Technische Universiteit of Eindhoven
*Department of Computer Science & Mathematics*

# A Diskless Cluster Architecture for Hashing Operations

*by*
Spyridon Antakis

*Supervisor*
Benne de Weger

*Mentor*
Padraic Garvey

*Intel Shannon, Ireland*
*August 2010*

## Acknowledgements

Passion comes within the mind. It is a state of mind which purely define us. It is an unavoidable consequence of our weak nature. There are no definitions or confinements and there should not be. Each individual is a part of a project called *"world"*. The mission is to accomplish personal balance between of a life that comes with choices. Anything else, just belongs to the process. We are carved from our experiences and we are judged from our actions. Shameless and proud for what we are, we should continue. At the moment on which we reach our mental completeness, we know that we arrived to our final destination. I do not need from you to understand, I do not need from you to believe, I just need to accept and proceed. The truth always lies at different levels, mine is inside my passions. Expectations arise and set, I just need to fulfill them. Seizing every single moment of life finally brings me to the ultimate conclusion, *"life is too short to have them all"*, but still is a great walk that I should enjoy. After all, it is the only way to really know myself. As each journey comes to an end, a circle closes and harmony persists. Through a long-run which was meant to shape me, I am rewarded. And now I know, here I stand to pay my respects to my inspirers.

Thank you, for being all along.

Spyridon Antakis

# Preface

The current document constitutes an attempt to investigate the behavior of a cluster based architecture with respect to the performance of several open source hashing implementations. Hashing algorithms find applicability in a majority of cryptographic operations and thus, such a research effort obtains a significant value. By considering also the fact that most of the benchmarked hashing implementations are included in all standard *Linux Distributions*, the presented results have an even more broad meaning. More precisely, by building a *diskless cluster* a number of well-known hashing tools are successfully tested, under a variety of different hardware. An interesting aspect of the entire experiment is the inclusion of a new hashing prototype, called *Skein.* This algorithm was developed from a group of security experts and at the moment, it has been qualified to the pre-final round of *NIST Hashing Competition.* At the end of 2010, this competition will define the final competitors for the standardization of *SHA-3.* Skein is believed to be one of the promising candidates for winning the competition and since its authors reported that is faster from SHA-2, it was considered tempting to be benchmarked and compared to the existing hashing algorithms. However, the benchmarking of this hybrid proposal was based on one of the few available implementations, which is written as a python library. Therefore, there are no significant comparative references and only the sample performance reports that Skein's authors published, represent a possible metric.

The initial inspiration for our research was derived from several different security aspects, but its main root lies on an existing vulnerability of the *Wi-Fi Protected Access (WPA) protocol.* The *Pre-Shared Key (PSK)* mode of this protocol is known to be vulnerable against *off-line dictionary attacks*, nevertheless the intensive hashing computations for performing the attack require an enormous processing power. As it was described in previous *WPA-PSK cryptanalysis attempts*, the concept of pre-computed hashing could be used for optimizing the attack. In practice, there is a vital need for powerful processing capabilities, since only then you could end up with desired results inside a feasible time frame. The technique of *hashing pre-computations* is not something new and an entire methodology for generating pre-computed lookup tables exists for several years now, mostly known as *rainbow tables.* This technique is still able to spare us *computational time* and decrease the *processing workload*, especially when *salting* is omitted.

Motivated purely by the idea of exploiting the scalability of hashing operations, the vision of building a diskless cluster architecture became an interesting task. Even though this document does not conduct experiments with the latest available hardware, it will become obvious during the review of the content that this does not oppose any limitations for using cutting edge processing hardware. The capabilities of the diskless cluster architecture are based on the resources that cluster includes and the restrictions that these resources induce. Thus, there are no specific hardware requirements and nothing stops an architect to transform this basic architecture to a dedicated *crypto-cluster.* As it will be briefly discussed in one of the chapters, todays emerging technology for cryptography focuses on developing embedded mechanisms inside the new generations of processors, that will be able to deliver *cryptographic acceleration* at amazingly high-rates. A combination of the described *diskless cluster architecture* with such *accelerators*, it is strongly believed that could provide an extremely powerful dedicated structure for cryptographic operations. Unfortunately, for reasons that we will briefly discuss under the content of this document, these experiments were not performed and were left as a potential future work. Nevertheless, all the needed details are included and the conclusions for the benefits for such a combination, it is believed that would be feasible to be extracted from the content of this research.

