# Preface

This thesis represents the finalization of our studies at the Norwegian University of Science and Technology (NTNU), Department of Telematics.

Mats would like to thank his family and friends for valuable support during the five years in Trondheim. A special gratitude goes to my better half, Ingrid Amalie Valen, for invaluable support.

Kim-André would also like to thank his family and good friends for a being such a fantastic support and giving me some great years in Trondheim.

Secondly, we want to thank our supervisor at NTNU, Danilo Gligoroski, for providing us with motivation and a belief that our research are of great importance for the community.

Thirdly, a special gratitude goes to our fellow students at our office ”F-257” for making the last year of our studies here at NTNU such a blast. Finally, thanks to everybody in the Communication Technology class of 2010, for five great years at NTNU. We made it!

Sincerely,

Mats Knutsen & Kim-André Martinsen

June 3, 2010

# Contents

# List of Figures

# List of Tables

# Acronyms

**AES** Advanced Encryption Standard

**ANSI** American National Standards Institute

**API** Application Programming Interface

**ARM** Advanced RISC Machine

**CPU** Central Processing Unit

**FIPS** Federal Information Processing Standard

**GUI** Graphical User Interface

**IDE** Integrated Development Environment

**JDK** Java Development Kit

**KAT** Known Answer Test

**kB** kilobytes

**MCK** Master Clock

**MD5** Message-Digest algorithm 5

**NIST** National Institute of Standards and Technology

**OS** Operating System

**RAM** Random-Access Memory

**RFID** Radio-Frequency Identification

**RISC** Reduced Instruction Set Computer

**ROM** Read-Only Memory

**SDK** Software Development Kit

**SHA** Secure Hash Algorithm

**SLCK** Slow Clock

**SPOT** Small Programmable Object Technology

**VM** Virtual Machine

# 1

# Introduction

The National Institute of Standards and Technology (NIST) is in the process of selecting a new cryptographic hash function through a public competition [3]. The winner(s) of this competition will be selected as the new Secure Hash Algorithm (SHA)-3 standard(s). At the First SHA-3 Candidate Conference several individuals stressed the need for supporting constrained platforms [3].

Hash functions may be designed for a specific purpose, such as message authentication, however in the context of becoming the SHA-3 algorithm(s) the candidate(s) need to fit a wider range of applications. While security is the highest criteria, performance and flexibility of the candidates are also important to emphasize. Since the new standard will need to operate on many platforms, research with the aim of producing measurements of the candidates on constrained platforms should be of interest for the comunity when evaluating the candidates. Fig. 1.1 depicts a typical purpose of a hash function.

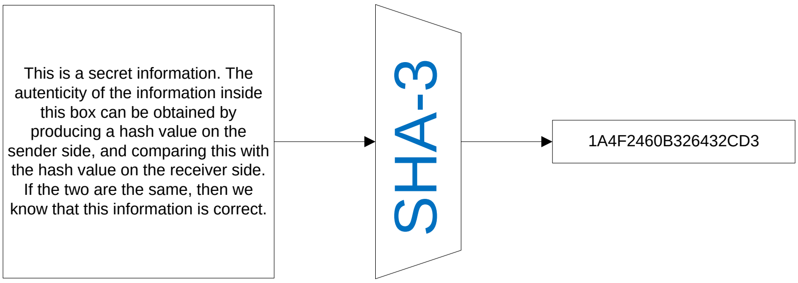

This is a secret information. The autenticity of the information inside this box can be obtained by producing a hash value on the sender side, and comparing this with the hash value on the receiver side. If the two are the same, then we know that this information is correct.
SHA-3
1A4F2460B326432CD3

Figure 1.1: A Typical Hash Function

The number of small entities connecting to the Internet of Things[1], is rapidly increasing. For instance, in 2008 5.085 billion smart cards shipped globally, while the estimates for 2010 are 5.455 billion smart cards [2]. Further, smart cards have a wide range of applications spanning from cryptography and health care to identification and authentication. Since next generation of smart cards running java are not currently available, the work conducted in this thesis will be carried out a Sun Small Programmable Object Technology (SPOT), running the same Virtual Machine (VM) and Central Processing Unit (CPU).

To our knowledge, no work has been demonstrated for comparing Java implementations of all the SHA-3 candidates on constrained platforms. In

[1] Internet of Things refers to the networked interconnection of everyday objects.

this thesis, we compare the implementation costs of the remaining SHA-3 candidates on a Sun SPOT running Java on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM costs. The device has many of the same properties as other constrained devices with regards to memory and computing efficiency. A further explanation is given in Sect. 2.5.

## 1.1 Method

The work in this thesis is conducted by the two authors in cooperation. The reference C implementation of the candidates is run in Microsoft Visual Studio, while the Java implementation is done using the Netbeans Integrated Development Environment (IDE). A debugger on both sides has been used to verify intermediate values, while implementing the algorithms.

Netbeans is further used to communicate with the Sun SPOT, with regards to implementing and getting feedback on the console when executing the hash function. Messages of different input lenghts have been tested, to give a bigger perspective as to how each candidate performs on a small device. The Java implementations are also being tested on a desktop computer as a reference.

## 1.2 Scope and Objectives

This thesis seek to answer the request for Java implementation on constrained devices. Measurements of Java implementations of all the 14 remaining candidates, run on a constrained device, will be produced. A summary of this work was submitted to the second SHA-3 hash conference that will held in Santa Barbara in period 23-24 August 2010. There exist no publicly available Java implementation of all candidates, so this work will hopefully give some input to the conference, as the candidates will be implemented in a new language on a new platform.

## 1.3 Outline

This work is outlined as follows:

**Chapter 2** presents the tools and background for this work.

**Chapter 3** describes the general design choices when implementing the candidate algorithms in Java.

**Chapter 4** gives an in-depth look at each candidate's implementation.

**Chapter 5** explaines how the measurements for performance and size was conducted. Further, our results from running the Java implementations on our constrained device are presented.

**Chapter 6** concludes this work, and proposes future work.

# 2

# Background Technologies

In this chapter cryptographic hash functions and selected hash constructions are presented. A summary of the National Institute of Standards and Technology (NIST) Cryptographic Hash Algorithm Competition will be given, before we describe the Advanced RISC Machine (ARM) processor architecture and the Sun SPOT, with the Squawk VM. Finally, the Netbeans development tool is presented.

## 2.1 Cryptographic Hash Functions

A hash function is a mathematical deterministic function that takes a binary input, referred to as the message *M*, and produces a fixed-size condensed version of the input message. The condensed version of the input message is referred to as the message digest *h*, and is the output of the function **H**. I.e. $h$=$\mathbf{H}(M)$ [42].

A *cryptographic* hash function is an algorithm where it is computational infeasible to find a matching input given you know the hashed output (the one-way property), or to find two inputs that maps to the same hash result(the collision-free property) [11, 47]. Tab. 2.1 depicts generally accepted requirements presented for a cryptographic hash function [47].

| **Requirement** | **Description** |
|---|---|
| Variable input size | H can be applied to a block of data of any size |
| Fixed output size | H produces a fixed-length output |
| Efficiency | H($x$) is relatively easy to compute for any given $x$, making both hardware and software implementations practical. |
| Preimage resistant(one-way property) | For any given hash value $h$, it is computationally infeasible to find $y$ such that H(y) = h. |
| Second preimage resistant(weak collision resistant) | For any given block $x$, it is computationally infeasible to find y ≠ x with H(y) = H(x). |
| Collision resistant(strong collision resistant) | It is computationally infeasible to find any pair (x,y) such that H(x) = H(y). |
| Pseudorandomness | Output of H meets standard tests for pseudorandomness. |

Table 2.1: Requirements for a Cryptographic Hash Function (taken from [47])

Cryptographic hash functions are versatile and are widely used in many applications such as password verification, message integrity, digital signatures, key derivation, message authentication codes and pseudorandom number generators. The various application purposes have different security requirements, and hence several algorithms may be better suited for a certain application [28].

The two most commonly used cryptographic hash functions are Message-Digest algorithm 5 (MD5) and SHA-1. However, MD5 is broken, and attacks can produce collisions with a complexity of $2^{10}$ [49]. In addition, SHA-1 was also successfully collision attacked with a complexity of $2^{63}$ [48]. NIST recognizes any good 160-bit hash function to have a lower limit of $2^{80}$ computations [28]. The weaknesses reported in the two most commonly used hash functions resulted in an initiative from NIST to develop a new hash function, this initiative is further discussed in Sect. 2.3.

## 2.2 Hash Function Constructions

Most of the hash functions in use today are designed as an iterative process, known as Merkle-Damgård construction. While there exist a wide range of designs, 50-60 known in 1993, atleast 75% of them were broken at that time [43]. Since 1993 another 30-40 designs have emerged, however most of them have been broken [44]. In the SHA-3 competition, numerous designs are represented, the following subsections will give an introduction to these.

### 2.2.1 Merkle-Damgård Construction

A great deal of cryptographic hash functions have an iterative design. Such a design hash data by iterating a basic compression function of subsequent blocks of data. Fig. 2.1 depicts an iterative design, known as Merkle-Damgård design. Here a message, $X$, is decomposed into $n$ blocks of data $x_1, ..., x_n$. The compression function, *f*, is then applied to each block and

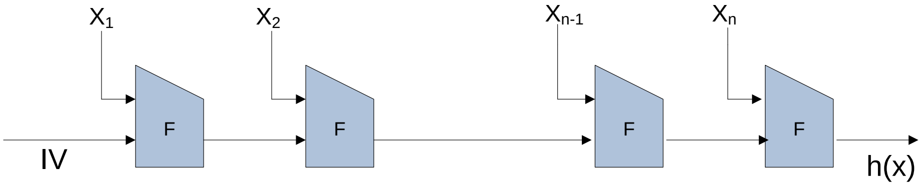


Figure 2.1: An Iterative Cryptographic Hash Function

also the result of the compression function of the previous block. The last compression step becomes the output *h(x)* [42].

In Merkle-Damgård constructions the compression function $f_{MD}$ is as follow: $f_{MD} : \{0,1\}^{m_c} \times \{0,1\}^n \rightarrow \{0,1\}^{m_c}$. Here, $m_c$ denotes the size of the chaining value, and $n$ denotes the block size for the compression function. Merkle-Damgård constructions use a padding called *Merkle-Damgård strengthening*. The messages are padded with a padding that encodes the length of the original message. The goal of this is to prevent collison and pseudo-collision attacks [42].

### 2.2.2 HAIFA Construction

The HAIFA[1] construction is quite similar to the Merkle-Damgård construction. However, it differs in the inputs added to the compression function. In a HAIFA compression function there are four inputs: the message, the initial value (IV), the number of bits hashed so far and the salt value [39]. The explicit use of input salt and the number of bits hashed so far to the compression function are quite distinguishing for this construction. The idea is that this input will alter the chaining values of each stage.

[1] The name HAIFA is taken from HAsh Iterative FrAmework

The compression function of HAIFA, *f*, is then proposed as follows. $f_{HAIFA}: \{0,1\}^{m_c} \times \{0,1\}^{n} \times \{0,1\}^{b} \times \{0,1\}^{s} \to \{0,1\}^{m_c}$. When compared with the Merkle-Damgård compression function presented in Sect. 2.2.1, the difference is the bits hashed so far, $b$, and the salt $s$. In cases where there is no need for adding salt, such as message authentication codes, the salt is set to 0 [39]. Fig. 2.2 depicts the HAIFA construction.

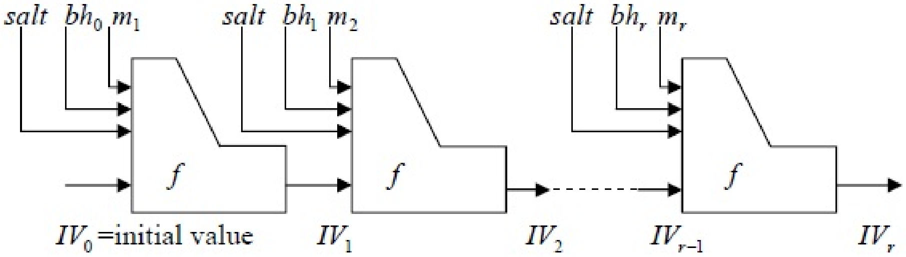


Figure 2.2: The HAIFA Construction (taken from [40])

### 2.2.3 Sponge Construction

Sponge constructions operate on states with *b*=*r*+*c* bits. Here $r$ is the *bitrate*, $b$ is *width* and $c$ represents the *capacity* [38]. Fig. 2.3 depicts the sponge construction.

The sponge construction starts with initializing all the bits in the state to zero. Then the input message is padded and split into blocks of $r$-bit length. After this the construction goes through two phases: the *absorbing phase* and the *squeezing phase* [38].

The absorbing phase *xors* the $r$-bit input message blocks into the first $r$-bits of the state, interleaved with applications of *f*. After all message blocks are processed, the absorb phase is ended and the construction switches to squeezing phase [38]. Fig. 2.3 depicts the absorbing phase

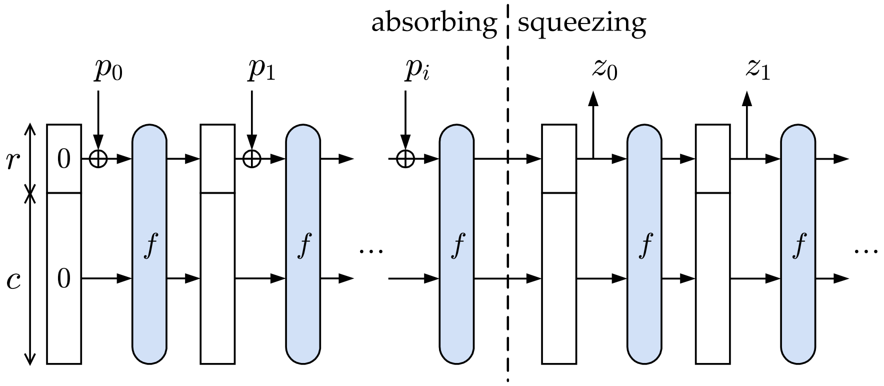


Figure 2.3: The Sponge Construction (taken from [38])

on the left-hand side, while the squeezing is on the right-hand side of the figure.

In the squeezing phase, the $r$ first bits of the state are used as output blocks, interleaved with the function $f$. Since the value of $b$ is greater than the value of $c$, the last $c$ bits of the state are never used for the output during the squeezing phase [38].

### 2.2.4 Wide-pipe and Double-pipe Construction

To make a hash function resistant against certain multi-collision-type attacks, a proposal to make the intermediate chaining values og Merkle-Damgård mode twice as long as the final hash value. This mode is known as the wide-pipe mode [33]. In wide-pipe constructions the size of the internal state of an $n$-bit hash function is increased to $w > n$ bit. While in the double-pipe design an internal state with size twice the hash size is maintained. In designs with a larger internal state, the idea is to improve protection against internal collision [40].

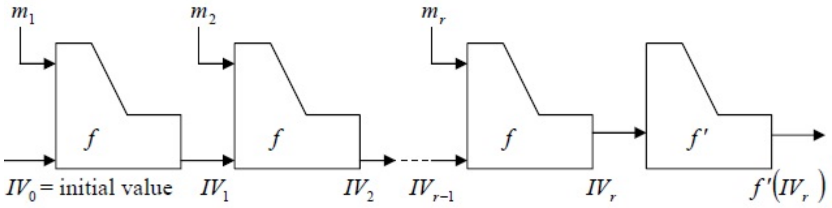


Figure 2.4: Wide-pipe Hash Constructions (taken from [40])

In the wide-pipe design, *two* compression functions are used, $f$ and *f'*. *f'* is invoked at the end of the computation. The compression functions are:

- $f$:$\{0,1\}^w \times \{0,1\}^m \rightarrow \{0,1\}^w$
- *f'*:$\{0,1\}^w \rightarrow \{0,1\}^n$

The input message M is divided into $r$-blocks, $M = m_1, m_2, .., m_r$. Fig. 2.4 depicts the process with the two compression functions. In the figure $IV_0$ is an initial value.

## 2.3 Cryptographic Hash Algorithm Competition

Feb. 2005 Prof. Xiaoyun Wang et al. announced a differential attack on the existing SHA-1 hash function. With the new discoveries, and later improvements to the attacks, Prof. Wang et al. claimed to have found a method to find collisions in the SHA-1 hash function with a time complexity of $2^{63}$ [48]. NIST recognizes any good 160-bit hash function to have a lower limit of $2^{80}$ computations to produce a hash collision, and hence the method proposed by Prof.Wang et al. represents a theoretical collision attack on SHA-1 [3, 28].

In response to the results presented by Prof.Wang et al. NIST opened a public competition to develop a new cryptographic hash algorithm. The winner(s) of the competition will be named *SHA*-3, and will complement the SHA-2 hash algorithms currently specified in Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standard [3]. The competition was opened November 7, 2007 and submissions to the competition were to be received by October 31, 2008. NIST further specified that the winning algorithm will be a publicly disclosed algorithm, available worldwide without royalties and intellectual properties. After submission deadline all submissions were made publicly available for review and comment [11]. Tab. 2.2 depicts the remaining timeline of the competition.

| **Year 4 (2010)** | |
|---|---|
| 2Q | Public comment period ends. |
| 2Q | Hold the Second Hash Function Candidate Conference. Discuss the analysis results on the submitted candidates.Submitters may identify possible improvements for their algorithms. |
| 3Q | Address the public comments on the submitted candidates; select the finalists. Prepare a report to explain the selection. Announce the finalists. Publish the selection report. |
| 4Q | Submitters of the finalist candidates announce any tweaks to their submissions. Final round begins. |
| **Year 5 (2011)** | |
| 4Q | Public comment period ends for the final round. |
| **Year 6 (2012)** | |
| 1Q | Host the Final Hash Function Candidate Conference. Submitters of the finalist algorithms discuss the comments on their submissions. |
| 2Q | Address public comments, and select the winner. Prepare a report to describe the final selection(s). Announce the new hash function(s). |
| 3Q | Draft the revised hash standard. Publish the draft standard for public comments. |
| 4Q | Public comment period ends. Address public comments. Send the proposed standard to the Secretary of Commerce for signature. |

Table 2.2: Remaining Tentative Timeline for the Hash Algorithm Competition (taken from [11])

NIST proposed three categories of evaluation criteria that will be used to measure the submitted candidate algorithms against each other. The criterias are security, cost and performance, and algorithm and implementation characteristics [1].

Security of the algorithm was identified as the most important factor when evaluating the candidates. In [1] NIST identifies a number of well-defined security properties that is expected of the winning candidate. This thesis will not go into further details of the security of the remaining SHA-3 candidates, as that is not within the scope of our research.

In [1] cost and performance were identified as the second-most important criterion upon evaluating the various candidates. Within the context of this competition, cost includes computational efficiency and memory requirements [1]. Computational efficiency refers to the speed of an algorithm. And as NIST states in [3]: *”NIST expects SHA-3 to offer improved performance over the SHA-2 family of hash algorithms at a given security strength”*. In the case of memory requirements, both code size and Random-Access Memory (RAM) are of interest. Further, NIST identifies cost as a particular concern for constrained platforms, and several remarks regarding the need for supporting constrained platforms such as mobile phones, smart cards and Radio-Frequency Identification (RFID) systems were made at the First SHA-3 Candidate Conference [3]. In constrained environments the computational power and RAM sizes are often the limiting factor, leading to resource consumption awareness for the candidate algorithms.

Regarding the third evaluation criteria, NIST states candidate algorithms with greater flexibility may be given preference over other algorithms. This means algorithms capable of running on a wide range of platforms, and algorithms that use parallelism or instruction set extensions to achieve a higher performance [3]. In addition, a SHA-3 submission needs to support hash results of 224, 256, 348 and 512 bits. Designers should present a full and detailed design documentation, which includes a reference implementation and optimized versions for 32-bit and 64-bit machines.

At the submission deadline, a total number of 64 entries were received by NIST, however only 51 met the minimum requirements and were announced as first round candidates. On July 24, 2009, 14 second round candidates were announced. The 14 candidates advancing to round 2 are depicted in Tab. 2.3 below [3].

| **Algorithm Name** | |
|---|---|
| BLAKE | JH |
| Blue Midnight Wish | Keccak |
| CubeHash | Luffa |
| ECHO | Shabal |
| Fugue | SHAvite-3 |
| Grøstl | SIMD |
| Hamsi | Skein |

Table 2.3: Round 2 Candidates (taken from [11])

As described in Tab. 2.2 the 14 round 2 candidates will go through extensive review before a final round heat of candidates will be selected. The final selection(s) and the announcement of the new hash function(s) will take place in second quarter of 2012 [11].

## 2.4 ARM Processor Architecture

The ARM is a 32-bit Reduced Instruction Set Computer (RISC) architecture. ARM architectures incorporate typical RISC features such as a large uniform register file, load/store architecture and simple addressing modes [6].

Originally intended for desktop computers, the simplicity of the ARM made it suitable for constrained devices, and today it is the dominant chip used in mobile phones and handheld devices, such as digital media and music players. The ARM architecture is licensable, companies holding or formerly holding a ARM licensee include among others Apple Inc, Atmel, NVIDIA and Samsung [5]. As of 2007, about 98% of all mobile phones sold each year contains at least one ARM processor [7].

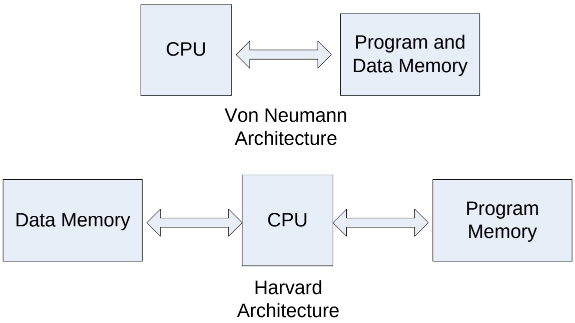


Figure 2.5: Harvard Architecture vs von Neumann Architecture

### 2.4.1 ARM920T Core

With the ARM9 family, ARM design moved from the von Neumann architecture to the Harvard architecture. The Harvard architecture is an architecture with physically separate storage and buses for instructions and data. In contrast, the von Neumann architecture uses the same bus for both data and instructions. With the Harvard architecture, a CPU can both read an instruction and perform a memory access at the same time. In the von Neumann this has to be carried out in two operations. Fig. 2.5 depicts the difference in the von Neumann and the Harvard architecture.

## 2.5 Sun SPOT

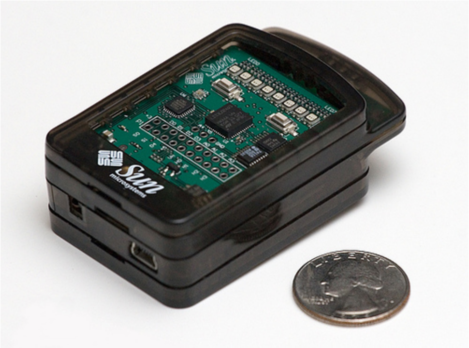


Figure 2.6: A Sun SPOT (taken from [36])

A Sun SPOT is a small embedded device developed at Sun Labs. The Sun SPOT is a programmable device originally developed for Wireless Sensor Network experimentation. A Sun SPOT is depicted in Fig. 2.6 [36].