The several sections of this dissertation try to give an in depth insight on the different involved issues, which arise from the proposed architecture. Even if it is almost impossible to exhaust all the existing aspects, our efforts focus on presenting all the details and information that could have a significant value for the research community. The perception of the author with respect to the revealed drawbacks and advantages is discussed and a concrete evaluation on the benchmarked hashing algorithms (e.g. MD5, SHA-1, SHA-2, Tiger, Whirlpool, Skein) is performed. Two basic test cases are applied and through a *statistical analysis* the reader is able to comprehend and highlight all the interesting points, by comparing also the variations of hashing algorithms. At this point it should be mentioned, that this document does not aim to provide extended details on the theoretical nature of hashing algorithms and only a sufficient necessary background is provided. Moreover, it does not attempt to apply any *brute force attacks* on the existing WPA-PSK vulnerability and the case study on WPA-PSK is presented only as an example for the usability that this architecture could have on the field of cryptanalysis. The main objective is to identify and investigate the behavior of the hashing algorithms on the *cluster-based* architecture. The focus remains strictly on the processing of the *computational workload* and the general behavior of the *cluster architecture.* The *high-end* goal is to inspire additional research on the field of *clustering for cryptography* and answer a number of *research questions* with respect to the proposed model.

# Contents

**Acknowledgements** 3

**Preface** 5

**List of Figures** 9

**1 Hashing on a Diskless Cluster** 11
1.1 Hash Function Design Principles 12
1.1.1 Definitions & Security Properties 12
1.1.2 Generic Attacks 13
1.2 Hashing Algorithms 13
1.2.1 MD5: Message Digest Algorithm 5 14
1.2.2 SHA-1: Secure Hash Algorithm 1 14
1.2.3 SHA-2: Secure Hash Algorithm 2 14
1.2.4 Whirlpool 15
1.2.5 Tiger 15
1.3 NIST Competition 15
1.3.1 Skein 15

**2 Areas of Applicability** 17
2.1 Security Protocols 17
2.2 Digital Forensics 18
2.3 Cryptanalysis Tools 19
2.4 A Case Study for Wi-Fi Protected Access (WPA) 20
2.4.1 The Key Hierarchy 20
2.4.2 The 4-Way Handshake 21
2.4.3 Performing the Attack 22
2.4.4 Previous Experiments 22

**3 On the way of Benchmarking** 24
3.1 General Concept 24
3.2 The Puzzle of Hardware 25
3.3 Database 28
3.4 Open Source Hashing Implementations 28
3.5 Shell Scripts 29
3.6 Research Questions 30

**4 Building a Scalable Cluster** 33
4.1 The Diskless Architecture 33
4.2 Clustering Tools 34
4.2.1 Diskless Remote Boot in Linux (DRBL) 34
4.2.2 TORQUE & Maui 36
4.3 Job Scheduling & Scripts 37

**5 Hardware Acceleration for Cryptography** 39
5.1 Intel EP80579 Processor 40
5.2 GPUs vs CPUs 41
5.2.1 Performance vs Cost 42
5.2.2 Uprising Technology 42

**6 Evaluating the Diskless Model** 43
6.1 Image Files 44
6.1.1 Hashing a CD Image 44
6.1.2 Hashing a DVD Image 45
6.1.3 Hashing the Database 46
6.1.4 Cluster vs Units 47
6.2 Dictionary File 48

6.2.1 Hashing 10.200 Words . . . 48
6.2.2 Cluster vs Units . . . 50
6.3 Performance & Diskless Architecture . . . 52
6.3.1 Remote Processing vs Local Processing (Q. 1) . . . 52
6.3.2 Network Latency (Q. 2) . . . 52
6.3.3 Hashing Implementations (Q. 3) . . . 53
6.3.4 Performance of Skein Algorithm (Q. 4) . . . 53
6.3.5 Better Processors (Q. 5, 7) . . . 53
6.3.6 Processing on the Cluster Nodes (Q. 6, 8) . . . 54
6.3.7 Scaling the Diskless Cluster Architecture (Q. 9) . . . 54
6.3.8 Optimizations (Q. 10) . . . 54

**7 Conclusions** . . . 56

**8 Future Work** . . . 57

**References** . . . 58

**A Appendix: Sample Scripts** . . . 61
A.1 Benchmarking scripts . . . 61
A.2 Python script for Skein algorithm . . . 64
A.3 Job Scheduling scripts for Cluster Nodes . . . 65