The hardware platform residing in the currently available Sun SPOTs, the eSPOT main board, has a main processor running a Java VM named *Squawk*. Squawk is further described in Sec. 2.5.2. Below are the eSPOT main board specifications [35].

- ARM920T 180MHz Processor
- 4MB Flash memory
- 512KB pseudo-static RAM
- 2.4 GHz IEEE 802.15.4 radio with integrated antenna
- 2 AT91 Timer Counters
- USB interface

In addition to this the Sun SPOT also contains a sensor board with sensors for temperature, light, and a three-axis accelerometer. The sensor board also contains eight LED lights, I/O pins and analog inputs. The Sun SPOT is powered by a 3.7V rechargeable 750 mAh lithium-ion battery [35].

Applications running on the Sun SPOT are called MIDlets. A MIDlet is an application that runs on implementations of the Mobile Information Device Profile, one of the Java ME specifications. Due to its easily programmable interface, and variety of sensors, the Sun SPOT is applicable for a wide range of purposes.

### 2.5.1 AT91 Timer Counter

As mentioned, the Sun SPOT includes two AT91 Timer Counters that can be used in a variety of ways. Each of the Timer Counters includes three identical 16-bit channels. Of these six Timer Counter channels, four are available for SPOT applications, the two remaining are reserved for system use [37]. Each channel has five internal clock input signals which can be configured by the programmer. The Timer can count in various ways, and the rate is determined by which clock is used. The five internal clock inputs are connected to the Master Clock (MCK), the Slow Clock (SLCK) and to divisions of the MCK [37]. Tab. 2.4 depicts the available clock speeds.

| **TC Clock Input** | **Clock** | **Clock Speed (KHz)** | **Time for One Tick(usec)** | **Maximum Duration (msec)** |
|---|---|---|---|---|
| TC_CLKS_MCK2 | MCK/2 | 29,952 | 0.0334 | 2.188 |
| TC_CLKS_MCK8 | MCK/8 | 7,488 | 0.1335 | 8.752 |
| TC_CLKS_MCK32 | MCK/32 | 1,872 | 0.5342 | 35.009 |
| TC_CLKS_MCK128 | MCK/128 | 468 | 2.1368 | 140.034 |
| TC_CLKS_SLCK | SLCK | 32.768 | 30.5176 | 2,000.0 |

Table 2.4: Available Clock Speeds (taken from [37])

In this work, the clocks are used in *Capture Mode* to read the value of the Counter before and after the performed computation.

### 2.5.2 Squawk Virtual Machine

A VM takes the place of the Operating System (OS) for which the program would be tailored for ordinarily. Java is ported to the platform creating a layer isolating the application from the specifics of the underlying hardware and OS, and hence there are no need to develop separate versions of the application [41].

The Squawk VM is a small Java VM designed for constrained devices. Squawk runs without any OS [2, 45, 46]. Constrained devices allows the VM design to be simpler and more compact, providing the OS functionality in the VM [46]. Most VMs for Java are written in native languages such as *C* and *assembler*. Squawk utilizes a different approach, as it is written almost entirely in Java. Implementing Squawk in Java provides ease of portability, and a seamless integration of the VM and the application resources such as threads and objects [45, 46]. Fig. 2.7 depicts the Squawk VM and the corresponding standard Java VM [34].

As can be seen from the figure, only the *I/O Library* and *Native Code* are not written in Java. Most VMs assume there is an OS running on the device. However, Squawk allows applications to run on the *bare metal* – directly on the CPU without the usage of an OS. This results in a lower amount of overhead and a better control of the device [34]. Hence, resource-constrained devices running Squawk will not require additional resources to support the execution and maintenance of an OS [34].

Further, the Squawk VM allows the developers to run multiple applications on one instance of the VM. Squawk also allows an application

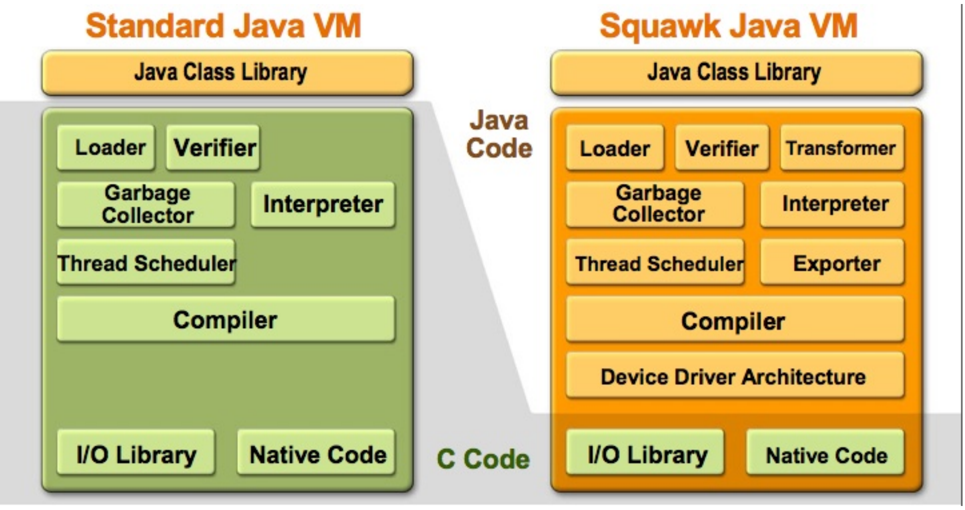


Figure 2.7: Standard Java VM vs Squawk Java VM (taken from [34])

to migrate a running device from one device to another. The migration actually allows the application to continue running from the point it was before the migration. With the ability to run multiple applications on the same VM instance, and by utilizing a more compact class file representation, the Squawk VM makes better use of the resources on a constrained device [34].

Fig. 2.8 depicts the *Split VM* architecture utilized by Squawk [34]. The split VM is a way to reduce the amount of work done by the devices. All the resource-consuming work, needed to perform class loading and optimizing, is done on the workstation while the execution is done on the device [34].

The size of the Java class files has previously been known as an issue for constrained devices [45]. To deal with this, Squawk includes a mechanism for serializing a graph of objects. This serialization of the objects becomes a collection of internal classes encapsulated in an object, referred

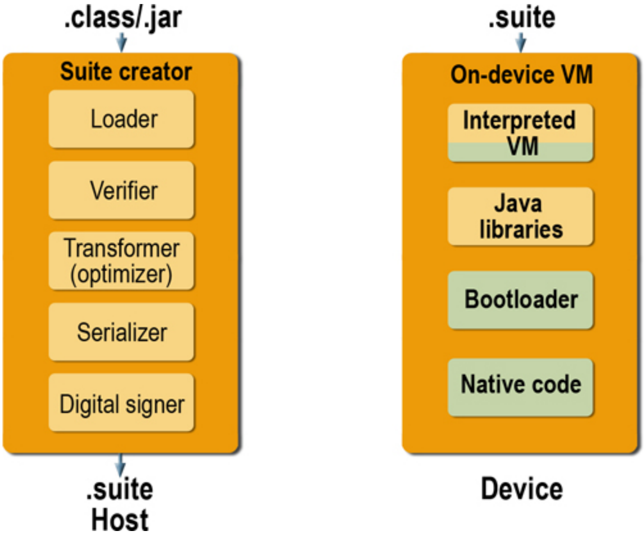


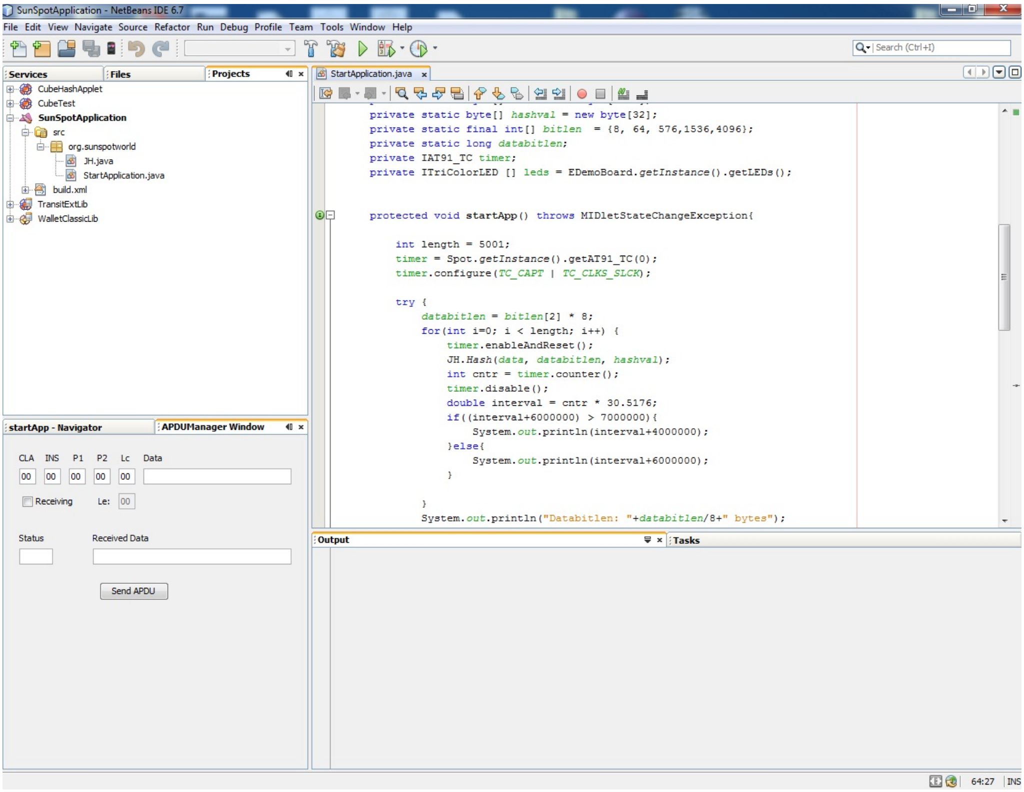
Figure 2.8: The *"Split VM"* Architecture (taken from [34])

to as a *suite* [46]. The suites are ready to use as-is and need no more transformation done by the device [34]. The design of the suite file was developed to be read serially. In addition the information in the suite file is in the best order for installation using as little RAM as possible [45]. On average, suite files are 35% the size of a corresponding class file [46].

## 2.6 Netbeans

Netbeans is an open-source IDE which supports development of all Java application types, as well as a wide range of other languages [27]. It is written in Java and may be used everywhere a Java VM is running. For development functionality with Java, a Java Development Kit (JDK) is required.

Netbeans was started as a Java IDE student project at Charles Uni-



```
    private static byte[] hashval = new byte[32];
    private static final int[] bitlen  = {8, 64, 576,1536,4096};
    private static long databitlen;
    private IAT91_TC timer;
    private ITriColorLED [] leds = EDemoBoard.getInstance().getLEDs();


    protected void startApp() throws MIDletStateChangeException{

        int length = 5001;
        timer = Spot.getInstance().getAT91_TC(0);
        timer.configure(TC_CAPT | TC_CLKS_SLCK);

        try {
            databitlen = bitlen[2] * 8;
            for(int i=0; i < length; i++) {
                timer.enableAndReset();
                JH.Hash(data, databitlen, hashval);
                int cntr = timer.counter();
                timer.disable();
                double interval = cntr * 30.5176;
                if((interval+6000000) > 7000000){
                    System.out.println(interval+4000000);
                }else{
                    System.out.println(interval+6000000);
                }


            }
            System.out.println("Databitlen: "+databitlen/8+" bytes");
```

Figure 2.9: The Netbeans Graphical User Interface

versity in Prague. In 1999 it was bought by Sun Microsystems and made open source the following year. Current version of Netbeans is 6.8, released December 10th 2010. Development of Netbeans has grown substantially since Sun Microsystems aquired the control of the project, also [10].

The use of a set of modular software components, called *modules*, allows applications to be developed. The work conducted in this thesis is carried out with the *Sun SPOT Plugin for Netbeans* [22]. This module allows the creation of a new SPOT project, containing all the files needed for a new Sun SPOT application, in NetBeans. Fig. 2.9 depicts the Graphical

User Interface (GUI) presented to the user when developing in Netbeans IDE.

# 3

# Java Implementation

The submitted candidates are written in the C programming language. To enable them to run on a Java platform, they have to be implemented in the Java language based on their current implementation. In this chapter we present some differences between the two programming languages, Java and C. Further, the inferface for our implementations are presented and our design choices are summarized. Finally we conclude with an application for verifying the correctness of our Java implementations.

## 3.1 Programming Java

In contrast to a low-level language, like C, the memory management in Java is not handled by the programmer. This makes it difficult to translate an optimized C-version of the candidates, as the two languages can differ in many ways. For instance, in C the *union* is a value that may have several representations in various formats. The primary usefulness of a union is to conserve space, since it provides a way of letting many different types be stored in the same space. The Java language guarantees type safety, so each member of a *union* has to be implemented as single variables which have to be synchronized when one of them are alternated. Other important differences between the two languages are summarized in Tab. 3.1 [14].

| **Feature** | **C** | **Java** |
|---|---|---|
| *Paradigms* | Function Oriented | Object Oriented |
| *Compilation* | Native machine code | Java VM byte code |
| *Memory Management* | Manual | Managed, using Garbage Collector |
| *Pointers* | Yes | No |
| *Bound Check Arrays* | No | Yes |
| *Complex Data Types* | Structures and Unions | Classes |
| *Unsigned Integers* | Yes | No |
| *Macros* | Yes | No |

Table 3.1: Programming C vs Java

Java compiles the code into byte code which can be run on several architectures. This is due to the fact that the code executes on a virtual machine like the Squawk VM described in Sect. 2.5.2. This makes software written in Java platform independent, and hard to optimize for a given platform. Characteristics like endianness do not have to be taken into consideration, as this is handled by the virtual machine.

## 3.2 Design Choices

The Java implementations of the candidates are based on the reference implementation representing the candidate in the second round. During the work of this thesis we have contacted several cryptographers and received different opinions about our strategy to use reference C implementations as a starting point for our Java implementation (not the optimized C codes). The main reason for the decision we made, is because the complexity of optimized C code does not allow easy (if at all possible) direct Java translation. The rest of this section will explain the general design choices we made for all candidates.

### 3.2.1 Message Digest

As the goal is to measure the algorithms on a small constrained device, we have only implemented the parts of each algorithm required to output a 256-bit message digest. Parts of the reference implementation which is not needed for this purpose is discarded. External functionality like salting and keyed hashing modes have not been implemented.

### 3.2.2 Input Length

The purpose of this work is to test the candidates on a constrained device. As such devices hold a limited amount of memory, we did not implement the possibility for hashing extremely long messages. This means removing the possibility for calling the *Update()* function several times.

Hashing extremely long messages would be very time consuming on a small device. The nature of such a device indicates that only relatively small amount of data will be hashed.

### 3.2.3 Fairness

Some of the reference implementations can be optimized, but the goal of this work was to give a fair impression of the performance of all candidates in Java. The implementation is also done by the two authors of this thesis in cooperation. The reason for this is to make sure that all implementations follow the same pattern, and all candidates are treated equally.

### 3.2.4 Error Handling

While translating to Java, we removed all error messages and exception handling mechanisms. The Java implementation is originally meant to be used by us for testing purposes, and hence we did not see any reason to implement error checking or error handling mechanisms. This is due to the fact that we only wanted to keep the code necessary to perform the hashing operation.

### 3.2.5 Correctness

Finally, the algorithms have been tested to make sure that they provide a correct output up to an input of 4096 bit. We give no guarantee that the algorithms produce correct output after this bit length is exceeded. The procedure of this testing if further explained in Sect. 3.5.

## 3.3 Application Programming Interface

The Java implementation of the candidates have been implemented with the same Application Programming Interface (API), as the one provided by NIST for the C implementations [4]. The API consist of a few data definitions, one data structure, and four functions to compute hashes. The rest of this section will give a quick overview of the API, and how it was adapted to Java.

### 3.3.1 Data Definitions

In C, the following *typedef* is given by the API to hold both arrays for the resulting hash value, as well as holding the data input:

```
typedef unsigned char BitSequence;
```

Java has no clear equivalent for unsigned char's, and no possibilities to define your own primitive data types. Hence the *BitSequence typedef* is replaced by the primitive data type *byte[]* which is an array in java that is able to hold 8-bit characters. Java also requires that the length of this array is set upon initialization.

To hold the datalength of the message being hashed, NIST proposed the following *typedef*:

```
typedef unsigned long long DataLength;
```

Java has no clear equivalent here either, so the primitive data type *long* is used for the java implementations. This is a signed data type which is able to hold 64-bit of data.

### 3.3.2 Data Structure

NIST proposed a hash state structure to contain all fields required to hold the state of the candidate algorithm. The following is required to implement in C:

```
typedef struct {
//hashbitlen + algorithm specific parameters
} hashState;
```

A structure like this cannot be implemented in Java. As only one instance of each algorithm is needed for this thesis, all state information is saved in *static* variables. Constants which are final are declared with *final*.

### 3.3.3 Function Calls

The NIST API specifies four function calls. Three methods for the incremental hashing, and one to perform the full hashing. The first specified is the *Init()* method to initialize the state of the candidate. The method is defined with the given parameters listed below:

```
HashReturn Init(hashState *state, int hashbitlen);
```

In the Java implementation, the state does not need to be given to the *Init()* method as all state information is available in static variables. The *hashbitlen* input is also removed as we only implemented the 256-bit part of each candidate. Hence the Java version of the initialization is implemented without any inputs.

Next, the API specifies the *Update()* fuction to perform all of the candidates compression functions. The following is taken from the API:

```
HashReturn Update(hashState *state, const BitSequence *data,
DataLength databitlen);
```

The state of the algorithm is again removed from the input, as the needed variables are available due to the static declaration. The two other parts of the input are changed to a byte array for holding the *data*, and a 64-bit integer to hold the *databitlen* as explained in Sect. 3.3.1

The post processing and output filtering is performed by the *Final()* method. The API proposes the following:

```
HashReturn Final(hashState *state, BitSequence *hashval);
```

As with *Init()* and *Update()*, the state is removed from the input and a byte array is used to hold the *hashval* variable. The size of this array is set to 32-bytes as this thesis only focuses on 256-bit message digests.

Finally the *Hash()* method is specified. This calls *Init()*, *Update()* and *Final()* once, in that order, to perform the full hashing and provide the resulting hash. The only difference in Java, except the previously mentioned, is that this method has no return value. This means that all error checking/handling is removed from the Java implementation. The candidates are instead tested manually that they provide correct output. To summarize, all candidates follow this interface:

```
public class candidate {

 private static *someStateVariable*;

 public static void Init();
 public static void Update(byte[] data, long databitlen);
 public static void Final(byte[] hashval);

 public static void Hash(byte[] data, long databitlen, byte[]
     hashval) {
   Init();
   Update(data, databitlen);
   Final(hashval);
```

```
  }
}
```

## 3.4 Tools for Implementation

The reference compiler used by NIST is the the ANSI C compiler in the Microsoft Visual Studio 2005 Professional Edition. To comply with this, the reference implementation of each candidate was run in Microsoft Visual Studio at the same time as the algorithms were implemented in Java with Netbeans.

A debugger in both applications were used to go through the implementation step-by-step to make sure that the intermediate values were correct before continuing implementing new stages of the algorithm. Finally, the implementations were tested against the Known Answer Test (KAT) to ensure its correctness for inputs up to 4096-bit, and this is described in detail in the following section.

## 3.5 Verifying Correctness

In order to verify that the translated versions of the candidates produce a correct output, a simple application was designed. The complete code of the application, *CheckKAT.java*, is fully presented in Appendix A. A description of the application is given below.

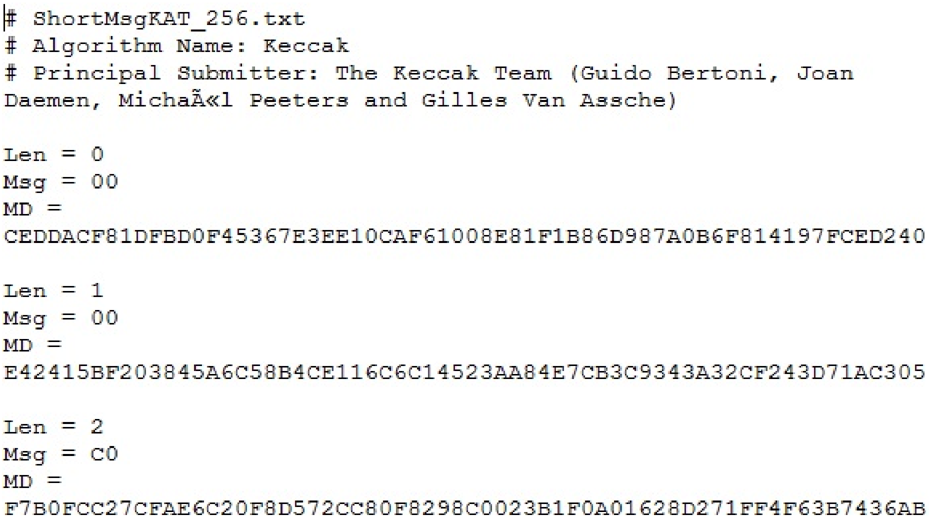
CheckKAT maintains an array with the Java implementations of the 14 candidates.

```
Class[] algorithms = { SHAvite3.class, Blake.class,
    BlueMidnightWish.class, Shabal.class, Skein.class, Luffa.
    class, Hamsi.class, Keccak.class, CubeHash.class, SIMD.
    class, Groestl.class, Echo.class, JH.class, Fugue.class };
```

This array is given as an input to the *start()* method, and processed in a *for* loop with the *performCheck()* method.

```
for(Class algorithm : algorithms){
    boolean result = performCheck(algorithm);
    System.out.println(result ? "100 Match": "";
}
```

In the *performCheck()* method, the submitted KAT files are read in and processed. Fig. 3.1 depicts an excerpt of the Keccak short message KAT file.



```
# ShortMsgKAT_256.txt
# Algorithm Name: Keccak
# Principal Submitter: The Keccak Team (Guido Bertoni, Joan
Daemen, MichaÃ«l Peeters and Gilles Van Assche)

Len = 0
Msg = 00
MD =
CEDDACF81DFBD0F45367E3EE10CAF61008E81F1B86D987A0B6F814197FCED240

Len = 1
Msg = 00
MD =
E42415BF203845A6C58B4CE116C6C14523AA84E7CB3C9343A32CF243D71AC305

Len = 2
Msg = C0
MD =
F7B0FCC27CFAE6C20F8D572CC80F8298C0023B1F0A01628D271FF4F63B7436AB
```

Figure 3.1: An Excerpt from the *ShortMsgKAT_256.txt* of Keccak

CheckKAT reads in the file, and stores the values of the *Len*, *Msg* and

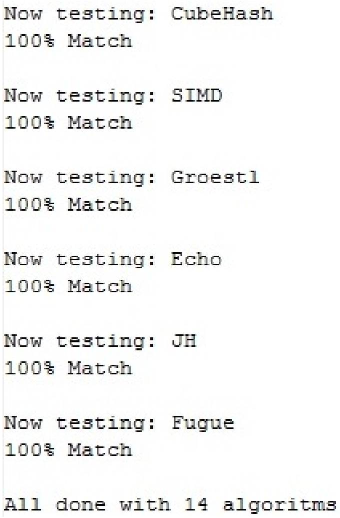
*MD* fields. Below is the code for storing the *Len* and *Msg* field.

```java
File file = new File(System.getProperty("user.dir") + "/shortmsgkat/"
        + clazz.getSimpleName() + "ShortMsgKAT_256.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(
        new FileInputStream(file)));
if (line.startsWith("Len")) {
    databitlen = Long.parseLong((line.split("= ")[1]));
}
if (line.startsWith("Msg")) {
    String value = null;
    int pointer = 0;
    String tmp = line.split("= ")[1];
    data = new byte[tmp.length() / 2];
    for (int i = 0; i < data.length; i++) {
        value = tmp.substring(pointer, pointer + 2);
        data[i] = (byte) Integer.parseInt(value.trim(), 16);
        pointer += 2;
    }
}
```

Once the fields are stored, they are sent as input to the corresponding hash function and the resulting message digest is compared to the *MD* field in the KAT file.

```java
method.invoke("foo", data, databitlen, hashval);
StringBuffer sb = new StringBuffer(hashval.length * 2);
for (int i = 0; i < 32; ++i) {
    sb.append(hexChar[(hashval[i] & 0xf0) >>> 4]);
    sb.append(hexChar[hashval[i] & 0x0f]);
}
if (!tmp.equals(sb.toString())) {
    error(databitlen, sb.toString(), tmp);
    return false;
}
```

In the code above, *tmp* contains the message digest from the KAT file, and *sb.toString()* is the computed message digest. If these are unequal, an error method is invoked. If not the application continues with reading and processing the fields.



```
Now testing: CubeHash
100% Match

Now testing: SIMD
100% Match

Now testing: Groestl
100% Match

Now testing: Echo
100% Match

Now testing: JH
100% Match

Now testing: Fugue
100% Match

All done with 14 algoritms
```

Figure 3.2: An Excerpt from the Output of CheckKAT.java

Fig. 3.2 depicts an excerpt from the output produced from running the CheckKAT application with all 14 candidates.

# 4

# Comments on the Candidates

This chapter will provide an in-depth comment on each of the candidates. Relevant technical aspects of the algorithms will be provided, in addition to our implementation of the compression functions. Examples from the reference C implementation will be compared to our Java equivalent code, to illustrate our design choices. Our subjective opinion will finally be given as to how we experienced the process with regards to ease of implementation.

## 4.1 BLAKE

BLAKE's iteration mode is based on a HAIFA construction. It's internal structure is a local wide-pipe, and the compression function is a modified version of the stream cipher ChaCha [8]. Fig. 4.1 illustrates the compression function, and the Java code for the same function is given below.

```
private static void compress(byte[] datablock, int pointer) {
  int offset = extraPadding ? pointer-- : pointer;

  m[0] = U8TO32_BE(datablock, 0 + offset);
  m[1] = U8TO32_BE(datablock, 4 + offset);
  m[2] = U8TO32_BE(datablock, 8 + offset);
  m[3] = U8TO32_BE(datablock, 12 + offset);
          (...)
  m[15] = U8TO32_BE(datablock, 60 + offset);

  v[0] = h32[0]; v[1] = h32[1];
  v[2] = h32[2]; v[3] = h32[3];
  v[4] = h32[4]; v[5] = h32[5];
  v[6] = h32[6]; v[7] = h32[7];
  v[8] = c32[0]; v[9] = c32[1];
  v[10] = c32[2];v[11] = c32[3];

  if (nullt != 0) {
    v[12] = c32[4]; v[13] = c32[5];
    v[14] = c32[6]; v[15] = c32[7];
  } else {
    v[12] = t32[0] ^ c32[4]; v[13] = t32[0] ^ c32[5];
    v[14] = t32[1] ^ c32[6]; v[15] = t32[1] ^ c32[7];
  }

  for (int round = 0; round < 10; ++round) {
    G32(0, 4, 8, 12, 0, round);  G32(1, 5, 9, 13, 1, round);
    G32(2, 6, 10, 14, 2, round); G32(3, 7, 11, 15, 3, round);
    G32(0, 5, 10, 15, 4, round); G32(1, 6, 11, 12, 5, round);
    G32(2, 7, 8, 13, 6, round);  G32(3, 4, 9, 14, 7, round);
  }

  h32[0] ^= v[0] ^ v[8];  h32[1] ^= v[1] ^ v[9];
  h32[2] ^= v[2] ^ v[10]; h32[3] ^= v[3] ^ v[11];
  h32[4] ^= v[4] ^ v[12]; h32[5] ^= v[5] ^ v[13];
  h32[6] ^= v[6] ^ v[14]; h32[7] ^= v[7] ^ v[15];
}
```

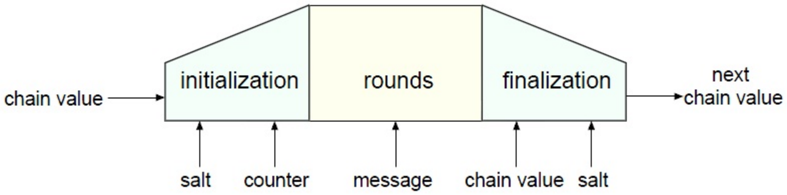


Figure 4.1: The local wide-pipe construction of BLAKE's compression function (taken from [8])

We implemented BLAKE-32 which compress message blocks of 512-bits, and the padding scheme is to append a '1' followed by '0's followed by a '1', followed by appending a 64-bit representation of the message [20].

BLAKE is relatively easy to implement in Java. It has some macros which are straight forward to implement as regular Java methods. An example is the macro *ROT32()* which looks like this in C [8]:

```
#define ROT32(x,n) ((x << (32-n))|( x >> n))
```

The Java equivalent of this macro is a method which takes the same input, and returns the same value as the macro body. As Java does not support unsigned integers, we also use an unsigned right shift operator for these operations in Java.

```
private static int ROT32(int x, int n) {
  return (x << 32 − n) | (x >>> n);
}
```

## 4.2 Blue Midnight Wish

Blue Midnight Wish has a double-pipe Merkle-Damgård design. It uses 16 32-bit variables as a double-pipe and it has three main functions [9]. The compression function is illustrated in Fig. 4.2. Our corresponding Java implementation is given below:

```
private static void Compression256(int[] M32, int[] H) {

  int i;
  int XL32, XH32;
  int[] Q = new int[32];
  int[] W = new int[32];

  W[0] = (M32[5] ^ H[5]) − (M32[7] ^ H[7]) + (M32[10] ^ H[10])
   + (M32[13] ^ H[13]) + (M32[14] ^ H[14]);
  W[1] = (M32[6] ^ H[6]) − (M32[8] ^ H[8]) + (M32[11] ^ H[11])
   + (M32[14] ^ H[14]) − (M32[15] ^ H[15]);
  W[2] = (M32[0] ^ H[0]) + (M32[7] ^ H[7]) + (M32[9] ^ H[9])
   − (M32[12] ^ H[12]) + (M32[15] ^ H[15]);
          (...)
  W[15] = (M32[12] ^ H[12]) − (M32[4] ^ H[4]) − (M32[6] ^ H[6])
   − (M32[9] ^ H[9]) + (M32[13] ^ H[13]);

  Q[0] = s32_0(W[0]) + H[1];
  Q[1] = s32_1(W[1]) + H[2];
  Q[2] = s32_2(W[2]) + H[3];
          (...)
  Q[15] = s32_0(W[15]) + H[0];

  for (i = 0; i < EXPAND_1_ROUNDS; i++)
    Q[i + 16] = expand32_1(i + 16, M32, H, Q);
  for (i = EXPAND_1_ROUNDS; i < EXPAND_1_ROUNDS +
    EXPAND_2_ROUNDS; i++)
    Q[i + 16] = expand32_2(i + 16, M32, H, Q);

  XL32 = Q[16] ^ Q[17] ^ Q[18] ^ Q[19] ^ Q[20]
   ^ Q[21] ^ Q[22] ^ Q[23];
  XH32 = XL32 ^ Q[24] ^ Q[25] ^ Q[26] ^ Q[27]
   ^ Q[28] ^ Q[29] ^ Q[30] ^ Q[31];

  H[0] = (shl(XH32, 5) ^ shr(Q[16], 5) ^ M32[0])
   + (XL32 ^ Q[24] ^ Q[0]);
            (...)
  H[3] = (shr(XH32, 1) ^ shl(Q[19], 5) ^ M32[3])
   + (XL32 ^ Q[27] ^ Q[3]);
            (...)
  H[15] = rotl32(H[3], 16) + (XH32 ^ Q[31] ^ M32[15]) + (shr(
      XL32, 2) ^ Q[22] ^ Q[15]);
}
```

Blue Midnight Wish compress message blocks of 512-bits, and the padding scheme is to append a '1' followed by '0's and finally the message length as an 64-bit integer. This is done to make sure that the output is a multiple of the input block size [20].

Blue Midnight Wish (BMW) is relatively harder to implement in Java compared with the other candidates. Inside the *Update()* function, the *M32* variable is an unsigned integer which share memory space with the data given as input [9]. Java ensures type safety, which means that an extra method has to be implemented in Java to synchronize our 8-bit *data* array with the 32-bit *M32()* array.

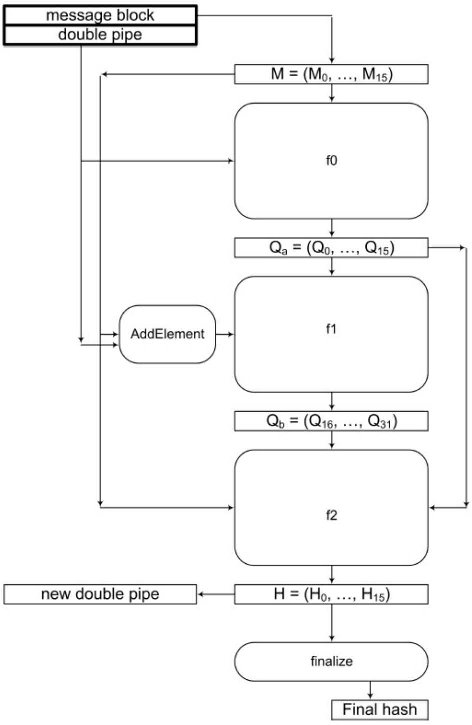


Figure 4.2: Compression function in Blue Midnight Wish (taken from [9])

```
private static void setM32(int[] M32, byte[] data, long databitlen) {
  int a = 0;
  try {
    for (int i = 0; i < M32.length; i++) {
      M32[i] = (data[a++] & 255) << 0;
      M32[i] |= (data[a++] & 255) << 8;
      M32[i] |= (data[a++] & 255) << 16;
      M32[i] |= (data[a++] & 255) << 24;
    }
  } catch (Exception e) {
  }
  M32Pointer = (int) Math.ceil((double) data.length / 4);
}
```

An extra integer is also implemented to work as a pointer for the *M32* array. Every time some of the data in M32 is consumed, the array pointer is set accordingly.

## 4.3 CubeHash

CubeHash is a sponge-like hash algorithm based on a fixed permutation [3]. CubeHash process message blocks by appending a '1', followed by as many '0's as required to get a multiple of the *b*-block size [20].

The *rounds()* function performs the permutation. Our implementation of the *rounds()* function is provided below, where we can see that the permutation uses additions, XORs and rotation.

```
private static void rounds() {
  int tmp = 0;
  for (int r = 0; r < 16; ++r) {
    for (int j = 0; j < 2; ++j)
      for (int k = 0; k < 2; ++k)
        for (int l = 0; l < 2; ++l)
          for (int m = 0; m < 2; ++m)
            x[1][j][k][l][m] += x[0][j][k][l][m];
    for (int j = 0; j < 2; ++j)
      for (int k = 0; k < 2; ++k)
        for (int l = 0; l < 2; ++l)
          for (int m = 0; m < 2; ++m)
            x[0][j][k][l][m] = rotateUpwards7(x[0][j][k][l][m])
                ;
    for (int k = 0; k < 2; ++k) {
      for (int l = 0; l < 2; ++l) {
        for (int m = 0; m < 2; ++m) {
```

```
        tmp = x[0][0][k][l][m];
        x[0][0][k][l][m] = x[0][1][k][l][m];
        x[0][1][k][l][m] = tmp;
      }
    }
  }
  for (int j = 0; j < 2; ++j)
    for (int k = 0; k < 2; ++k)
      for (int l = 0; l < 2; ++l)
        for (int m = 0; m < 2; ++m)
          x[0][j][k][l][m] ^= x[1][j][k][l][m];
  for (int j = 0; j < 2; ++j) {
    for (int k = 0; k < 2; ++k) {
      for (int m = 0; m < 2; ++m) {
        tmp = x[1][j][k][0][m];
        x[1][j][k][0][m] = x[1][j][k][1][m];
        x[1][j][k][1][m] = tmp;
      }
    }
  }
  for (int j = 0; j < 2; ++j)
    for (int k = 0; k < 2; ++k)
      for (int l = 0; l < 2; ++l)
        for (int m = 0; m < 2; ++m)
          x[1][j][k][l][m] += x[0][j][k][l][m];
  for (int j = 0; j < 2; ++j)
    for (int k = 0; k < 2; ++k)
      for (int l = 0; l < 2; ++l)
        for (int m = 0; m < 2; ++m)
          x[0][j][k][l][m] = rotateUpwards11(x[0][j][k][l][m]);
  for (int j = 0; j < 2; ++j) {
    for (int l = 0; l < 2; ++l) {
      for (int m = 0; m < 2; ++m) {
        tmp = x[0][j][0][l][m];
        x[0][j][0][l][m] = x[0][j][1][l][m];
        x[0][j][1][l][m] = tmp;
        }
      }
    }
    for (int j = 0; j < 2; ++j)
      for (int k = 0; k < 2; ++k)
        for (int l = 0; l < 2; ++l)
          for (int m = 0; m < 2; ++m)
            x[0][j][k][l][m] ^= x[1][j][k][l][m];
    for (int j = 0; j < 2; ++j) {
      for (int k = 0; k < 2; ++k) {
        for (int l = 0; l < 2; ++l) {
          tmp = x[1][j][k][l][0];
          x[1][j][k][l][0] = x[1][j][k][l][1];
          x[1][j][k][l][1] = tmp;
        }
      }
    }
  }
}
```

CubeHash is relatively easy to implement in Java [12]. No special considerations have to be taken, except from using the unsigned right shift operator for Java in both the *rotateUpwards7()* and *rotateUpwards11()* functions, as shown below. The Java implementation is very similar to the reference implementation.

```
private static int rotateUpwards7(int i) {
  return ((i << 7) | (i >>> 25));
}
private static int rotateUpwards11(int i) {
  return ((i << 11) | (i >>> 21));
  }
```

## 4.4 ECHO

ECHO follows the HAIFA construction, and is a wide-pipe hash algorithm [3]. The design embodies the goal of reusing, and thereby *echoing*, as many aspects of the Advanced Encryption Standard (AES) as possible [15]. The Java implementation of the *Compress()* function is provided as follows:

```java
private static void Compress() {
  Backup();
  counter_hi = messlenhi;
  counter_lo = messlenlo;
  for (int i = 0; i < rounds; i++) {
    BigSubWords();
    BigShiftRows();
    BigMixColumns();
  }
  BigFinal();
}
```

First off, the *BigSubWords()* transformation applies two AES rounds to each of the 16 words of the state. This is illustrated in Fig. 4.3. The Java code for this is:

```java
private static void BigSubWords() {
  k1[0][1] = (counter_hi >> 0);
  k1[1][1] = (counter_hi >> 8);
  k1[2][1] = (counter_hi >> 16);
  k1[3][1] = (counter_hi >> 24);

  for (int j = 0; j < 4; j++) {
    for (int i = 0; i < 4; i++) {
      k1[0][0] = (counter_lo >> 0);
      k1[1][0] = (counter_lo >> 8);
      k1[2][0] = (counter_lo >> 16);
      k1[3][0] = (counter_lo >> 24);

      aes(tab[i][j], k1);
      aes(tab[i][j], k2);

      counter_lo++;
      if (counter_lo == 0) {
        counter_hi++;
        k1[0][1] = (counter_hi >> 0);
        k1[1][1] = (counter_hi >> 8);
        k1[2][1] = (counter_hi >> 16);
        k1[3][1] = (counter_hi >> 24);

      }
    }
  }
}
```

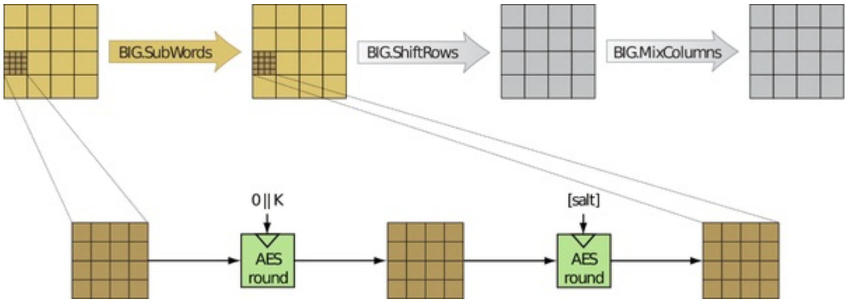


Figure 4.3: BigSubWords in Echo (taken from [16])

Below follows the *BigShiftRows()* transformation. This function mimics AES's *ShiftRows*, but on 128-bit words. This is illustrated in Fig. 4.4. The Java code for this is:

```
private static void BigShiftRows() {
  byte[][][] tmp = new byte[4][4][4];
  int m;

  for (int i = 1; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      m = (j + i) 4;for (int k = 0; k < 4; k++ {
        for (int l = 0; l < 4; l++) {
          tmp[j][k][l] = tab[i][m][k][l];
        }
      }
    }
    for (int j = 0; j < 4; j++) {
      for (int k = 0; k < 4; k++) {
        for (int l = 0; l < 4; l++) {
          tab[i][j][k][l] = tmp[j][k][l];
        }
      }
    }
  }
}
```

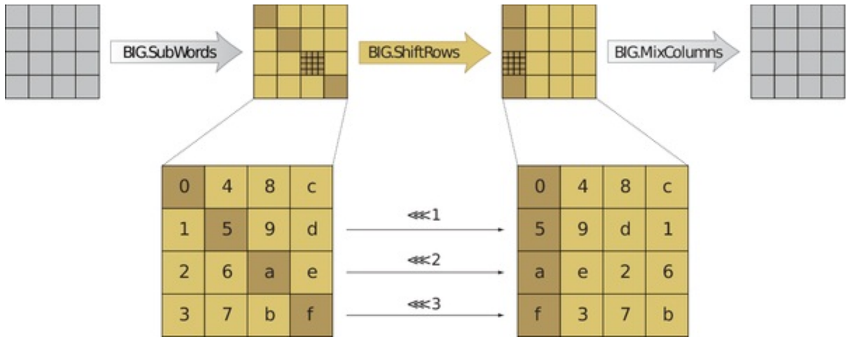


Figure 4.4: BigShiftWords in Echo (taken from [16])

The *BigMixColumns()* transformation applies AES's *MixColumns* to 4-tuples of bytes throughout the state. This is illustrated in Fig. 4.5. The Java code for this is:

```
private static void BigMixColumns() {
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      for (int k = 0; k < 4; k++) {
        NewMix4Bytes(i, j, k);
      }
    }
  }
}
```

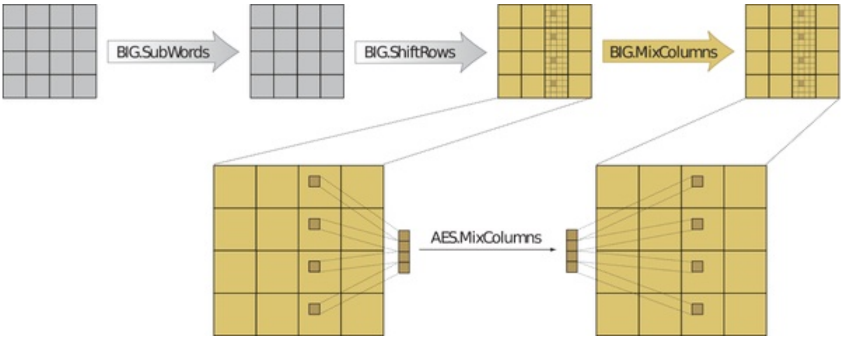


Figure 4.5: BigMixColums in Echo (taken from [16])

ECHO process message blocks of size 1536-bit. The padding is done by appending a '1' followed by '0's, followed by a 16-bit representation of the message digest size followed by a 128-bit representation of the message length such that the output is a multiple of the input block size [20].

ECHO is relatively harder to implement in Java, compared with the other candidates. The reference implementation in C uses an *Addresses* pointer array which points to positions in the four dimensional *tab* array [15].

```
for(j=0; j<4; j++)
   for(i=0; i<4; i++)
      for(l=0; l<4; l++)
         for(k=0; k<4; k++){
            state->tab[i][j][k][l] = 0;
            state->Addresses[m++] = & state->tab[i][j][k][l];
         }
```

In functions like Pop and Push, the algorithm uses the *Addresses* array to access the *tab* array in an easy way. The Pop function, in C, is as follows [15]:

```
word8 Pop(hashState *state) {
  return * state->Addresses[state->index++];
}
```

Our Java equivalent is a bit different as it has to access the tab array directly via the index variable:

```
private static int Pop() {
  int a = index / 64;
  int b = (index > 15) ? ((index / 16) MOD 4) : 0;
  int c = (index > 3) ? ((index / 4) MOD 4) : 0;
  int d = ((index - (index / 4)) + (index / 4)) MOD 4;
  index++;
  return tab[b][a][d][c];
}
```

This can cause some overhead with regards to performance, and requires some extra work when implementing ECHO in Java.

## 4.5 Fugue

Fugue maintains a large state, and is a variant of the sponge construction [17]. The padding scheme of Fugue is to pad with '0's to get a multiple of the block size, and then add an extra block containing the message length [20].