**B Appendix: Photographs** . . . 66

## List of Figures

1 4-Way Handshake . . . . . 21
2 Description of the Hardware . . . . . 27
3 Database . . . . . 28
4 Hashing Commands . . . . . 29
5 The Cluster Architecture . . . . . 33
6 State of Cluster Nodes . . . . . 36
7 Monitoring with qstat - TORQUE . . . . . 37
8 Monitoring with showq - Maui . . . . . 38
9 High Level Model . . . . . 40
10 Hashing a CD Image (630 MB) . . . . . 44
11 Hashing a DVD Image (4 GB) . . . . . 45
12 Hashing the Database (6,5 GB) . . . . . 46
13 Cluster vs Units (entire database - 6,5 GB) . . . . . 47
14 10.200 Words . . . . . 48
15 Skein - Hashing a single word on Archimedes . . . . . 49
16 Skein - Hashing a single word on Pythagoras . . . . . 49
17 10.200 Words - Zoomed Graph . . . . . 50
18 Cluster vs Units (10.200 words) . . . . . 51
19 Cluster Nodes ( 4 x 2.550 words) . . . . . 51
20 The Diskless Cluster . . . . . 66
21 Cluster Nodes . . . . . 66

# 1 Hashing on a Diskless Cluster

In the field of digital security, hashing has many different usage purposes. The features that a hash is able to deliver constitute a key point for a number of fundamental security scenarios and in combination with the other cryptographic operations, provide the necessary security guarantees. In this research effort the main goal is to propose an alternative cluster architecture for performing *intensive hashing operations.* Of course, the necessary information with respect to the *hash function design principles* are given, however the document focuses on existing implementation scenarios. In today's real-life security applications, there are cases on which *processing power* and *security properties* must be successfully balanced. As a result, in the last years, the *IT Industry* has begun to invest in the design of dedicated cryptographic *hardware accelerators*, which treat the problem of intensive processing at the lowest level.

Inspired by a novel architecture setup, we create a *diskless cluster* from scratch and we benchmark its performance on different hashing suites. In our setup we use quite old processors for the cluster nodes, but we expect that this will not affect directly the evaluation of our model. If someone substitutes the processors that we use in our experiments with hardware that is able to accelerate, then the performance should increase accordingly. The limitations that might arise from the *cluster-based* architecture are considered to be quite independent from the used processors and they are related mainly to the established network connections between the systems. These limitations are the ones on which our investigation is mainly focused and constitute the source for the most of our research questions. The objective is to understand and present the critical points with respect to the performance that this architecture will deliver and at the same time, compare them to the one that a regular architecture approach is able to offer. The specificity of the whole experiment emerges from the fact that the described architecture is *diskless* and only a central hard disk is used as a *storage unit.* Moreover, the operating system is loaded on the fly through the network and all the cluster nodes are controlled by a main board through ethernet interfaces. All these differences increase the possibilities of easy extensibility and make the entire configuration of the nodes quite straightforward. However, they also introduce several additional performance issues, which will be discussed and evaluated under the content of this document.

The conducted experiments are structured to allow us to explore different hardware resources, introducing also a metric of comparison for the various test cases. By benchmarking a majority of popular hashing algorithms, we are able to have a better insight into their performance. More precisely, through the open source hashing implementations which are freely available, we assess the different behavior on a variety of *data size inputs* and identify the distinct *computational cost* that each of them induces. At this task an additional python library is successfully compiled and installed and we obtain an original performance evaluation for the *skein* algorithm. Nevertheless, the reader should keep in mind that this python implementation of skein is an early design attempt and therefore, this might introduce additional performance latencies. The initial intention was to benchmark even more *SHA-3 candidates* from the ones that have already been qualified on the pre-final round of *National Institute of Standards and Technology (NIST)* competition. However, the *lack of suitable software implementations* that could be successfully installed on our cluster architecture, contributed to their *exclusion* from our tests.

In the upcoming sections of *chapter 1*, we present the *characteristics* of hash functions and we give a brief summary for the ongoing *NIST competition.* We provide a basic background for the non-familiar reader and we highlight the *security principles* with which hash functions are designed, identifying also an interesting relation with computational power. In *chapter 2*, an attempt is made to correlate our efforts directly with the security fields and several interesting *areas of applicability* that include intensive hashing operations are presented. Furthermore, a detailed case study for the cryptanalysis of a commonly used wireless security protocol is described, allowing the author to argue about the potentials that the extra computational processing power reveals on this area. After the first introductory sections, *chapter 3* dives directly into the details for the structure of the entire benchmark. An analytical description is given with respect to the *hardware*, the *software* and the *database* which was used. The precise concept of testing is described and the functionality that the written *shell scripts* offered is discussed. Finally, the main research questions are formulated in an attempt to make clear the objectives of our investigation and prepare the reader for the content of the following chapters. In *chapter 4* the steps for building the cluster are described and a short discussion is presented about the general features of the *diskless architecture.* The specific reasons for which this architecture is considered to be easily scalable are stated and the *clustering tools* which were used for the setup are introduced. The whole process of configuration is extensively described, providing in this way an in depth understanding of the functionality of the cluster. In *chapter 5*, the technology of cryptographic hardware accelerators is presented as an interesting alternative for extra processing power and some of the features of the *Intel EP80579 Integrated Processor*, are discussed. *Chapter 6* represents the core of our *data analysis* and the statistical presentation of our benchmark. A discussion about the different studied cases is given, allowing us to argue about the limitations and the directions for improvement that apply to the diskless architecture. A dedicated part of this chapter expresses the thoughts of the author and gives the personal point of view for the research questions which have been formulated in *chapter 3.* The main aim is to inform the reader about the concept of clustering and initiate his critical thinking for further discussion. Finally, *chapters 7 and 8* conclude and summarize the *advantages* and *disadvantages* of the proposed architecture, by stating also ideas for future experiments.

## 1.1 Hash Function Design Principles

Even though we will not describe in depth all the details of hash functions, it is considered important to discuss the fundamental properties and security related issues. The current section will highlight the *basic principles* of modern hash functions and it will give the overall picture of the most common generic attacks [1].

### 1.1.1 Definitions & Security Properties

Below, we briefly recall a standard *mathematical notation* which is used in the cryptographic literature:

- $\{0,1\}^n$ : The set of all binary strings of length $n$.
- $\{0,1\}^*$ : The set of all finite binary strings.
- $A \times B$ : The set of all pairs $(v, w)$, where $v \in A$ and $w \in B$.
- $H : A \mapsto B$ : A function $H$ from set $A$ to set $B$.
- $H : \{0,1\}^* \times \{0,1\}^{\mathcal{L}} \mapsto \{0,1\}^n$ : A function $H$ of two arguments, the first of which is a binary string of arbitrary length, the second is a binary string of length $\mathcal{L}$, returning a binary string of length $n$.

**Hash Function:** Let, $\mathcal{L}$ and $n$ be *positive integers*. We call $f$ a *hash function* with $n$-bit output and $\mathcal{L}$-bit key, if $f$ is a *deterministic* function that takes two inputs, the first of arbitrary length and the second of length $\mathcal{L}$-bit and it outputs a binary string of length $n$.

The *$\mathcal{L}$-bit key* is considered to be known unless indicated differently and most of the time in order to avoid any confusion with cryptographic keys, we refer to this key as *index*. Even if our entire work is using hashing implementations that do not include the index feature, we decided to present a more general definition for the hash function. Of course, this definition is an informal version that serves only the purposes of our document and a more formal definition could be found in [1]. The level of security that a hash function provides, is based on the properties of *one-wayness*, *second-preimage resistance* and *collision resistance.* These 3 main properties are framed on the idea of a *computationally-bounded adversary* and in practice, it is assumed that it is computational *infeasible* for someone to perform the needed calculation required for comprising the security of a hash function. In a more formal way, an *acceptable* hash function design must comply with the following property definitions:

**One Wayness:** A hash function $H$ is *one-way*, if for a random key $k$ and an $n$-bit string $w$, it is hard for the attacker presented with $k$ and $w$ to find $x$ so that $H_k(x) = w$.

**Second-Preimage Resistant:** A hash function $H$ is *second-preimage resistant*, if it is hard for the attacker presented with a random key $k$ and a random string $x$ to find $y \neq x$ so that $H_k(x) = H_k(y)$.

**Collisions Resistant:** A hash function $H$ is *collision resistant*, if it is hard for the attacker presented with a random key $k$ to find $x$ and $y \neq x$ so that $H_k(x) = H_k(y)$.

Collision resistance constitutes the *strongest* property from all three. However, it is also the *hardest* to satisfy and the *easiest* to be broken. As it can be extracted from the previous definitions, collision resistance *implies* second-preimage resistance. On the other hand, the second-preimage resistance and one-wayness are considered to be *incomparable*, in a sense that these properties do not require one another. Although, most of the designs which are *one-way* but not *second-preimage resistance*, are usually quite contrived. Of course, the construction of a secure hash function is not limited only on the satisfaction of the pre-mentioned properties. Several other conditions should be met, such as the *adequate alternation* of bits after flipping a single input bit or the *infeasibility* to reliable guess input bits based on the produced digest. The *non-inclusion* of such additional security features does not necessarily lead into the complete breaking of a hash function, but it introduces doubts about the security guarantees that this function offers. Finally, a secure hash function even though it must be computationally infeasible to be inverted, should still be *easily computable.* The security principles of hashing certainly hide a deeper theory and different aspects must be investigated before any implementation. However, since the intentions of this document does not target the hash function design decisions, the reader is referred to the relevant literature for further details [1] [3].