Fugue is one of the hardest candidates to implement in Java. It has a couple of unions like the following [17]:

```
typedef union {
  uint32    d;
  uint8     b[4];
}           hash32_s;
typedef hash32_s* hash32_p;

hash32_s    State[36];
```

The two variables, *d* and *b* in this case, are sharing memory space, and are often written to subsequently. In Java, the *State* variable has to be implemented as an inner class with *d* and *b* as its member variables. Every time one of them is written to, they have to be synchronized, before a read/write operation is conducted on the other. The Java class for the given union is:

```
private static class M {
  int d;
  byte[] b = new byte[4];

  private void syncFromByte() { ..code.. }
  private void syncFromInt()  { ..code.. }

}
```

This especially causes overhead in the *DoneFugue()* function, with a small code sample given below:

```
Col_Xor_RORn(p, p, 0, 0);
for (int i = 0; i < State.length; i++)
  State[i].syncFromInt();
Super_Mix();
for (int i = 0; i < State.length; i++)
  State[i].syncFromByte();
Col_Xor_RORn(p - 1, p + 1, 0, 0);
for (int i = 0; i < State.length; i++)
  State[i].syncFromInt();
Super_Mix();
for (int i = 0; i < State.length; i++)
  State[i].syncFromByte();
```

At the core of Fugue is a permutation called *SuperMix()*. It consists of a layer of AES S-box substitutions followed by the multiplication with a $16 \times 16$ constant $GF(2^8)$ matrix from the left [21]. A graphical representation of this function is included in Fig. 4.6 from the documentation of Fugue.

```
private static void Super_Mix() {
  M[] U = new M[4];
  for (int i = 0; i < U.length; i++)
    U[i] = new M();
  M D = new M();
  M[] W = new M[4];
  for (int i = 0; i < W.length; i++)
    W[i] = new M();
  int r, c;

  for (r = 0; r < 4; r++)
    for (c = 0; c < 4; c++)
      U[c].b[r] = aessub[BYTES(r, c) & 255];
  for (r = 0; r < 4; r++)
```

```
    for (c = 0; c < 4; c++)
      if (r != c)
        D.b[r] ^= U[c].b[r];

  for (r = 0; r < 4; r++)
    for (c = 0; c < 4; c++)
      W[c].b[r] = (byte) (gf2mul[U[c].b[0] & 255].b[m[0].b[r] & 255]
          ^ gf2mul[U[c].b[1] & 255].b[m[1].b[r] & 255]
          ^ gf2mul[U[c].b[2] & 255].b[m[2].b[r] & 255]
          ^ gf2mul[U[c].b[3] & 255].b[m[3].b[r] & 255] ^ gf2mul[D.b[r] & 255].b[m[r].b[c] & 255] & 255);

  for (r = 0; r < 4; r++)
    for (c = 0; c < 4; c++)
      State[COLUMN((c - r) & 3)].b[r] =W[c].b[r];
}
```

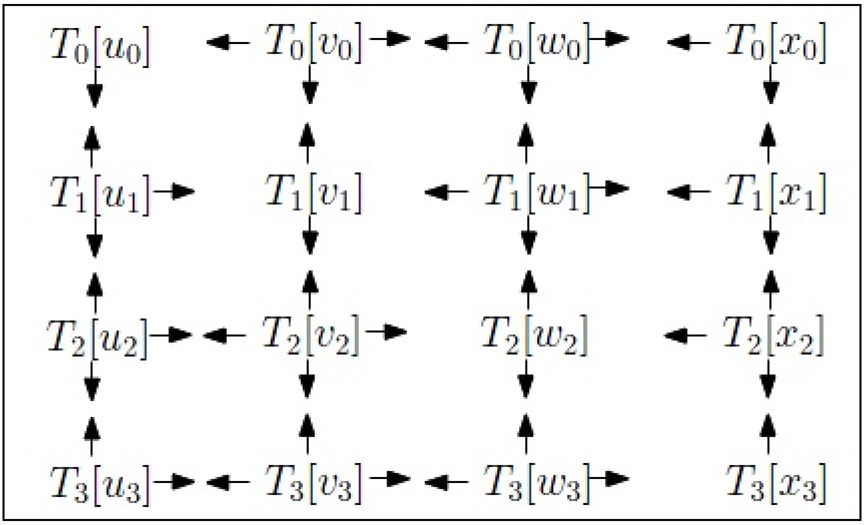


Figure 4.6: SMIX function of Fugue (taken from [17])

As mentioned, the hash state of the algorithm is relatively big and the reference implementation contains several *macros* and *inline* functions. With the extra inner classes, and small compensations which have to be made for *macros*, the Java implementation has a large potential for improvements with regards to both speed and size.

## 4.6 Grøstl

Grøstl is an iterated hash algorithm with a compression function built from two fixed, large, distinct permutations [18]. It is a wide-pipe Merkle-Damgård hash construction with post-processing [3]. The *Transform()* function of Grøstl is depicted in Fig. 4.7, with the Java implementation given below.

Our 256-bit implementation of Grøstl compress message blocks of 512-bits, and the padding scheme is to append a '1' followed by '0's, followed by the number of message blocks as an 64-bit integer such that the output is a multiple of the input block size [20].

```java
private static void Transform(byte[] input, int msglen, int
    offset) {

  int[][] temp1 = new int[8][16];
  int[][] temp2 = new int[8][16];

  for (; msglen >= 64; msglen -= 64, offset += 64) {
    for (int i = 0; i < 8; i++) {
      for (int j = 0; j < 8; j++) {
        temp1[i][j] = chaining[i][j]
            ^ input[(j * 8 + i) + offset];
        temp2[i][j] = input[(j * 8 + i) + offset];
      }
    }

    P(temp1);
    Q(temp2);

    for (int i = 0; i < 8; i++) {
      for (int j = 0; j < 8; j++) {
        chaining[i][j] ^= temp1[i][j] ^ temp2[i][j];
      }
    }

    block_counter++;
  }
}
```

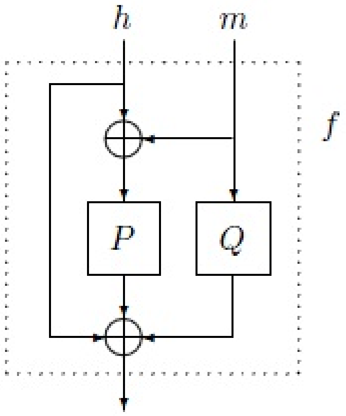


Figure 4.7: Compression function of Grøstl (taken from [18])

Grøstl is relatively easy to implement in Java.

## 4.7 Hamsi

Hamsi is relatively okay to implement in Java. Some *macros* and *inline* functions have been given some extra attention, but nothing that stands out compared to the other algorithms with the same functions.

On the other hand, Hamsi uses a three-dimensional array with the given dimensions: *T256[4][256][8]* [19]. Our constrained platform gives an error here, which says that this array exceeds the allowed constant size in Java. Hence, we were unable to compile this code for the constrained platform, and Hamsi is left out when measuring on the Sun SPOT. No further comment on the implementation will be given here due to this fact.

## 4.8 JH

JH uses a construction which is reminiscent of a sponge construction to build a hash algorithm out of a single, large, fixed permutation [3]. Our Java implementation of the bijective function *E8()* is provided below, while the illustration of this is function is depicted in Fig. 4.8.

```java
private static void E8() {
  int t0, t1, t2, t3;
  int[] tem = new int[256];

  for (int i = 0; i < 256; i++) {
    t0 = (H[i >> 3] >> (7 - (i & 7))) & 1;
    t1 = (H[(i + 256) >> 3] >> (7 - (i & 7))) & 1;
    t2 = (H[(i + 512) >> 3] >> (7 - (i & 7))) & 1;
    t3 = (H[(i + 768) >> 3] >> (7 - (i & 7))) & 1;
    tem[i] = (t0 << 3) | (t1 << 2) | (t2 << 1) | (t3 << 0);
  }
  for (int i = 0; i < 128; i++) {
    A[i << 1] = tem[i];
    A[(i << 1) + 1] = tem[i + 128];
  }
  for (int i = 0; i < 35; i++) {
    R8();
    update_roundconstant();
  }

  last_half_round_R8();

  for (int i = 0; i < 128; i++)
    H[i] = 0;
     for (int i = 0; i < 128; i++) {
       tem[i] = A[i << 1];
       tem[i + 128] = A[(i << 1) + 1];
  }
  for (int i = 0; i < 256; i++) {
    t0 = (tem[i] >> 3) & 1;
    t1 = (tem[i] >> 2) & 1;
    t2 = (tem[i] >> 1) & 1;
    t3 = (tem[i] >> 0) & 1;

    H[i >> 3] |= t0 << (7 - (i & 7));
    H[(i + 256) >> 3] |= t1 << (7 - (i & 7));
    H[(i + 512) >> 3] |= t2 << (7 - (i & 7));
    H[(i + 768) >> 3] |= t3 << (7 - (i & 7));
  }
}
```

The code for the function *E8()*, provided above, gives an example as to why the reference implementation of this algorithm will perform poorly. Most operations are done by iterating through arrays, and performing operations directly on the arrays. The reference implementation of the candidate has a lot of potential for improvements on performance. On the other hand it is relatively easy to implement in Java.

JH compress message blocks of 512-bits, and the padding scheme is to append a '1' followed by '0's followed by a 128-bit representation of the message length [20].

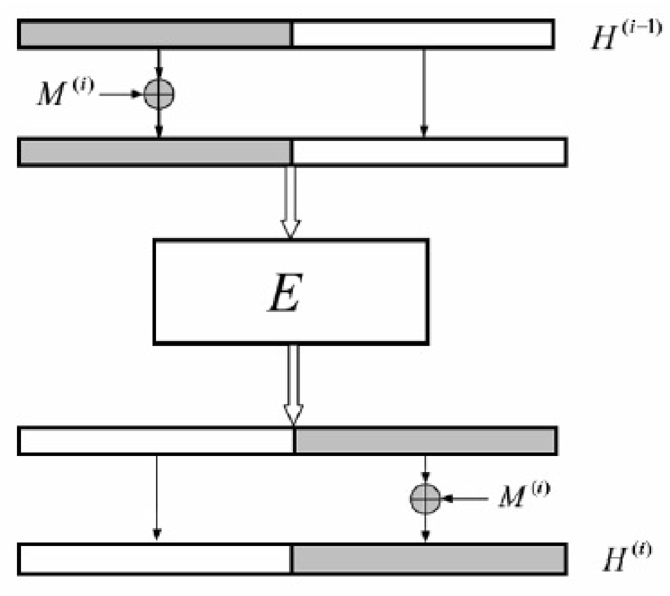


Figure 4.8: Compression function of JH (taken from [23])

## 4.9 Keccak

Keccak uses a large fixed permutation which can be seen as an SP-network, with 5-bit wide S-boxes, or as a combination of a linear mixing operation and a very simple nonlinear mixing operation [3]. The algorithm follows a sponge construction. The permutation function is implemented as:

```
private static void KeccakPermutationOnWords(long[] state) {
  for (int i = 0; i < 24; i++) {
    theta(state);
    rho(state);
    pi(state);
    chi(state);
    iota(state, i);
  }
}
```

Each of the functions in the permutation is implemented as follows:

```
private static void theta(long[] A) {

  long[] C = new long[5];
  long[] D = new long[5];

  for (int x = 0; x < 5; x++) {
    C[x] = 0;
    for (int y = 0; y < 5; y++)
      C[x] ^= A[index(x, y)];
      D[x] = ROL64(C[x], 1);
  }
  for (int x = 0; x < 5; x++)
    for (int y = 0; y < 5; y++)
      A[index(x, y)] \^= D[(x + 1) \ 5] Ĉ[(x + 4 \ 5];
```

```
private static void rho(long[] A) {
  for (int x = 0; x < 5; x++)
    for (int y = 0; y < 5; y++)
      A[index(x, y)] = ROL64(A[index(x, y)], KeccakRhoOffsets[
          index(x, y)]);
}
```

```
private static void pi(long[] A) {
  long[] tempA = new long[25];

  for (int x = 0; x < 5; x++)
    for (int y = 0; y < 5; y++)
      tempA[index(x, y)] = A[index(x, y)];
  for (int x = 0; x < 5; x++)
    for (int y = 0; y < 5; y++)
      A[index(0 * x + 1 * y, 2 * x + 3 * y)] = tempA[index(x, y
          )];
}
```

```
private static void chi(long[] A) {
  long[] C = new long[5];
  for (int y = 0; y < 5; y++) {
    for (int x = 0; x < 5; x++)
      C[x] = A[index(x, y)] ^ ((~A[index(x + 1, y)]) & A[index(x + 2, y)]);
    for (int x = 0; x < 5; x++)
      A[index(x, y)] = C[x];
  }
}
```

```
private static void iota(long[] A, int indexRound) {
  A[index(0, 0)] ^= KeccakRoundConstants[indexRound];
}
```

The padding scheme of Keccak is to append a ’1’ followed by ’0’s to a multiple of ’8’, followed by a digest specific constant, followed by a ’1’ and ’0’s to a multiple of the input block size [20].

Keccak is relatively hard to implement in Java, based on its reference implementation in C. The code base of the reference implementation is quite big, so some time were spent to find which parts of the code were relevant, as both displaying intermediate values as well as endian specific code is removed from the Java implementation [24].

A couple of extra methods are included in this Java implementation for synchronizing two arrays of different bit-lengths. An example is shown in the code below where the *KeccakPermutation()* function takes in a 8-bit array, which is passed on to *KeccakPermutaionOnWord()* function shown above, as a 64-bit array:

```java
private static void KeccakPermutation(byte[] state) {
  long[] stateLong = new long[state.length / 8];
  stateToLong(stateLong);
  KeccakPermutationOnWords(stateLong);
  stateFromLong(stateLong);
}
```

This causes some overhead to the java implementation, but it is still quite similar to the reference implementation.

## 4.10 Luffa

Luffa is a variant of the sponge construction, using a linear mixing operation and several fixed 256-bit permutations in place of a single wider permutation [3]. The Java implementation of the round function is given below as well as the code for the *mi()* and *tweak()* function. The same round function is depicted in Fig. 4.9.

```java
private static void rnd(int[] c) {
  mi();
  tweak();
  for (int i = 0; i < 8; i++) {
    step(c);
  }
}
```

```java
private static void mi() {
  int[] t = new int[40];
  for (int i = 0; i < 8; i++) {
    t[i] = 0;
    for (int j = 0; j < 3; j++) {
      t[i] ^= chainv[i + 8 * j];
    }
  }
  mult2(t, 0, 0);
  for (int j = 0; j < 3; j++) {
    for (int i = 0; i < 8; i++) {
      chainv[i + 8 * j] ^= t[i];
    }
  }
  for (int j = 0; j < 3; j++) {
    for (int i = 0; i < 8; i++) {
      chainv[i + 8 * j] ^= buffer[i];
    }
    mult2(buffer, 0, 1);
  }
  return;
}
```

```
private static void tweak() {
  for (int j = 0; j < 3; j++) {
    for (int i = 4; i < 8; i++) {
      chainv[(8 * j) + i] = (chainv[(8 * j) + i] << j) | (
          chainv[(8 * j) + i] >>> (32 - j));
    }
  }
  return;
}
```

Luffa is relatively easy to implement in Java. A couple of extra methods have to be implemented for synchronizing an 8-bit array with a 32-bit array. The methods *syncBuffer()* and *setByteBuffer()* takes an 8-bit array *p* as input, and synchronizes it with the static 32-bit *buffer* array [25].

Luffa compress message blocks of 256-bits, and the padding scheme is to append a ’1’ followed by as many ’0’s as required to get a multiple of the input block size [20].

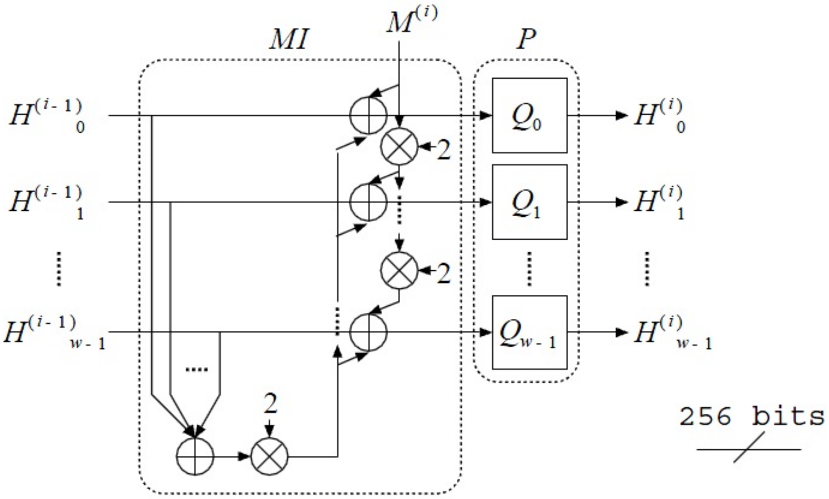


Figure 4.9: Round function of Luffa (taken from [25])

## 4.11 Shabal

Shabal is a hash algorithm that is constructed using chaining mode, which can be seen as a variant of a wide-pipe Merkle-Damgård hash construction [3]. Shabal processes message blocks with a size of 512-bits, and the padding scheme is to append a '1' followed by as many '0's as required to get a multiple of the input block size [20]. The graphical representation of the inner message rounds is depicted in Fig. 4.10. The following is our implementation of these rounds in Java:

```
input_block_add(m);
xor_counter();
apply_perm(m);
input_block_sub(m);
swap_bc();
```

```
private static void input_block_add(int[] m) {
  for (int i = 0; i < sM; i++) {
    B[i] = T32(B[i] + m[i]);
  }
}
```

```
private static void xor_counter() {
  A[0] ^= Wlow;
  A[1] ^= Whigh;
}
```

```
private static void apply_perm(int[] m) {
  int i, j;
  int xAm1, xA;
  for (i = 0; i < sM; i++) {
    int t;
    t = B[i];
    B[i] = (T32(t << 17) | (t >>> 15));
  }
  for (j = 0; j < 3; j++) {
    for (i = 0; i < sM; i++) {
      int tB;
      xA = xA(i, j);
      xAm1 = xAm1(i, j);
      xA = U(xA ^ V(T32(xAm1 << 15) | (xAm1 >>> 17)) ^ C[(8 +
          sM - i) MOD sM]) ^ B[(i + 13) MOD sM] ^ (B[(i + 9)
          MOD sM] & ~B[(i + 6) MOD sM]) ^ m[i];
      A[(i + sM * j) MOD nR] = xA;
      A[(i + sM * j + (nR - 1)) MOD nR] = xAm1;
      tB = B[i];
      B[i] = T32(((tB << 1) | (tB >>> 31)) ^ ~xA);
    }
  }
  for (j = 0; j < (3 * nR); j++) {
    A[(3 * nR - 1 - j) MOD nR] = T32(A[(3 * nR - 1 - j) MOD nR]
          + C[(3 * nR * sM + 6 - j) MOD sM]);
```

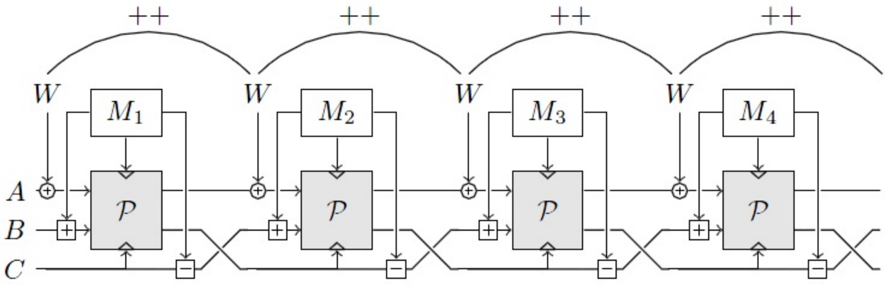


Figure 4.10: Message rounds of Shabal (taken from [29])

```
  }
}
```

```
private static void input_block_sub(int[] m) {
  for (int i = 0; i < sM; i++)
    C[i] = T32(C[i] - m[i]);
}
```

```
private static void swap_bc() {
  int t;
  for (int i = 0; i < sM; i++) {
    t = B[i];
    B[i] = C[i];
    C[i] = t;
  }
}
```

Shabal is relatively okay to implement in Java. Some extra code is provided in the reference implementation which is C specific. An example of this is [29]:

```
#ifdef ULLONG_MAX
typedef unsigned long long DataLength;
#else
typedef unsigned long DataLength;
#endif
```

There exists no equivalent for this in Java. Our strategy was to find the most convenient way to implement this, and since DataLenght is defined as a 64-bit long for all other candidates, we chose to keep it this way. An extra 32-bit integer is also introduced in the code, and is used as an array pointer in Java.

## 4.12 SHAvite-3

SHAvite-3 is a HAIFA hash algorithm, based on the AES building blocks. The compression function shown below is a keyed permutation that is used in the Davies-Meyer construction [3].

```
private static void Compress256(final byte[] message_block, int offset, byte[] chaining_value, long counter) {
  int[] pt = new int[8];
  int[] ct = new int[8];
  int[] msg_u32 = new int[16];
  int[] cnt = new int[2];
  for (int i = 0; i < 8; i++)
    pt[i] = U8TO32_LITTLE(chaining_value, 4 * i);
  for (int i = 0; i < 16; i++)
    msg_u32[i] = U8TO32_LITTLE(message_block, (4 * i) + offset);
  cnt[1] = (int) (counter >> 32);
  cnt[0] = (int) (counter & 0xFFFFFFFFL);
  E256(pt, ct, msg_u32, cnt);
```

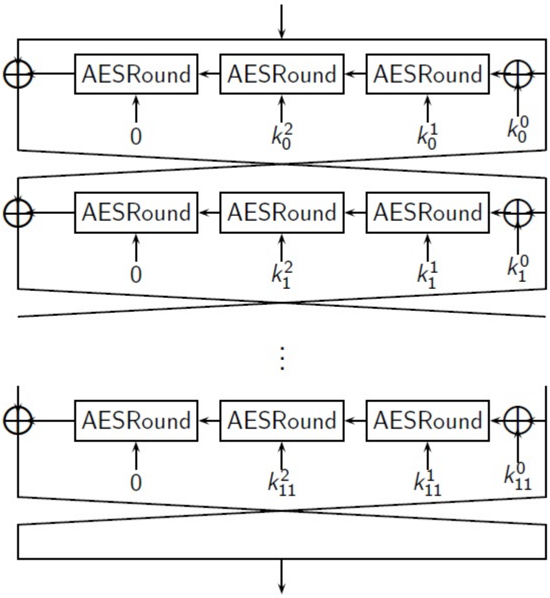


Figure 4.11: Rounds of E256 (taken from [30])

```
   for (int i = 0; i < 8; i++)
     pt[i] ^= ct[i];
   for (int i = 0; i < 8; i++)
     U32TO8_LITTLE(chaining_value, i * 4, pt[i]);
}
```

Our implementation of the *E256()* function is provided here, and illustrated in Fig. 4.11:

```
private static void E256(int[] pt, int[] ct, int[] message, int[] counter) {

   int[] state = new int[8];
   int[] input = new int[4];
   int[] output = new int[4];
   int[] rk = new int[144];
```

```
  for (int i = 0; i < 8; i++)
    state[i] = pt[i];
  for (int i = 0; i < 16; i++)
      rk[i] = message[i];
  MessageExpansion256(rk, counter);
  for (int i = 0; i < 12; i++) {
    for (int j = 0; j < 4; j++)
      input[j] = state[4 + j];
    roundAESkeyfirst(input, output, rk, 12 * i);
    for (int j = 0; j < 4; j++)
      input[j] = output[j];
    roundAESkeyfirst(input, output, rk, 12 * i + 4);
    for (int j = 0; j < 4; j++)
      input[j] = output[j];
    roundAESkeyfirst(input, output, rk, 12 * i + 8);
    for (int j = 0; j < 4; j++)
      state[j] ^= output[j];
    for (int j = 0; j < 4; j++) {
      int temp = state[j];
    state[j] = state[j + 4];
    state[j + 4] = temp;
  }
}
for (int i = 0; i < 8; i++)
  ct[i] = state[i];
}
```

SHAvite-3 process message blocks of size 512-bit. The padding scheme is done by appending a ’1’ followed by ’0’s, followed by the message length, followed by the digest length such that the output is a multiple of the input block size [20].

SHAvite-3 is relatively okay to implement in Java. The reference implementation explains very well each step necessary to perform the hashing.

A *portable.h* file is included with SHAvite-3 which contains several C specific mechanisms [30]. Some time were spent to find the correct definitions for use with Java, as some of them were endian-, processor- or application specific. The java implementation for the macro *U8TO32* is shown below:

```
private static int U8TO32_LITTLE(byte[] c, int offset) {
  int value;
  value = 255 & c[offset++];
  value |= (255 & c[offset++]) << 8;
  value |= (255 & c[offset++]) << 16;
  value |= (255 & c[offset++]) << 24;
  return value;
}
```

The reference implementation in C offered other ways to implement the same macro with regards to endianess. An offset is also added as array pointers are not supported in Java.

## 4.13 SIMD

SIMD is a wide-pipe Merkle-Damgård hash construction, where the compression function is built from a Feistel-like cipher in Davies-Meyer mode [31]. SIMD has an internal state which is twice as big as the output size. The compression function is implemented in Java as shown below, while Fig. 4.12 depicts it.

```
private static void SIMD_Compress(byte[] M, int finale) {
  int[][] W = new int[32][8];
  int[][] IV = new int[4][8];
  for (int i = 0; i < n_feistels; i++) {
    IV[0][i] = A[i];
    IV[1][i] = B[i];
    IV[2][i] = C[i];
    IV[3][i] = D[i];
  }
  message_expansion(W, M, finale);
  for (int j = 0; j < n_feistels; j++) {
    A[j] ^= PACK(M, 4 * j);
    B[j] ^= PACK(M, 4 * j + 4 * n_feistels);
    C[j] ^= PACK(M, 4 * j + 8 * n_feistels);
    D[j] ^= PACK(M, 4 * j + 12 * n_feistels);
  }

  Round(W, 0, 3, 23, 17, 27);
  Round(W, 1, 28, 19, 22, 7);
  Round(W, 2, 29, 9, 15, 5);
  Round(W, 3, 4, 13, 10, 25);

  Step(IV[0], 32, 4, 13, 0);
  Step(IV[1], 33, 13, 10, 0);
  Step(IV[2], 34, 10, 25, 0);
  Step(IV[3], 35, 25, 4, 0);

}
```

SIMD process message blocks of size 512-bit. The padding scheme is done by padding with ’0’s to get a multiple of the block size, and then add an extra block containing the message length. [20].

SIMD is relatively hard to implement in Java. SIMD uses an IV array of length 16 to store parts of the state information. The reference implementation uses four 32-bit pointers, to represent and operate on these 16 values. An example from the C-code is provided here [31]:

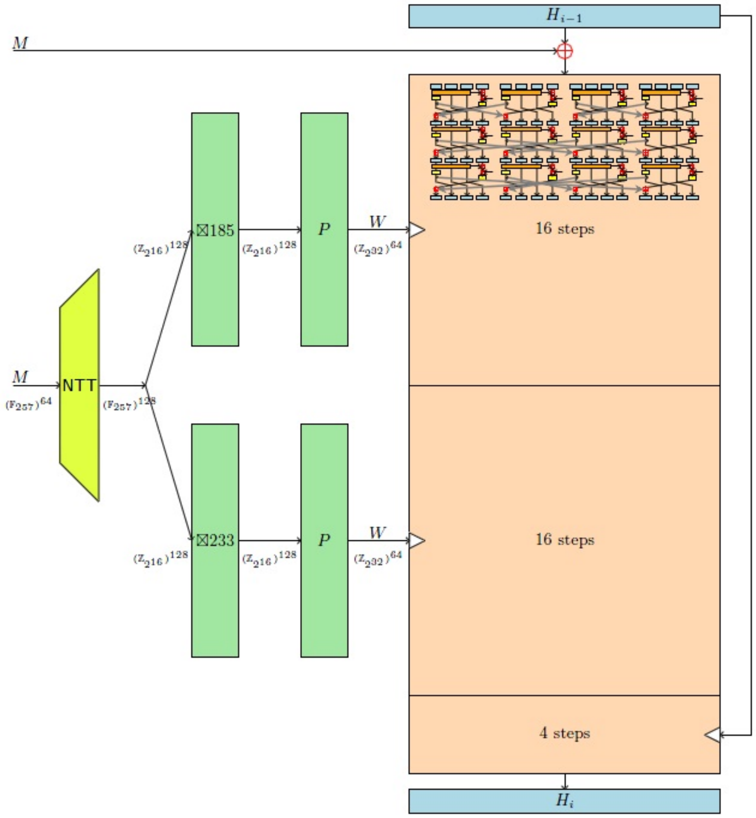

$H_{i-1}$
$M$
$M$
$(\mathbb{F}_{257})^{64}$
NTT
$(\mathbb{F}_{257})^{128}$
$(\mathbb{Z}_{2^{16}})^{128}$
⊠185
$(\mathbb{Z}_{2^{16}})^{128}$
$P$
$W$
$(\mathbb{Z}_{2^{32}})^{64}$
16 steps
$(\mathbb{Z}_{2^{16}})^{128}$
⊠233
$(\mathbb{Z}_{2^{16}})^{128}$
$P$
$W$
$(\mathbb{Z}_{2^{32}})^{64}$
16 steps
4 steps
$H_i$

Figure 4.12: Compression function of SIMD-256 (taken from [31])

```
u32 *A, *B, *C, *D;
n = 4;
state->A = malloc((4*n+4)*sizeof(u32));
state->A += ((u32*)NULL - state->A)&3;
state->B = state->A+n;
state->C = state->B+n;
state->D = state->C+n;

if (IV)
```

```
  memcpy(state->A, IV, 4*n*sizeof(u32));
else
  memset(state->A, 0, 4*n*sizeof(u32));
```

The way to solve this in Java was to make $A$, $B$, $C$ and $D$ integer arrays of length four. Instead of treating the four as pointers, they become regular integer arrays in our Java implementation:

```
private static int[] A = new int[4];
private static int[] B = new int[4];
private static int[] C = new int[4];
private static int[] D = new int[4];

System.arraycopy(IV, 0, A, 0, 4);
System.arraycopy(IV, 4, B, 0, 4);
System.arraycopy(IV, 8, C, 0, 4);
System.arraycopy(IV, 12, D, 0, 4);
```

This causes some small changes throughout the algorithm on how to read/write $A$, $B$, $C$ and $D$.

## 4.14 Skein

Skein is a variant of a Merkle-Damgård hash construction and consist of basically three components which is Threefish, Unique Block Iteration and Optional Argument System [32]. The final parts of the process block function using the Threefish-256 block cipher is provided below: Fig. 4.13 shows a summary of how Threefish works.

```
  private static void Skein_256_Process_Block(byte[] blkPtr, long blkCnt, int byteCntAdd) {
          (...)
      for (int i = 0; i < WCNT; i++) /* do the first full key injection */
```

```
        X[i] = (w[i] + ks[i]);

      X[WCNT - 3] += ts[0];
      X[WCNT - 2] += ts[1];

      for (int r = 1; r <= 72 / 8; r++) {
        X[0] += X[1]; X[1] = RotL_64(X[1], 14);
        X[1] ^= X[0]; X[2] += X[3];
        X[3] = RotL_64(X[3], 16); X[3] ^= X[2];

          (...)

        X[0] += X[3]; X[3] = RotL_64(X[3], 5);
        X[3] ^= X[0]; X[2] += X[1];
        X[1] = RotL_64(X[1], 37); X[1] ^= X[2];

        InjectKey(2 * r - 1, WCNT, X, ks, ts);

        X[0] += X[1]; X[1] = RotL_64(X[1], 25);
        X[1] ^= X[0]; X[2] += X[3];
        X[3] = RotL_64(X[3], 33); X[3] ^= X[2];

          (...)

        X[0] += X[3]; X[3] = RotL_64(X[3], 32);
        X[3] ^= X[0]; X[2] += X[1];
        X[1] = RotL_64(X[1], 32); X[1] ^= X[2];

        InjectKey(2 * r, WCNT, X, ks, ts);
      }

      for (int i = 0; i < WCNT; i++)
        ctxX[i] = (X[i] ^ w[i]);
      Skein_Clear_First_Flag();
      offset += 32;

    }
```

```

  }
```

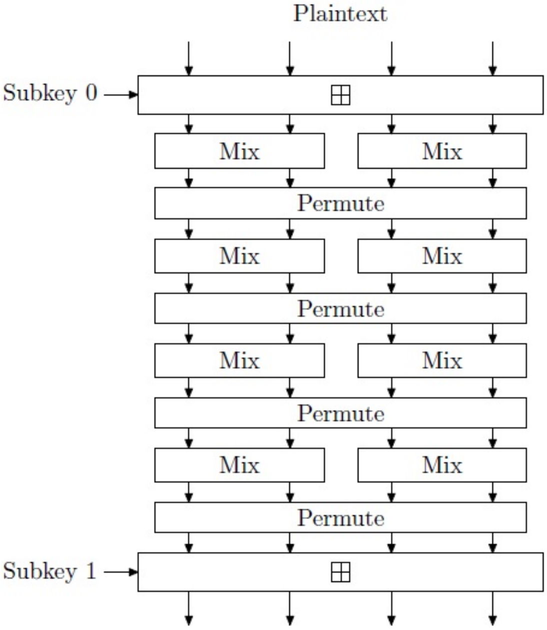


Figure 4.13: Four of the 72 rounds of the Threefish-256 block cipher (taken from [32])

Skein is relatively hard to implement in Java. Skein also uses a *union* like shown below [32]:

```
union {
  u08b_t   b[SKEIN_256_STATE_BYTES];
  u64b_t   w[SKEIN_256_STATE_WORDS];
} cfg;
```

This causes an extra method *syncFromLong()* to be created. This method only has to be invoked once, and does not cause much overhead.

Skein has a large code base, and uses a lot of macros. An example of a macro which is hard to implement in Java is the following [32]:

```
#define SKEIN_T1_BLK_TYPE(T)     (((u64b_t) (SKEIN_BLK_TYPE_##T))
     << SKEIN_T1_POS_BLK_TYPE)
```

This is a *macro* invoking another *macro*, depending on the input $T$. Some changes had to be made, for instance by calling the second *macro* directly. This causes some changes from reference implementation in C, to the java implementation. The padding scheme in Skein is done by padding with '0's to get a multiple of the input block size [20].

## 4.15 Summary

The Java implementations of the candidates are based on the reference implementations from the second round of the NIST competition. Only the parts of the algorithm needed to output a 256-bit message digest is implemented. The API provided by NIST for the C reference implementation is being used for all candidates, and this chapter has summarized the most important aspects of each candidate.

# 5

# Measurements and Results

This chapter presents an overview of our comparison of the remaining SHA-3 candidates, in terms of performance and cost. We present the target architectures and their spesifications. Further, a comment on measurements not conducted in this thesis is given and a quick introduction for measuring on constrained devices. The experiments conducted to perform measurements, and the format of the results are described. Finally, the results from the measurements are presented.

## 5.1 Target Architectures

The primary work of this thesis is to measure the performance of the candidates on a constrained device. In addition to this, the same code was measured on a desktop system for comparison purposes. The following systems were used:

- A Sun SPOT, running Squawk VM version 5.0 (red). The core CPU is a ARM920T core (ARMv4 architecture), clocked at 180MHz.

- A PC with an Intel Pentium 4, clocked at 3.0GHz. The OS is Windows 7, running in 32-bit mode. The system is running the Java VM, version 1.6.

## 5.2 Out of Scope Measurements

Currently available Java Cards, which are smart cards running Java technology, are running on 8-bit or 16-bit architectures with the Java Card 2 version. In contrast to Java Card 3, this version has very limited language support. For instance, features such as long and multi-dimensional arrays are not supported. This makes the process of implementing the C reference code to Java Card 2, a very large job. Despite the fact that Java Card 2 is widespread and currently used in many applications and smart cards, the work conducted in this thesis will not be suitable for Java Card 2.

## 5.3 Measurements on Constrained Devices

In constrained devices, power consumption and the area required to implement the algorithm are limiting factors [28]. Power consumption usually reduces to computational efficiency, which will be measured in *cycles/byte*. The size, in kilobytes (kB), of the compiled file loaded on the Squawk-capable constrained device is used as the measurement for Read-Only Memory (ROM) requirements of the algorithms.

## 5.4 Measuring Performance

The performance of the candidates were measured on the Sun SPOT. Applications running on SPOTs are Java applications, called MIDlets, as mentioned in Sect. 2.5. In order to make the application a MIDlet, the application must extend the *MIDlet* class, as shown with the code below. The complete code for *DeployOnSpot.java* can be found in Appendix B.

```
public class DeployOnSpot extends MIDlet
       implements TimerCounterBits{
```

In addition to extending *MIDlet* class, the TimerCounterBits interface is also implemented. This allows access to the various clocks, introduced in Sect. 2.5.1, used for timing the performance on the Sun SPOT. The choice of clock depends on the duration of the measurement. Some algorithms are fast and will only need the MCK, while others will need to be measured with the SLCK.

In order to perform the measurements the following code were executed on the Sun SPOT.

```
private static final int[] bitlen  = {8, 64, 576,1536,4096};
private static long databitlen;
private static byte[] data = new byte[4097];
private static byte[] hashval = new byte[32];
private IAT91_TC timer;
protected void startApp() throws MIDletStateChangeException{

        int length = 4500;
        timer = Spot.getInstance().getAT91_TC(0);
        timer.configure(TC_CAPT | TC_CLKS_SLCK);

        databitlen = bitlen[0] * 8;
        for(int i=0; i < length; i++) {
           timer.enableAndReset();
           CubeHash.Hash(data, databitlen, hashval);
           int cntr = timer.counter();
           timer.disable();
           double interval = cntr * 30.5176;
           System.out.println(interval);
        }
}
```

The code above displays the essential part of the MIDlet deployed on a Sun SPOT while measuring the performance of CubeHash, for an input of one byte. First, a number of variables are decleared. Arrays for message digest, the various bit lengths and the input data to the hash function. In addition a *IAT91_TC timer* is declared. The MIDlet has no main method, and *startApp()* is the invoked function upon execution once deployed to the Sun SPOT.

Inside the *startApp()* the following code is used to instantiate the timer, by choosing a Timer Counter.

```
timer = Spot.getInstance().getAT91_TC(0);
```

Further, the timer is configured with the preferred clock speed, in this case the SLCK, and set to *Capture Mode*.

```
timer.configure(TC_CAPT | TC_CLKS_SLCK);
```

The *databitlen* field is set to the correct byte size to be measured and a *for* loop is run 4500 times to measure the performance of the candidate. Inside the loop is where the results are generated. First, the timer is reset and started and then the specified algorithm, in this case CubeHash, is executed with the corresponding inputs. The timer is then read after execution, and the result stored in an *int*.

```
timer.enableAndReset();
CubeHash.Hash(data, databitlen, hashval);
int cntr = timer.counter();
timer.disable();
```

The number of ticks stored in *cntr* are then multiplied with the time for one tick, depicted in Tab. 2.4, and the time spent executing the *Hash()* function is printed.

```
double interval = cntr * 30.5176;
System.out.println(interval);
```

This loop is run 4500 times, for each of the input sizes and each of the candidates, and the results are loaded into a spreadsheet for processing. The results, depicted in Sect. 5.6, are presented as: lower-quartile, median and upper-quartile measured in Cycles/byte. The Cycles/byte calculation is performed with the following parameters: Time in seconds spent performing hash (Ts), frequency of the CPU in Hz(F) and message input length in bytes (L). The formula for calculating Cycles/byte for a candidates is:

$$Cycles/byte = \frac{Ts * F}{L}$$

## 5.5 Measuring Cost

Applications deployed onto the Squawk VM are first compiled to regular Java *class* files and then further to a *suite* file. As described in Sect. 2.5.2 Squawk VM includes a mechanism for serializing a graph of objects, these objects becomes a collection of internal classes encapsulated in the *suite.* As mentioned, a *suite* file is on average 35% the size of the corresponding class file [46].

As each application cannot be deployed to the device without extra application code, this work has only measured the size of the *class* file corresponding to each algorithm. This is the compiled code before it is compiled into a *suite* with the rest of the files needed.

## 5.6 Results

This section will present the results of the performance tests, as well as required ROM size for each candidate. Each candidate will be presented in alphabetical order. A simple graph for both the Intel platform and the Sun SPOT platform will be presented, depicting the cycles/byte for each of the given inputs explained in the preceeding sections. Finally this section will present a summary for each of the platform with all candidates. The tables will show median cycles/byte as well as upper and lower quartile. The tables are sorted by lowest to highest median times for each of the inputs.

Fig.5.1-13 depicts the performance of the 14 candidates, on both the Intel- and the Sun SPOT platform. Tab.5.1-3 depicts the results on the Intel, while Tab.5.4-6 displays the Sun SPOT results. Finally, Tab.5.7 is the ROM cost of the candidates.

## BLAKE

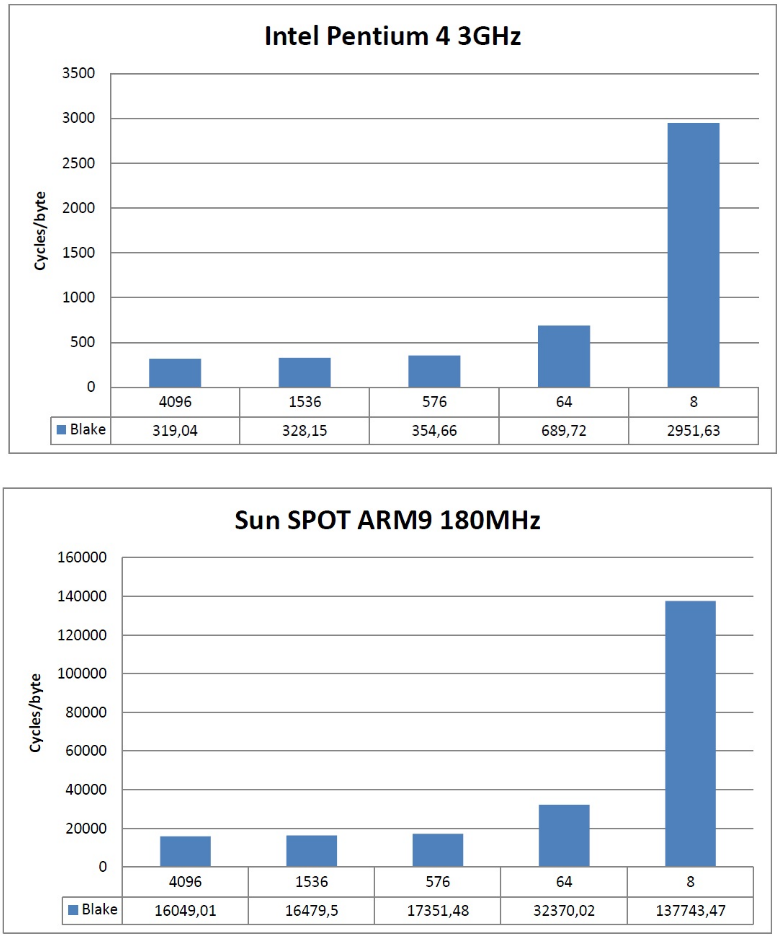


Figure 5.1: BLAKE Performance

*Required ROM size: 7.31 kB*

## Blue Midnight Wish

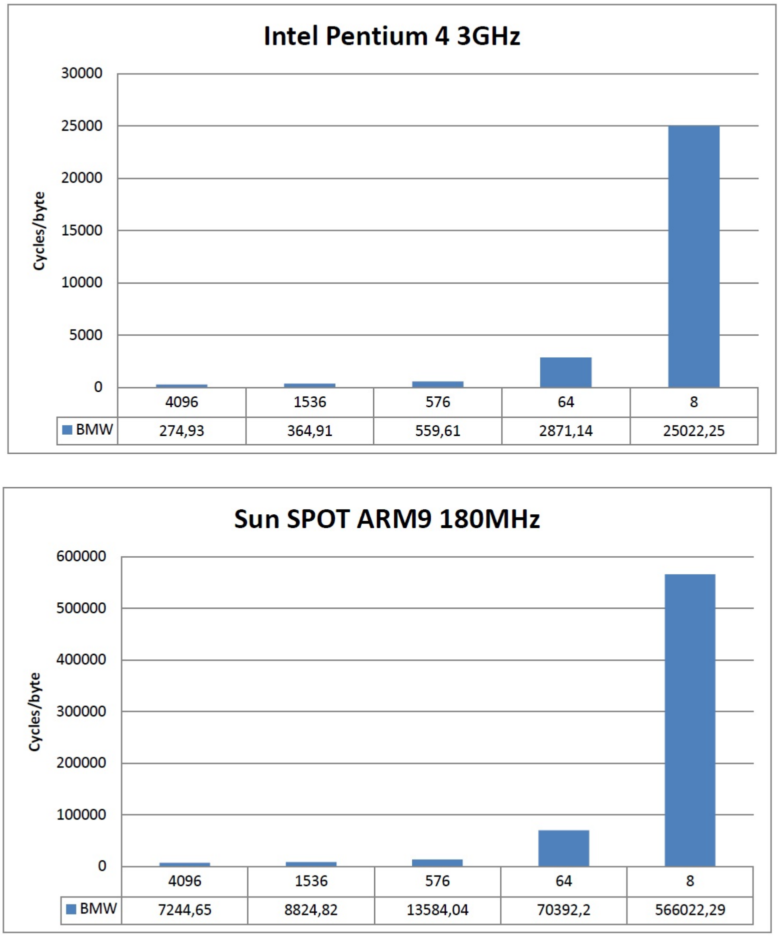
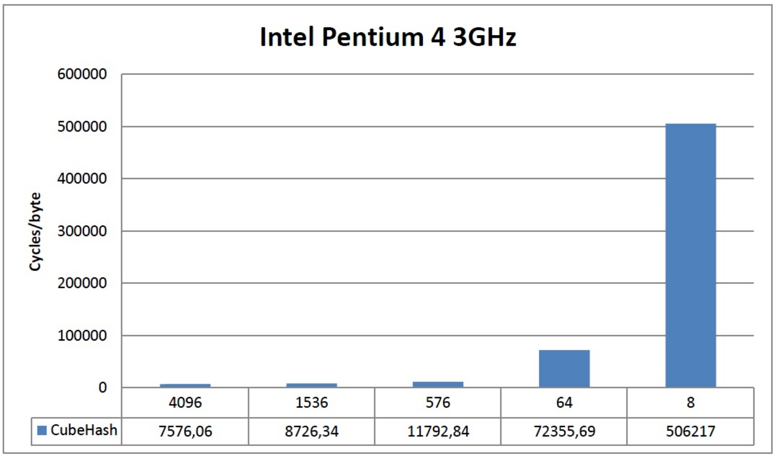


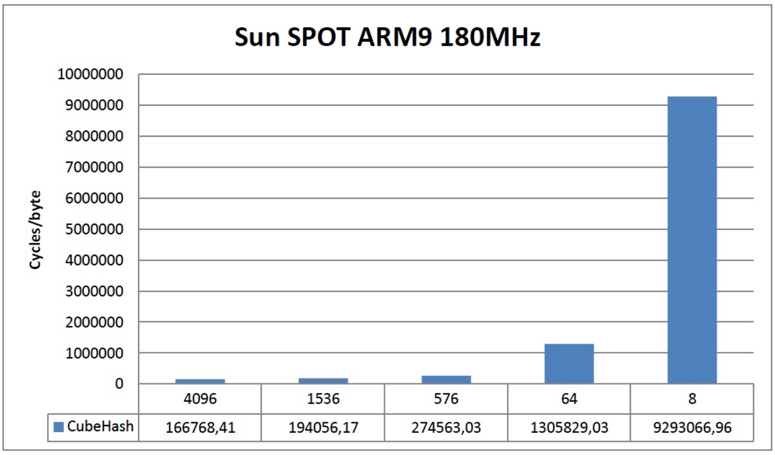
Figure 5.2: Blue Midnight Wish Performance

*Required ROM size: 6.86 kB*

## CubeHash



| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ CubeHash | 7576,06 | 8726,34 | 11792,84 | 72355,69 | 506217 |



| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ CubeHash | 166768,41 | 194056,17 | 274563,03 | 1305829,03 | 9293066,96 |

Figure 5.3: CubeHash Performance

*Required ROM size: 4.19 kB*

## ECHO

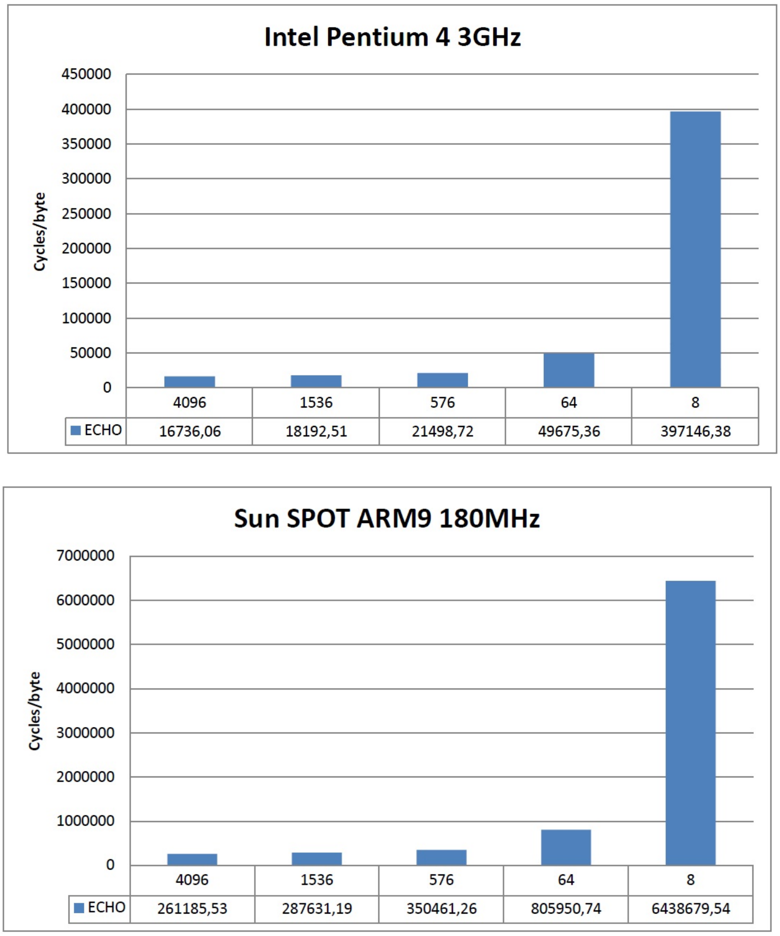


Figure 5.4: ECHO Performance

*Required ROM size: 11.6 kB*

## Fugue

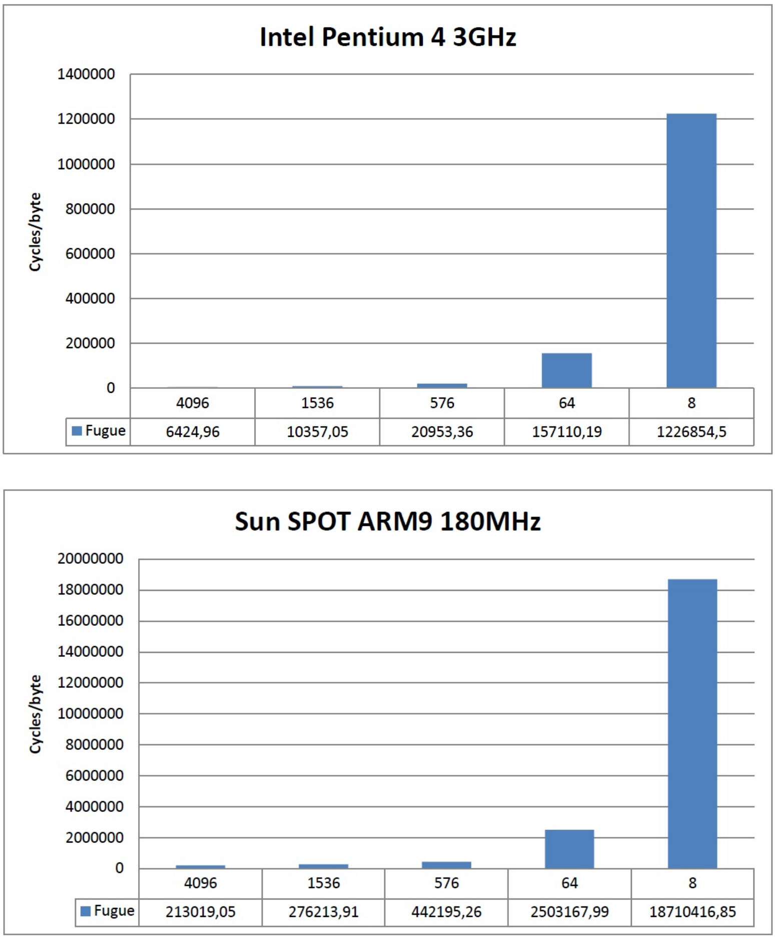
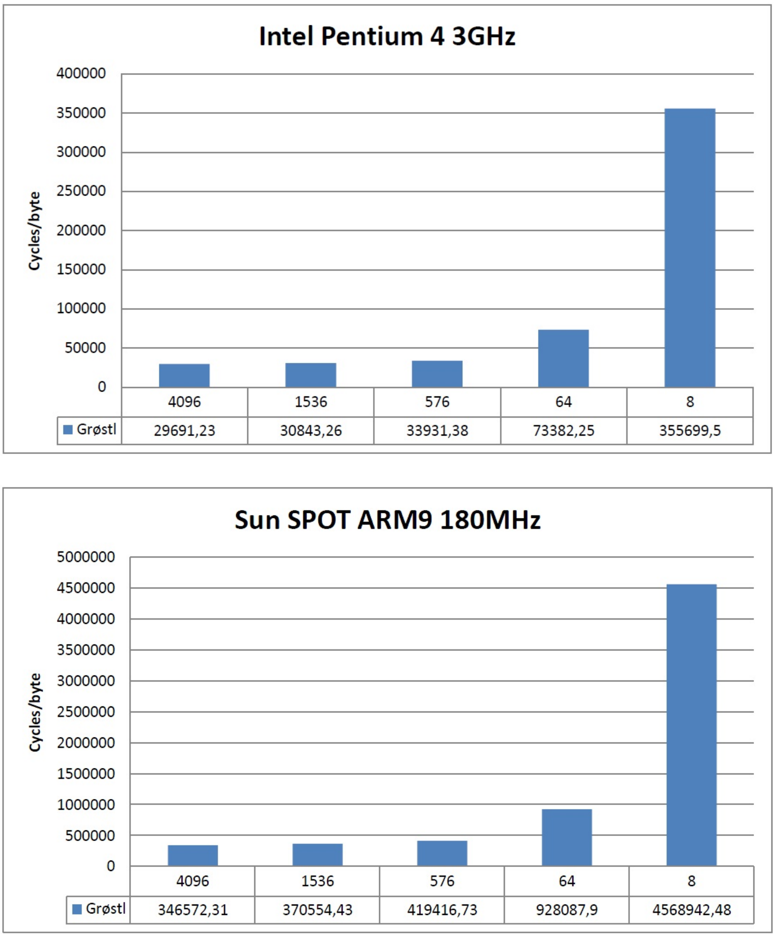


Figure 5.5: Fugue Performance

*Required ROM size: 10.06 kB*

## Grøstl



| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Grøstl | 29691,23 | 30843,26 | 33931,38 | 73382,25 | 355699,5 |

| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Grøstl | 346572,31 | 370554,43 | 419416,73 | 928087,9 | 4568942,48 |

Figure 5.6: Grøstl Performance

*Required ROM size: 6.36 kB*

## JH

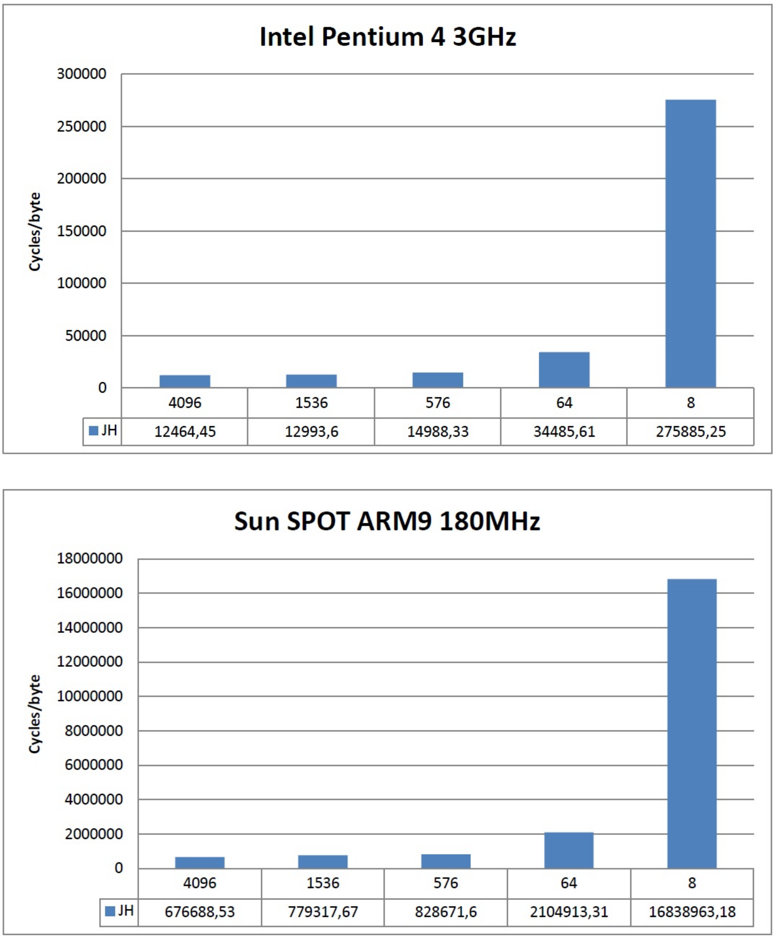
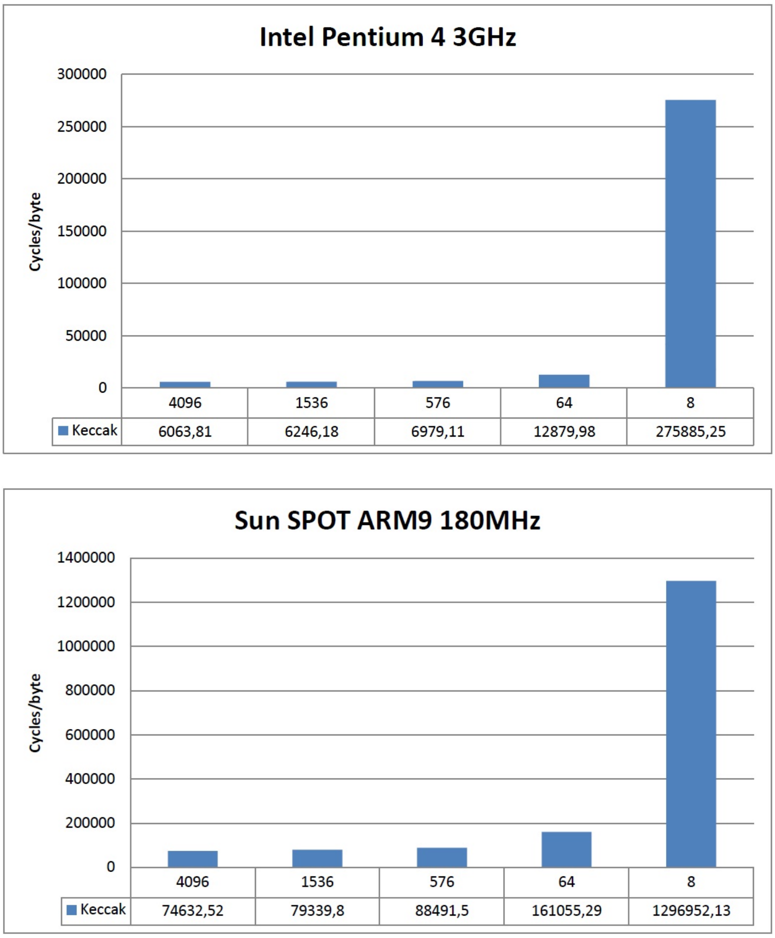


Figure 5.7: JH Performance

*Required ROM size: 4.93 kB*

## Keccak


Intel Pentium 4 3GHz

| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Keccak | 6063,81 | 6246,18 | 6979,11 | 12879,98 | 275885,25 |

Sun SPOT ARM9 180MHz

| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Keccak | 74632,52 | 79339,8 | 88491,5 | 161055,29 | 1296952,13 |

Figure 5.8: Keccak Performance

*Required ROM size: 5.94 kB*

## Luffa

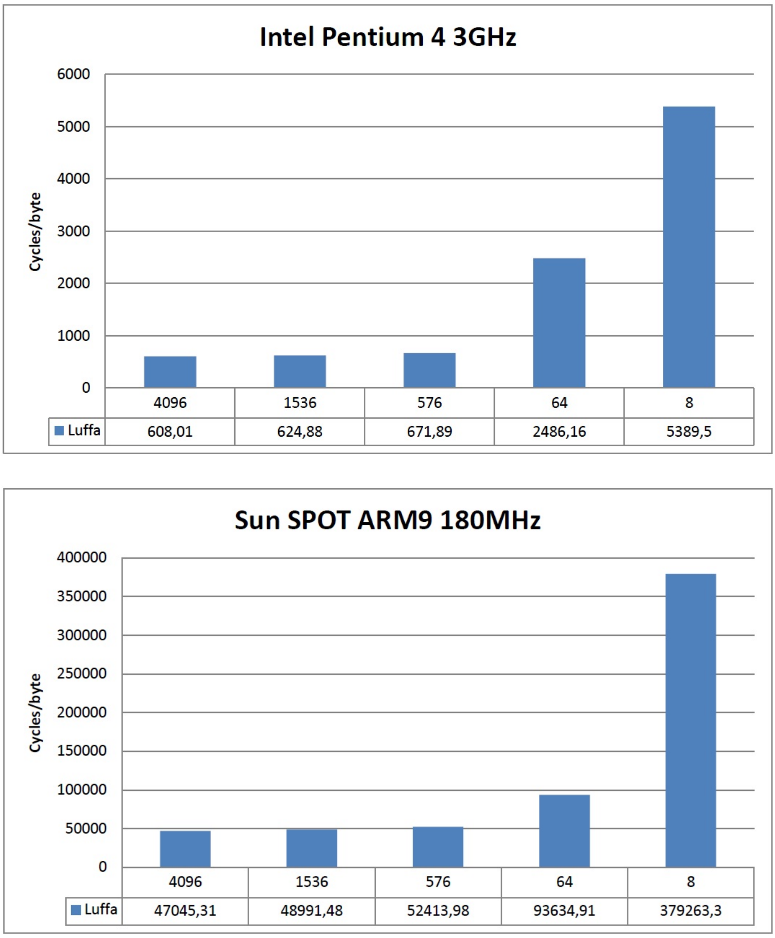
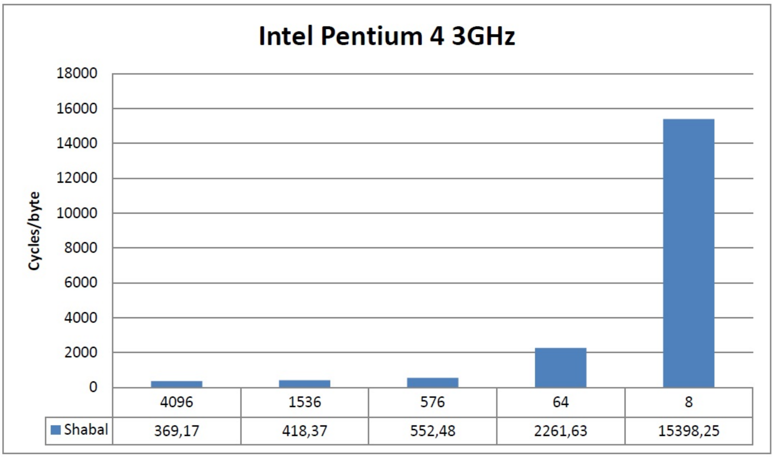


Figure 5.9: Luffa Performance

*Required ROM size: 6.26 kB*

## Shabal



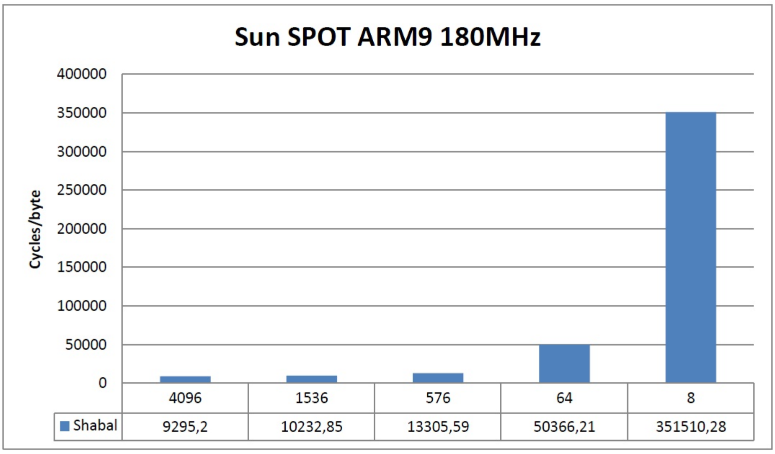


Figure 5.10: Shabal Performance

*Required ROM size: 3.65 kB*

### SHAvite-3

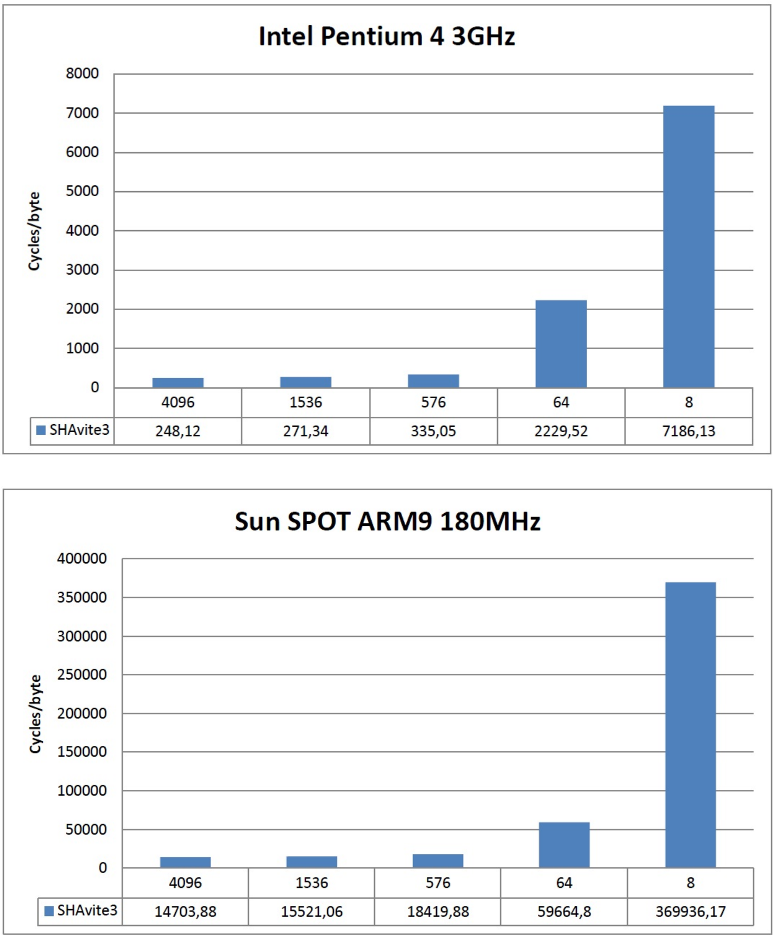
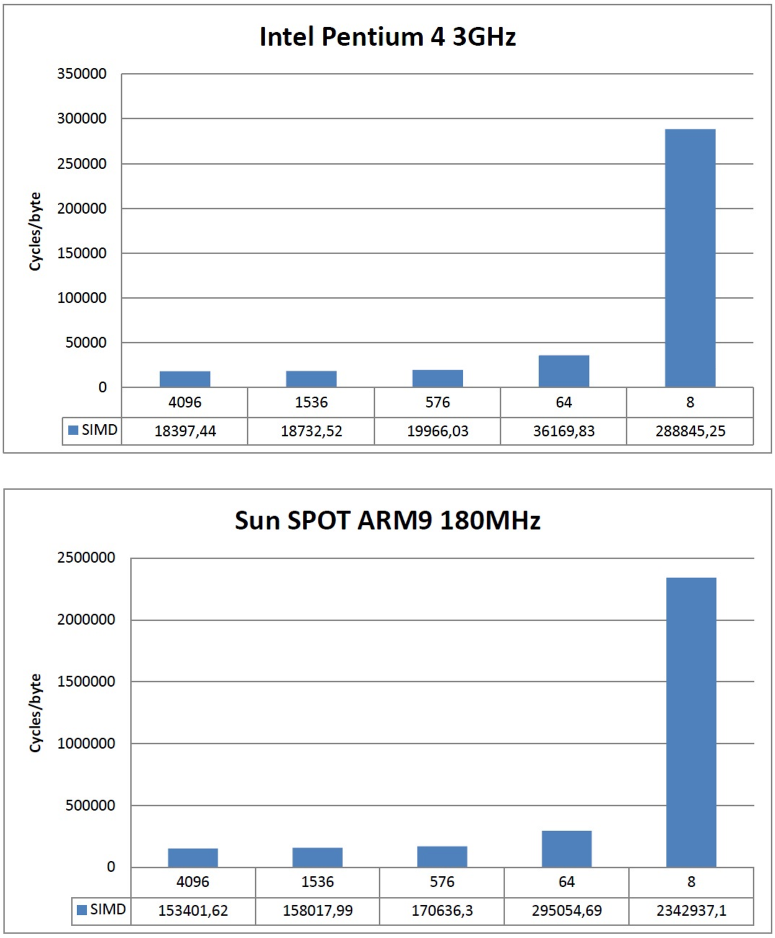


Figure 5.11: SHAvite-3 Performance

*Required ROM size: 17.6 kB*

## SIMD



Intel Pentium 4 3GHz

| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ SIMD | 18397,44 | 18732,52 | 19966,03 | 36169,83 | 288845,25 |

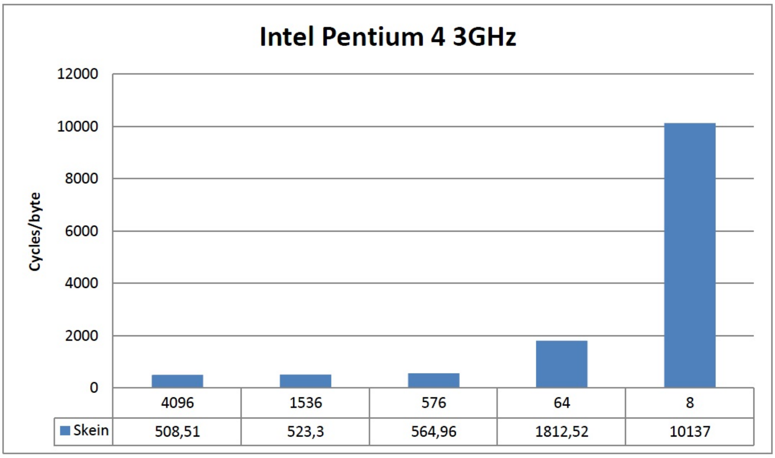
Sun SPOT ARM9 180MHz

| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ SIMD | 153401,62 | 158017,99 | 170636,3 | 295054,69 | 2342937,1 |

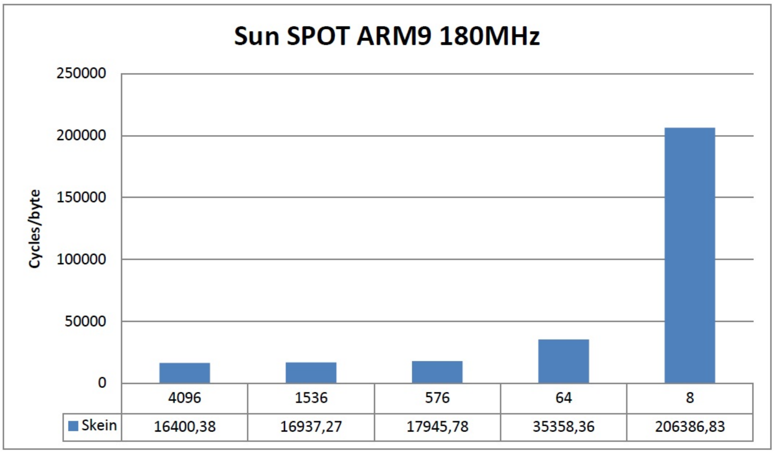
Figure 5.12: SIMD Performance

*Required ROM size: 6.13 kB*

## Skein



| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Skein | 508,51 | 523,3 | 564,96 | 1812,52 | 10137 |



| | 4096 | 1536 | 576 | 64 | 8 |
|---|---|---|---|---|---|
| ■ Skein | 16400,38 | 16937,27 | 17945,78 | 35358,36 | 206386,83 |

Figure 5.13: Skein Performance

*Required ROM size: 7.10 kB*

| **Cycles/byte for 4096 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 245,38 | 248,12 | 317,46 | SHAvite3256 |
| 272,80 | 274,93 | 281,46 | BMW256 |
| 315,68 | 319,04 | 329,70 | BLAKE256 |
| 367,64 | 369,17 | 387,42 | Shabal256 |
| 495,9 | 508,51 | 514,13 | Skein256 |
| 602,62 | 608,01 | 695,07 | Luffa256 |
| 2332,46 | 2343,57 | 2530,97 | Hamsi256 |
| 6013,94 | 6063,81 | 6248,78 | Keccak256 |
| 6322,39 | 6424,96 | 6537,82 | Fugue256 |
| 7442,60 | 7576,06 | 7728,87 | CubeHash256 |
| 12383,79 | 12464,45 | 12643,56 | JH256 |
| 16607,95 | 16736,06 | 16919,29 | ECHO256 |
| 18218,33 | 18397,44 | 18528,83 | SIMD256 |
| 29628,20 | 29691,23 | 29883,10 | Grøstl256 |

| **Cycles/byte for 1536 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 268,14 | 271,34 | 385,74 | SHAvite3256 |
| 318,13 | 328,15 | 338,72 | BLAKE256 |
| 359,04 | 364,91 | 375,01 | BMW256 |
| 417,15 | 418,37 | 477,38 | Shabal256 |
| 512,62 | 523,30 | 566,04 | Skein256 |
| 618,55 | 624,88 | 735,63 | Luffa256 |
| 2369,46 | 2377,24 | 2583,17 | Hamsi256 |
| 6221,30 | 6246,18 | 6549,47 | Keccak256 |
| 8550,46 | 8726,34 | 8937,34 | CubeHash256 |
| 10266,70 | 10357,05 | 10866,79 | Fugue256 |
| 12948,53 | 12993,60 | 13444,31 | JH256 |
| 18017,60 | 18192,51 | 18688,41 | ECHO256 |
| 18678,86 | 18732,52 | 19221,21 | SIMD256 |
| 30785,39 | 30843,26 | 31155,77 | Grøstl256 |

Table 5.1: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 576 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 330,51 | 335,05 | 604,98 | SHAvite3256 |
| 351,84 | 354,66 | 367,78 | Blake256 |
| 550,70 | 552,48 | 825,94 | Shabal256 |
| 552,34 | 559,61 | 581,36 | BMW256 |
| 552,84 | 564,96 | 801,61 | Skein256 |
| 664,36 | 671,89 | 928,79 | Luffa256 |
| 2450,28 | 2468,35 | 2908,26 | Hamsi256 |
| 6954,92 | 6979,11 | 7305,88 | Keccak256 |
| 11530,05 | 11792,84 | 12139,01 | CubeHash256 |
| 14025,29 | 14988,33 | 15552,61 | JH256 |
| 19949,92 | 19966,03 | 20633,97 | SIMD256 |
| 20594,54 | 20953,36 | 21975,75 | Fugue256 |
| 21447,48 | 21498,72 | 22522,79 | ECHO256 |
| 33865,05 | 33931,38 | 34276,03 | Grøstl256 |

| **Cycles/byte for 64 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 680,57 | 689,72 | 748,01 | Blake256 |
| 1343,27 | 1812,52 | 1889,51 | Skein256 |
| 1124,42 | 2229,52 | 2455,31 | SHAvite3256 |
| 2258,28 | 2261,63 | 3812,37 | Shabal256 |
| 1266,67 | 2486,16 | 2526,57 | Luffa256 |
| 2802,23 | 2871,14 | 2995,61 | BMW256 |
| 3528,40 | 3592,92 | 6070,05 | Hamsi256 |
| 12815,51 | 12879,98 | 14057,14 | Keccak256 |
| 34333,25 | 34485,61 | 42122,44 | JH256 |
| 36073,10 | 36169,83 | 37182,34 | SIMD256 |
| 49545,17 | 49675,36 | 51605,04 | ECHO256 |
| 62471,59 | 72355,69 | 74907,43 | CubeHash256 |
| 73213,21 | 73382,25 | 74007,53 | Grøstl256 |
| 153739,06 | 157110,19 | 162306,47 | Fugue256 |

Table 5.2: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 8 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 2951,25 | 2951,63 | 3088,56 | BLAKE256 |
| 5260,04 | 5389,50 | 10670,87 | Luffa256 |
| 7023,61 | 7186,13 | 7579,10 | SHAvite3256 |
| 6300,14 | 10137,00 | 10456,70 | Skein256 |
| 11785,20 | 12190,50 | 12981,87 | Hamsi256 |
| 15362,36 | 15398,25 | 15470,33 | Shabal256 |
| 24660,25 | 25022,25 | 25829,00 | BMW256 |
| 103339,02 | 103809,75 | 105316,47 | Keccak256 |
| 274813,00 | 275885,25 | 320095,42 | JH256 |
| 288105,91 | 288845,25 | 291901,48 | SIMD256 |
| 354701,32 | 355699,50 | 357306,72 | Grøstl256 |
| 396175,13 | 397146,38 | 405545,12 | ECHO256 |
| 494397,41 | 506217,00 | 528178,64 | CubeHash256 |
| 1181568,87 | 1226854,50 | 1250182,79 | Fugue256 |

Table 5.3: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 4096 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 7225,57 | 7244,65 | 8323,44 | BMW256 |
| 9265,59 | 9295,20 | 9305,80 | Shabal256 |
| 14247,87 | 14703,88 | 15155,43 | SHAvite3256 |
| 15829,13 | 16049,01 | 17146,23 | BLAKE256 |
| 16261,50 | 16400,38 | 16416,46 | Skein256 |
| 45044,45 | 47045,31 | 47964,37 | Luffa256 |
| 73869,55 | 74632,52 | 75418,86 | Keccak256 |
| 152272,21 | 153401,62 | 154566,34 | SIMD256 |
| 162051,47 | 166768,41 | 170644,45 | CubeHash256 |
| 211721,88 | 213019,05 | 214382,31 | Fugue256 |
| 259742,44 | 261185,53 | 262626,81 | ECHO256 |
| 341576,27 | 346572,31 | 351576,22 | Grøstl256 |
| 670755,25 | 676688,53 | 682705,99 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

| **Cycles/byte for 1536 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 8801,49 | 8824,82 | 11141,11 | BMW256 |
| 10052,41 | 10232,85 | 10238,78 | Shabal256 |
| 15489,48 | 15521,06 | 17663,36 | SHAvite3256 |
| 15899,37 | 16479,50 | 17219,64 | BLAKE256 |
| 16755,94 | 16937,27 | 17001,17 | Skein256 |
| 48255,04 | 48991,48 | 49225,08 | Luffa256 |
| 78237,43 | 79339,80 | 80538,97 | Keccak256 |
| 156868,67 | 158017,99 | 159768,41 | SIMD256 |
| 192055,46 | 194056,17 | 197254,42 | CubeHash256 |
| 274624,17 | 276213,91 | 278397,26 | Fugue256 |
| 285548,90 | 287631,19 | 289857,88 | ECHO256 |
| 358732,56 | 370554,43 | 381650,82 | Grøstl256 |
| 775918,61 | 779317,67 | 782712,08 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

Table 5.4: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Cycles/byte for 576 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 12771,16 | 13305,59 | 13318,75 | Shabal256 |
| 13321,63 | 13584,04 | 19580,24 | BMW256 |
| 16950,20 | 17351,48 | 18905,47 | BLAKE256 |
| 17782,80 | 17945,78 | 17970,88 | Skein256 |
| 18376,43 | 18419,88 | 21094,73 | SHAvite3256 |
| 52356,80 | 52413,98 | 52737,47 | Luffa256 |
| 87915,88 | 88491,50 | 91843,89 | Keccak256 |
| 166456,86 | 170636,30 | 173316,46 | SIMD256 |
| 271857,87 | 274563,03 | 279125,11 | CubeHash256 |
| 344922,19 | 350461,26 | 356261,96 | ECHO256 |
| 405246,67 | 419416,73 | 421033,69 | Grøstl256 |
| 438264,92 | 442195,26 | 446376,91 | Fugue256 |
| 819573,33 | 828671,60 | 837601,74 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

| **Cycles/byte for 64 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 31304,13 | 32370,02 | 33474,77 | BLAKE256 |
| 34681,42 | 35358,36 | 35690,64 | Skein256 |
| 48044,74 | 50366,21 | 52670,80 | Shabal256 |
| 59303,28 | 59664,80 | 67866,37 | SHAvite3256 |
| 70075,04 | 70392,20 | 108408,90 | BMW256 |
| 93142,73 | 93634,91 | 93810,89 | Luffa256 |
| 160815,70 | 161055,29 | 169345,14 | Keccak256 |
| 294708,55 | 295054,69 | 313714,86 | SIMD256 |
| 804118,77 | 805950,74 | 832831,25 | ECHO256 |
| 925242,90 | 928087,90 | 932236,31 | Grøstl256 |
| 1293450,33 | 1305829,03 | 1338317,15 | CubeHash256 |
| 2056855,65 | 2104913,31 | 2167669,40 | JH256 |
| 2499328,19 | 2503167,99 | 2567218,87 | Fugue256 |
| N/A | N/A | N/A | Hamsi256 |

Table 5.5: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Cycles/byte for 8 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 135025,91 | 137743,47 | 150609,67 | BLAKE256 |
| 203277,62 | 206386,83 | 211322,02 | Skein256 |
| 325306,92 | 351510,28 | 354179,03 | Shabal256 |
| 366775,27 | 369936,17 | 415334,07 | SHAvite3256 |
| 374296,11 | 379263,30 | 384047,28 | Luffa256 |
| 563348,94 | 566022,29 | 879709,11 | BMW256 |
| 1295028,32 | 1296952,13 | 1363139,68 | Keccak256 |
| 2340421,70 | 2342937,10 | 2491367,39 | SIMD256 |
| 4563850,93 | 4568942,48 | 4704900,83 | Grøstl256 |
| 6422366,05 | 6438679,54 | 6652598,15 | ECHO256 |
| 9239108,19 | 9293066,96 | 9536356,32 | CubeHash256 |
| 16458116,05 | 16838963,18 | 17339729,84 | JH256 |
| 18689930,39 | 18710416,85 | 19212191,51 | Fugue256 |
| N/A | N/A | N/A | Hamsi256 |

Table 5.6: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Hash** | **Size in kB** |
|---|---|
| Shabal | 3,65 |
| CubeHash | 4,19 |
| JH | 4,93 |
| Keccak | 5,94 |
| SIMD | 6,13 |
| Luffa | 6,26 |
| Grøstl | 6,36 |
| BMW | 6,86 |
| Skein | 7,10 |
| BLAKE | 7,31 |
| Fugue | 10,06 |
| ECHO | 11,6 |
| SHAVite3 | 17,6 |
| Hamsi | N/A |

Table 5.7: Cost Measurements

## 5.7 Summary of the Results

The results, presented in the previous section, shows that there exists relatively large differences between the candidates. Some of the candidates performs best on the lower inputs, while others are again favouring the larger input messages. As previously mentioned, Hamsi was unable to compile for the Sun SPOT.

The difference between the Intel and the Sun SPOT results confirms the need for research regarding performance on constrained platforms.

Our Java implementations are relatively small, and should fit well on most constrained devices available today. However, the largest implementation, SHAvite-3, is almost five times larger than Shabal, which is the smallest implementation.

# 6 Conclusions and Future Work

The result of this work is Java implementations of the 14 candidates remaining in NIST's public competition to develop a new cryptographic hash algorithm. At the time of writing, there exist no other publicly available Java implementations of **all** candidates[1]. The code written by the authors is based on the reference implementation, in C, of the candidates remaining in the second round.

The implementation is done by both authors of this thesis in cooperation, to make sure that all candidates are treated equally. Our design choices in general are provided in Chap. 3. Chap 4 provides an in-depth

[1] The sphlib project has translated 9 of the 14 candidates. http://www.saphir2.com/sphlib/

look at each candidate.

The Java implementations are tested on a Java enabled Sun SPOT as well as on a regular desktop computer for reference. The performance is measured in cycles/byte. We have also measured the ROM size required to store the compiled Java implementation of each candiate.

A paper with our results has been submitted to the Second SHA-3 Candidate Conference at the University of California in Santa Barbara. The paper is attached in Appendix D. The purpose of this conference is to discuss the 14 remaining second-round candidates, and to obtain valuable feedback for the selection of the finalists. Authors will be notified June 18, 2010 if their paper has been accepted.

Our results measured in cycles/byte show that the Java implementations of the candidates execute fairly poor on the Sun SPOT platform. [26] provides a list of measurements of some of the C implementations, and this confirms what we explained in Chap 4; there is still room for improvements on our implementations. Because of this, the next section propose some future work related to this thesis.

## 6.1 Future Work

Due to time limits, the priority of this work has been to get a working implementation of all 14 candidates. Hence, none of the candidates have been given any special effort in terms of optimization. We propose future work to consist of giving each candidate's Java implementation a closer look, as to how one can optimize for both speed and size. A possibility here can be to create two implementations of each candidate, one optimized for size, and one optimized for speed.

The 32-bit Java Card 3 technology has just arrived, and it should be an interesting task to implement the candidates on this platform to get extensive testing for a future smart card technology.

# Bibliography

[1] Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (sha–3) family, October 2007. [online] http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

[2] A closer look at the squawk vm, April 2009. [online] http://research.sun.com/spotlight/2009/2009-04-16_squawk_closer.html.

[3] Status report on the first round of the sha-3 cryptographic hash algorithm competition, September 2009. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf.

[4] Ansi c cryptographic api profile for sha-3 candidate algorithm submissions, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/documents/SHA3-C-API.pdf.

[5] Arm, April 2010. [online] http://www.arm.com/index.php.

[6] Arm instruction set architectures, April 2010. [online] http://www.arm.com/products/processors/technologies/instruction-set-architectures.php.

[7] Armed for the living room, May 2010. [online] http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html.

[8] Blake - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/BLAKE_Round2.zip.

[9] Blue midnight wish - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Blue_Midnight_Wish_Round2.zip.

[10] A brief history of netbeans, June 2010. [online] http://netbeans.org/about/history.html.

[11] Cryptographic hash algorithm competition, April 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[12] Cubehash - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/CubeHash_Round2.zip.

[13] Daim - digital arkivering og innlevering av masteroppgaver, May 2010. [online] http://daim.idi.ntnu.no/.

[14] The differences between java, c and c++, May 2010. [online] http://www.programmersheaven.com/2/FAQ-JAVA-Differences-Between-JAVA-And-C-CPP.

[15] Echo - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/ECHO_Round2.zip.

[16] Echo hash function, June 2010. [online] http://crypto.rd.francetelecom.com/echo/description/.

[17] Fugue - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Fugue_Round2.zip.

[18] Grøstl - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Grostl_Round2.zip.

[19] Hamsi - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Hamsi_Round2.zip.

[20] A hardware wrapper for the sha-3 hash algorithms, June 2010. [online] http://eprint.iacr.org/2010/124.pdf/.

[21] High-speed hardware implementations of blake, blue midnight wish, cubehash, echo, fugue, grøstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein, June 2010. [online] http://eprint.iacr.org/2009/510.pdf.

[22] Java.net - the source for java technology collaboration, May 2010. [online] https://netbeans-spot.dev.java.net/.

[23] Jh - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/JH_Round2.zip.

[24] Keccak - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Keccak_Round2.zip.

[25] Luffa - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Luffa_Round2.zip.

[26] Measurements of hash functions, June 2010. [online] http://bench.cr.yp.to/results-hash.html.

[27] Netbeans ide 6.8 features, May 2010. [online] http://netbeans.org/features/index.html.

[28] Nist comments on cryptanalytic attacks on sha-1, April 2010. [online] http://csrc.nist.gov/groups/ST/hash/statement.html.

[29] Shabal - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Shabal_Round2.zip.

[30] Shavite-3 - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/SHAvite-3_Round2.zip.

[31] Simd - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/SIMD_Round2.zip.

[32] Skein - round 2 submission package, May 2010. [online] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Skein_Round2.zip.

[33] Speeding up the widepipe: Secure and fast hashing, June 2010. [online] http://www.cosic.esat.kuleuven.be/publications/article-1449.pdf.

[34] The squawk project, April 2010. [online] http://research.sun.com/projects/squawk/.

[35] Sun spot theory of operation, red release 5.0, April 2010. [online] https://www.sunspotworld.com/docs/Red/SunSPOT-TheoryOfOperation.pdf.

[36] Sunspotworld, April 2010. [online] `https://www.sunspotworld.com/`.

[37] Using the at91 timer/counter, May 2010. [online] `http://www.sunspotworld.com/docs/AppNotes/TimerCounterAppNote.pdf`.

[38] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Assche. sponge functions, 2007.

[39] Eli Biham and Orr Dunkelman. A framework for iterative hash functions: Haifa. In *In Proceedings of Second NIST Cryptographic Hash Workshop, 2006 . Available from: www.csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm*, 2006.

[40] Murali Krishna Reddy Danda. Design and analysis of hash functions. Master's thesis, Victoria University, 2007.

[41] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999.

[42] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.

[43] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[44] Bart Preneel. The state of hash functions and the nist sha-3 competition. pages 1–11, 2009.

[45] Nik Shaylor. A java virtual machine architecture for very small devices. In *In Language, Compiler, and Tool Support for Embedded Sys-*

*tems: Proceedings of LCTES '03, number 38(7) in ACM SIGPLAN Notices*, pages 34–41. ACM Press, 2003.

[46] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java&#8482; on the bare metal. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, New York, NY, USA, 2005. ACM.

[47] William Stallings. *Cryptography and Network Security: Principles and Practice.* Prentice Hall, 2010.

[48] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *In Proceedings of Crypto*, pages 17–36. Springer, 2005.

[49] Tao Xie and Dengguo Feng. How to find weak input differences for md5 collision attacks. Cryptology ePrint Archive, Report 2009/223, 2009. [online] `http://eprint.iacr.org/`.

# A

# The CheckKAT Application

This appendix presents the Java code for the CheckKAT application used to verify the correctness of the translated Java implementations.

```
public class CheckKAT {

  static long databitlen = 0;
  static byte[] data = null;

  public static void main(String[] args) {

    Class[] algorithms = { SHAvite3.class, Blake.class,
        BlueMidnightWish.class, Shabal.class, Skein.class,Luffa.
        class, Hamsi.class, Keccak.class, CubeHash.class, SIMD.
        class, Groestl.class, Echo.class, JH.class, Fugue.class
```

```
      };
    start(algorithms);
  }

  private static void start(Class[] algorithms) {
    for (Class algorithm : algorithms) {
      System.out.println("Now testing: " + algorithm.
          getSimpleName());
      boolean result = performCheck(algorithm);
      System.out.println(result ? "100percent Match" : " ");
      System.out.println(" ");
    }
    System.out.println("All done with "+algorithms.length+"
        algoritms");
  }

  public static boolean performCheck(Class clazz) {

    File file = new File(System.getProperty("user.dir")+"/
        shortmsgkat/"+ clazz.getSimpleName() + "ShortMsgKAT_256
        .txt");

    Method method = null;
    try {
      method = clazz.getDeclaredMethod("Hash", new Class[] {
          byte[].class, long.class, byte[].class });
    } catch (Exception e1) {
      System.out.println("Unable to create reflection method");
    }

    try {
      BufferedReader br = new BufferedReader(new
          InputStreamReader(new FileInputStream(file)));
      String line;
      while ((line = br.readLine()) != null) {

        if (line.startsWith("Len")) {
```

```
       databitlen = Long.parseLong((line.split("= ")[1]));
     }
     if (line.startsWith("Msg")) {
       String value = null;
       int pointer = 0;
       String tmp = line.split("= ")[1];
       data = new byte[tmp.length() / 2];
       for (int i = 0; i < data.length; i++) {
         value = tmp.substring(pointer, pointer + 2);
         data[i] = (byte) Integer.parseInt(value.trim(), 16);
         pointer += 2;
       }
     }

     if (line.startsWith("MD")) {
       byte[] hashval = new byte[32];
       try {
         method.invoke("foo", data, databitlen, hashval);
       } catch (Exception e) {
         System.err.println("Algorithm threw an Exception. Stacktrace is commented out\n");
       }

       StringBuffer sb = new StringBuffer(hashval.length * 2);
       for (int i = 0; i < 32; ++i) {
         sb.append(hexChar[(hashval[i] & 0xf0) >>> 4]);
         sb.append(hexChar[hashval[i] & 0x0f]);

       }
       String tmp = line.split("= ")[1];
       if (!tmp.equals(sb.toString())) {
         error(databitlen, sb.toString(), tmp);
         return false;
       }
    }
```

```
     }
     br.close();
    } catch (IOException e) {
      System.err.println(clazz.getSimpleName()+ " ShortMsgKAT_256 not found");
      return false;
    }
    return true;
  }

  private static void error(long databitlen, String hashval, String tmp){
    System.out.println("Got an error for len: " + databitlen);
    System.out.println("Your hash:  " + hashval);
    System.out.println("Correct is: " + tmp);
  }

  static char[] hexChar = { '0', '1', '2', '3', '4', '5','6', '7', '8', '9','A', 'B', 'C', 'D', 'E', 'F' };
}
```

# B

# The DeployOnSpot Application

This appendix presents the Java code for the *DeployOnSpot* application used to deploy the candidates on the Sun SPOT. In this example the specific candidate to be deployed is substituted with *CandidateName*.

In the code shown below, some generalizations are done. For instance, the *TimeForOneTick* variable represents the values for the distinct clocks depicted in Tab. 2.4. Further, the *TC Clock Input*, also specified in Tab. 2.4, is represented as the variable *Clock* in the *configure()* method.

```
import com.sun.spot.peripheral.IAT91_TC;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.peripheral.TimerCounterBits;

import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

/**
 * The startApp method of this class is called
 * by the VM to start the application.
 *
 * The manifest specifies this class as MIDlet-1,
 * which means it will be selected for execution.
 */
public class DeployOnSpot extends MIDlet implements TimerCounterBits {


    private static byte[] data = new byte[4097];
    private static byte[] hashval = new byte[32];
    private static final int[] bitlen  = {8, 64, 576,1536,4096};
    private static long databitlen;
    private IAT91_TC timer;
    private ITriColorLED [] leds = EDemoBoard.getInstance().getLEDs();


    protected void startApp() throws MIDletStateChangeException {

        int length = 5001;
        timer = Spot.getInstance().getAT91_TC(0);
        timer.configure(TC\_CAPT | Clock);

        try {
            databitlen = bitlen[i] * 8;
            for(int i=0; i < length; i++) {
                timer.enableAndReset();
                CandidateName.Hash(data, databitlen, hashval);
                int cntr = timer.counter();
                timer.disable();
                double interval = cntr * TimeForOneTick;
                System.out.println(interval);
            }
            System.out.println("Databitlen: "+databitlen/8+" bytes");
            success();
        } catch (Exception e) {
            fail();
            System.err.println("Got the following exception: "+e);
        }
        notifyDestroyed();
    }
```

# C Attachments

The attachments to this thesis consist of an electronic attachment and a DVD. The electronic attachment is submitted to the DAIM system [13]. The DVD is appended to this thesis.

## C.1 Electronic Attachments

The electronic attachment (MasterThesis.zip), uploaded to DAIM, includes the following directories:

- **Thesis** all files related to the writing of this thesis.

- **Paper** all files related to the writing of the paper submitted to Second SHA-3 Conference.
- **Source-Code** all code produced throughout the work of this thesis.
- **Results** Excel spreadsheets with all data collected from the measurements of the candidates.

## C.2 Attached DVD

Appended to this thesis is a DVD containing all the source-code produced when translating the candidates from C to Java, all files related to the writing of this thesis, the script used to verify correctness of the translated candidates, all collected data from measurements and also the paper submitted to the Second SHA-3 Candidate Conference. The paper can also be found in Appendix D.

# D

# Paper Submitted to the Second SHA-3 Candidate Conference

# Java Implementation and Performance Analysis of the Second Round SHA-3 Candidates on Constrained Devices

Kim Andre Martinsen, Mats Knutsen, Danilo Gligoroski

Faculty of Information Technology, Mathematics and Electrical Engineering, Institute of Telematics,
Norwegian University of Science and Technology, Trondheim, Norway,
e-mail: {kimandm, matskn}@stud.ntnu.no , Danilo.Gligoroski@item.ntnu.no

**Abstract**

In this paper, we show the results of our implementation of the second round SHA-3 candidates in Java and perform a cost and performance analysis of them on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM requirements. The analysis is conducted on the Sun SPOT platform, by Sun Microsystems, with a Squawk Virtual Machine.

## 1 Introduction

The National Institute of Standards and Technology (NIST) is in the process of selecting a new cryptographic hash algorithm through a public competition [1]. At the First SHA-3 Candidate Conference several individuals stressed the need for supporting constrained platforms such as mobile phones and smart cards [1].

To our knowledge, no work has been demonstrated for comparing Java implementations of the SHA-3 candidates on constrained platforms. In this paper, we compare the implementation costs of the remaining SHA-3 candidates on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM costs.

In the remaining part of this section, we attempt to explain the importance of considering Java implementations on constrained devices for hash implementations. Firstly, the number of constrained devices, such as smart cards, surrounding us daily is rapidly increasing. In 2008 5.085 billion smart cards shipped globally, while the estimates for 2010 are 5.455 billion smart cards [2]. Further, smart cards have a wide range of applications spanning from cryptography and health care to identification and authentication.

Java Card is the Java initiative for running Java applications on smart cards. With Java being one of the most popular programming languages, Java Card simplifies the development of applications for smart cards. Java Card 3.0, released late 2009, is tailored for next generation of smart cards, which have 32-bit processors [3].

For Java to run on the ARM, a Virtual Machine(VM) resides on top of the CPU. This VM, Squawk VM, is designed for the next generation of smart cards [4]. Squawk is a small VM running without any Operating System (OS), this allows for a simpler and more compact VM. Java VMs are typically written in native languages, such as C and C++. Squawk however, has a different approach as it is written almost entirely in Java. This provides ease of portability and a seamless integration of the VM and the application resources such as threads and objects [4].

At the time of writing, no commercial version of the new Java Card running version 3.0 is available. To compensate for this, the performance tests are conducted on a Sun SPOT[1]. The Sun SPOT is a small embedded device developed at Sun Labs. As with Java Card 3.0, Sun SPOT is running the Squawk VM on a low-cost 32-bit processor [4].

This paper is structured as follows. Section 2 gives a brief description of the challenges faced, when writing C code to Java code. Section 3 gives an overview of our performance and cost measurements. In Section 4, the results from our measurements are presented, while Section 5 concludes the findings from this paper.

# 2 Java Implementation of the Candidates

In contrast to low-level languages, like C, the memory management in Java is not handled by the programmer. For instance, in C the *union* is a value that may have several representations in various formats. The primary usefulness of a union is to conserve space, since it provides a way of letting many different types be stored in the same space. The Java language guarantees type safety, so each member of a *union* has to be implemented as single variables which have to be synchronized when one of them are alternated.

The Java implementations of the candidates are based on the reference implementation representing the candidate in the second round. To our knowledge, no optimized Java implementation of all the candidates are publicly presented. With the challenges presented above, and also considering time constraints,

[1] SPOT stands for Small Programmable Object Technology

none of the candidates have been optimized in any way. The goal of this work is to give a balanced view of the candidates implemented on a new platform.

# 3 Overview of Our Performance and Cost Measurements

This section presents an overview of our comparison of the remaining SHA-3 candidates, in terms of performance and cost. In constrained devices, power consumption and the area required to implement the algorithm are limiting factors [1]. Power consumption usually reduces to computational efficiency, which will be measured in *cycles/byte*. The size, in kilobytes (kB), of the compiled file loaded on the Squawk-capable constrained device is used as the measurement for ROM requirements of the algorithms.

In our comparison, we consider Java translated implementations of the submitted round 2 reference implementations. Further, our main target for estimation and comparison are those variants of the hash functions which produce a 256-bit message digest. External functionality like salting or keyed hashing modes has not been implemented.

## 3.1 Measuring Performance

To measure the performance of the candidates, a Java MIDlet[2] were deployed on the Sun SPOT. The MIDlet executed each of the algorithms 4500 times for the follow input lengths: 8, 64, 576, 1536 and 4096 bytes. Each Sun SPOT processor board has two AT91 Timer Counters that are part of the ARM920T system-on-a-chip (SOC) [5]. A AT91 Timer Counter were used to measure the time consumed by each algorithm, while processing the input messages. The output of these AT91 Timer Counters is microseconds.

Our results are presented as a lower-quartile, a median and a upper-quartile of the 4500 measurements taken of each candidate, for each of the five input lengths.

## 3.2 Measuring Cost

Applications deployed onto the Squawk VM are first compiled to regular Java *class* files and then further to a *suite* file. Squawk VM includes a mechanism

[2] A MIDlet is an application that runs on implementations of the Mobile Information Device Profile, one of the Java ME specifications.

for serializing a graph of objects, these objects becomes a collection of internal classes encapsulated in the *suite*. On average, a suite file is 35% the size of the corresponding class file [4].

As each candidate can not be deployed to the device without extra application code, we have only measured the size of the class file corresponding to each algorithm. This is the compiled code before it is packed into a *suite* with the rest of the files needed.

# 4 Results

This section presents the results generated from our work. Worth mentioning is that Hamsi was not able to compile for Squawk VM, and hence no results for Hamsi are presented in this section. This is due to Hamsi exceeding a Java constant size restriction in Squawk.

## 4.1 Performance Results

In addition to measure the performance of the Java implementations on the Sun SPOT, a performance test was conducted on a Intel Pentium 4 3GHz desktop computer as a frame of reference. The results from the Intel test are depicted in Tab. 1-3.

Tab. 4-6 depicts the results from the performance test on the Sun SPOT ARM920T 180MHz running Squawk VM.

## 4.2 Cost Results

Tab. 7 depicts our cost measurements of the SHA3 candidates.

| **Cycles/byte for 4096 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 245,38 | 248,12 | 317,46 | SHAvite3256 |
| 272,80 | 274,93 | 281,46 | BMW256 |
| 315,68 | 319,04 | 329,70 | BLAKE256 |
| 367,64 | 369,17 | 387,42 | Shabal256 |
| 495,9 | 508,51 | 514,13 | Skein256 |
| 602,62 | 608,01 | 695,07 | Luffa256 |
| 2332,46 | 2343,57 | 2530,97 | Hamsi256 |
| 6013,94 | 6063,81 | 6248,78 | Keccak256 |
| 6322,39 | 6424,96 | 6537,82 | Fugue256 |
| 7442,60 | 7576,06 | 7728,87 | CubeHash256 |
| 12383,79 | 12464,45 | 12643,56 | JH256 |
| 16607,95 | 16736,06 | 16919,29 | ECHO256 |
| 18218,33 | 18397,44 | 18528,83 | SIMD256 |
| 29628,20 | 29691,23 | 29883,10 | Grøstl256 |

| **Cycles/byte for 1536 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 268,14 | 271,34 | 385,74 | SHAvite3256 |
| 318,13 | 328,15 | 338,72 | BLAKE256 |
| 359,04 | 364,91 | 375,01 | BMW256 |
| 417,15 | 418,37 | 477,38 | Shabal256 |
| 512,62 | 523,30 | 566,04 | Skein256 |
| 618,55 | 624,88 | 735,63 | Luffa256 |
| 2369,46 | 2377,24 | 2583,17 | Hamsi256 |
| 6221,30 | 6246,18 | 6549,47 | Keccak256 |
| 8550,46 | 8726,34 | 8937,34 | CubeHash256 |
| 10266,70 | 10357,05 | 10866,79 | Fugue256 |
| 12948,53 | 12993,60 | 13444,31 | JH256 |
| 18017,60 | 18192,51 | 18688,41 | ECHO256 |
| 18678,86 | 18732,52 | 19221,21 | SIMD256 |
| 30785,39 | 30843,26 | 31155,77 | Grøstl256 |

Table 1: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 576 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 330,51 | 335,05 | 604,98 | SHAvite3256 |
| 351,84 | 354,66 | 367,78 | Blake256 |
| 550,70 | 552,48 | 825,94 | Shabal256 |
| 552,34 | 559,61 | 581,36 | BMW256 |
| 552,84 | 564,96 | 801,61 | Skein256 |
| 664,36 | 671,89 | 928,79 | Luffa256 |
| 2450,28 | 2468,35 | 2908,26 | Hamsi256 |
| 6954,92 | 6979,11 | 7305,88 | Keccak256 |
| 11530,05 | 11792,84 | 12139,01 | CubeHash256 |
| 14025,29 | 14988,33 | 15552,61 | JH256 |
| 19949,92 | 19966,03 | 20633,97 | SIMD256 |
| 20594,54 | 20953,36 | 21975,75 | Fugue256 |
| 21447,48 | 21498,72 | 22522,79 | ECHO256 |
| 33865,05 | 33931,38 | 34276,03 | Grøstl256 |

| **Cycles/byte for 64 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 680,57 | 689,72 | 748,01 | Blake256 |
| 1343,27 | 1812,52 | 1889,51 | Skein256 |
| 1124,42 | 2229,52 | 2455,31 | SHAvite3256 |
| 2258,28 | 2261,63 | 3812,37 | Shabal256 |
| 1266,67 | 2486,16 | 2526,57 | Luffa256 |
| 2802,23 | 2871,14 | 2995,61 | BMW256 |
| 3528,40 | 3592,92 | 6070,05 | Hamsi256 |
| 12815,51 | 12879,98 | 14057,14 | Keccak256 |
| 34333,25 | 34485,61 | 42122,44 | JH256 |
| 36073,10 | 36169,83 | 37182,34 | SIMD256 |
| 49545,17 | 49675,36 | 51605,04 | ECHO256 |
| 62471,59 | 72355,69 | 74907,43 | CubeHash256 |
| 73213,21 | 73382,25 | 74007,53 | Grøstl256 |
| 153739,06 | 157110,19 | 162306,47 | Fugue256 |

Table 2: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 8 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 2951,25 | 2951,63 | 3088,56 | BLAKE256 |
| 5260,04 | 5389,50 | 10670,87 | Luffa256 |
| 7023,61 | 7186,13 | 7579,10 | SHAvite3256 |
| 6300,14 | 10137,00 | 10456,70 | Skein256 |
| 11785,20 | 12190,50 | 12981,87 | Hamsi256 |
| 15362,36 | 15398,25 | 15470,33 | Shabal256 |
| 24660,25 | 25022,25 | 25829,00 | BMW256 |
| 103339,02 | 103809,75 | 105316,47 | Keccak256 |
| 274813,00 | 275885,25 | 320095,42 | JH256 |
| 288105,91 | 288845,25 | 291901,48 | SIMD256 |
| 354701,32 | 355699,50 | 357306,72 | Grøstl256 |
| 396175,13 | 397146,38 | 405545,12 | ECHO256 |
| 494397,41 | 506217,00 | 528178,64 | CubeHash256 |
| 1181568,87 | 1226854,50 | 1250182,79 | Fugue256 |

Table 3: Performance Measurements on Intel Pentim 4 3GHz

| **Cycles/byte for 4096 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 7225,57 | 7244,65 | 8323,44 | BMW256 |
| 9265,59 | 9295,20 | 9305,80 | Shabal256 |
| 14247,87 | 14703,88 | 15155,43 | SHAvite3256 |
| 15829,13 | 16049,01 | 17146,23 | BLAKE256 |
| 16261,50 | 16400,38 | 16416,46 | Skein256 |
| 45044,45 | 47045,31 | 47964,37 | Luffa256 |
| 73869,55 | 74632,52 | 75418,86 | Keccak256 |
| 152272,21 | 153401,62 | 154566,34 | SIMD256 |
| 162051,47 | 166768,41 | 170644,45 | CubeHash256 |
| 211721,88 | 213019,05 | 214382,31 | Fugue256 |
| 259742,44 | 261185,53 | 262626,81 | ECHO256 |
| 341576,27 | 346572,31 | 351576,22 | Grøstl256 |
| 670755,25 | 676688,53 | 682705,99 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

| **Cycles/byte for 1536 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 8801,49 | 8824,82 | 11141,11 | BMW256 |
| 10052,41 | 10232,85 | 10238,78 | Shabal256 |
| 15489,48 | 15521,06 | 17663,36 | SHAvite3256 |
| 15899,37 | 16479,50 | 17219,64 | BLAKE256 |
| 16755,94 | 16937,27 | 17001,17 | Skein256 |
| 48255,04 | 48991,48 | 49225,08 | Luffa256 |
| 78237,43 | 79339,80 | 80538,97 | Keccak256 |
| 156868,67 | 158017,99 | 159768,41 | SIMD256 |
| 192055,46 | 194056,17 | 197254,42 | CubeHash256 |
| 274624,17 | 276213,91 | 278397,26 | Fugue256 |
| 285548,90 | 287631,19 | 289857,88 | ECHO256 |
| 358732,56 | 370554,43 | 381650,82 | Grøstl256 |
| 775918,61 | 779317,67 | 782712,08 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

Table 4: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Cycles/byte for 576 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 12771,16 | 13305,59 | 13318,75 | Shabal256 |
| 13321,63 | 13584,04 | 19580,24 | BMW256 |
| 16950,20 | 17351,48 | 18905,47 | BLAKE256 |
| 17782,80 | 17945,78 | 17970,88 | Skein256 |
| 18376,43 | 18419,88 | 21094,73 | SHAvite3256 |
| 52356,80 | 52413,98 | 52737,47 | Luffa256 |
| 87915,88 | 88491,50 | 91843,89 | Keccak256 |
| 166456,86 | 170636,30 | 173316,46 | SIMD256 |
| 271857,87 | 274563,03 | 279125,11 | CubeHash256 |
| 344922,19 | 350461,26 | 356261,96 | ECHO256 |
| 405246,67 | 419416,73 | 421033,69 | Grøstl256 |
| 438264,92 | 442195,26 | 446376,91 | Fugue256 |
| 819573,33 | 828671,60 | 837601,74 | JH256 |
| N/A | N/A | N/A | Hamsi256 |

| **Cycles/byte for 64 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 31304,13 | 32370,02 | 33474,77 | BLAKE256 |
| 34681,42 | 35358,36 | 35690,64 | Skein256 |
| 48044,74 | 50366,21 | 52670,80 | Shabal256 |
| 59303,28 | 59664,80 | 67866,37 | SHAvite3256 |
| 70075,04 | 70392,20 | 108408,90 | BMW256 |
| 93142,73 | 93634,91 | 93810,89 | Luffa256 |
| 160815,70 | 161055,29 | 169345,14 | Keccak256 |
| 294708,55 | 295054,69 | 313714,86 | SIMD256 |
| 804118,77 | 805950,74 | 832831,25 | ECHO256 |
| 925242,90 | 928087,90 | 932236,31 | Grøstl256 |
| 1293450,33 | 1305829,03 | 1338317,15 | CubeHash256 |
| 2056855,65 | 2104913,31 | 2167669,40 | JH256 |
| 2499328,19 | 2503167,99 | 2567218,87 | Fugue256 |
| N/A | N/A | N/A | Hamsi256 |

Table 5: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Cycles/byte for 8 bytes** | | | |
|---|---|---|---|
| **Quartile** | **Median** | **Quartile** | **Hash** |
| 135025,91 | 137743,47 | 150609,67 | BLAKE256 |
| 203277,62 | 206386,83 | 211322,02 | Skein256 |
| 325306,92 | 351510,28 | 354179,03 | Shabal256 |
| 366775,27 | 369936,17 | 415334,07 | SHAvite3256 |
| 374296,11 | 379263,30 | 384047,28 | Luffa256 |
| 563348,94 | 566022,29 | 879709,11 | BMW256 |
| 1295028,32 | 1296952,13 | 1363139,68 | Keccak256 |
| 2340421,70 | 2342937,10 | 2491367,39 | SIMD256 |
| 4563850,93 | 4568942,48 | 4704900,83 | Grøstl256 |
| 6422366,05 | 6438679,54 | 6652598,15 | ECHO256 |
| 9239108,19 | 9293066,96 | 9536356,32 | CubeHash256 |
| 16458116,05 | 16838963,18 | 17339729,84 | JH256 |
| 18689930,39 | 18710416,85 | 19212191,51 | Fugue256 |
| N/A | N/A | N/A | Hamsi256 |

Table 6: Performance Measurements on Sun SPOT (ARM920T 180MHz)

| **Hash** | **Size in kB** |
|---|---|
| Shabal | 3,65 |
| CubeHash | 4,19 |
| JH | 4,93 |
| Keccak | 5,94 |
| SIMD | 6,13 |
| Luffa | 6,26 |
| Grøstl | 6,36 |
| BMW | 6,86 |
| Skein | 7,10 |
| BLAKE | 7,31 |
| Fugue | 10,06 |
| ECHO | 11,6 |
| SHAVite3 | 17,6 |
| Hamsi | N/A |

Table 7: Cost Measurements

# 5 Conclusions

In this paper, we compared the performance and cost of the remaining SHA-3 candidates on a low-cost ARM920T running the Java based Squawk VM. The results indicate that the computational efficiency of our Java implementations is weak compared to optimized C.

We confirm that software implementations on constrained devices are relatively slow, if not designed properly. Candidates requiring a lot of memory management mechanisms, and with poor reference implementations, can have poor performance on constraint platforms running Java. Some of these are mentioned in the introduction, and was the motivation behind this work.

# References

[1] National Institute of Standards and Technology: “Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition“. Available: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf` (2010/04/10), September 2009.

[2] Eurosmart: “Market Overview“. Available: http://www.eurosmart.com/index.php/publications/market-overview.html (2010/04/27).

[3] Oracle: “Java Card 3.0.1 Platform Specification“. Available: http://java.sun.com/javacard/3.0.1/specs.jsp (2010/04/27).

[4] N. Shaylor, D. N. Simon and W. R. Bush: “A Java virtual machine architecture for very small devices”, 2003. Available: http://www.grundu.net/papers/squawk.pdf

[5] R. Goldman: “Using the AT91 Timer/Counter”, 2007. Available: http://www.sunspotworld.com/docs/AppNotes/TimerCounterAppNote.pdf