### 1.1.2 Generic Attacks

As a *generic attack* we can define an attack that applies to every hash function, independently of how well the design principles have been followed. Compared to attacks that exploit specific flaws of a particular design, generic attacks treat all hash functions as black boxes. As it is mentioned in [1], the best known way for *inverting* a hash function and for *finding* a second-preimage under the model of black box, is by performing *brute force attacks*. By taking a closer look at the inversion problem of $H_k(x)$, which states that given $w$ and $k$ find $x$ so that $H_k(x) = w$, the approach of testing arbitrary strings until we get a hit for $w$ could be applied to every hash function. Assuming that a *hash function* should satisfy certain levels of randomness, we could say that for a random function $H$ it would take on average $2^{n-1}$ evaluations for a possible hit, where $n$ is the length of $w$ in bits.

In the field of collision attacks the possibilities of succeeding completely change, due to the *birthday paradox*. According to the paradox, the probability that two people share the same birthday among 23 randomly chosen persons is almost 51%. By applying the same idea to the collision attacks, a careful analysis reveals that in order to achieve probability better than 50% of finding a collision in a hash function with $n$-bit output, it is sufficient to evaluate the function on approximately $1.2 \cdot 2^{n/2}$ randomly chosen inputs. This is equivalent to the formula used for the birthday paradox, which gives us: $\lceil 1.2 \cdot \sqrt{365} \rceil = 23$.

In order for a hash function to satisfy the proper requirements for security, it must maintain the upper bounds for the pre-mentioned properties. Usually it is stated that a hash function has an *ideal security*, if the best attacks known against it, are generic. That in practice means that for the properties of one-wayness and second-preimage resistant the ideal limit is set to $2^{n-1}$ and for the collision resistant is defined (according to birthday paradox) to $1.2 \cdot 2^{n/2}$. Even though for some applications a lower level of security is sufficient to ensure protection, in the theoretical world this is not the case. In the field of cryptanalysis a hash function primitive is considered to be broken if it is shown that it provides security less than ideal and that's the occasion on which a hash algorithm is characterized as *theoretically insecure*.

On the process of *mounting* generic attacks a number of limitations appear and more precisely, the restrictions on hardware capabilities constitute the key factor for preventing any successful attack. As it is stated in [2], a *naive implementation* of the birthday attack would need a storage support for $2^{\frac{n}{2}}$ previously computed elements in a data structure that provides quick access and sophisticated look-ups. In this race for resources, there is a clear *imbalance* between the *cost of memory* and *computational processing*. The CPU cycles are not expensive and the attempt for equivalent memory capacity is at the moment *non-feasible*. Imagine that $2^{40}$ CPU cycles are easily affordable by any commercial processor, but the direct access on $2^{40}$ bytes implies the existence of 1TB Random Access Memory (RAM). Of course, the equivalent capacity of storage can be easily covered by using large hard disks. However, an *unavoidable trade-off* is introduced, due to the required access calls for retrieving data from the hard disk and loading them into the memory. A general remark is that processors are able to process faster than we can provide the input data and this significantly restricts the scaling of the attack [35] [36]. Of course, there are several advanced techniques which try to balance these *trade-offs* and provide optimized ways for performing generic attacks. However, the limitation with respect to the processing speed and the data access remains still present. Moreover, an other important factor with respect to the hardware resources, is the form of the attack. If the attack is not possible to be distributed and it must run sequential, then the speed which an algorithm will yield, is bounded by the frequency of the processor. However, in the case on which the used algorithm can be successfully parallelized, the limitation are set from the *adversary's budget* to provide additional processing units.

At this point even though that our research is not focused on hash function generic attacks, we strongly believe that the proposed architecture could successfully contribute to *parallelizable attacks* and possibly provide an interesting alternative approach for performing such experiments. The main advantage would be the fact that the proposed architecture is easily scalable and configurable and it supports also the *Open Message Passing Interface (Open-MPI)*[1], which is used for parallelization in many commercial cluster architectures. However, the major drawback would still remain on the *cost* for the acquisition of the supplementary hardware.

## 1.2 Hashing Algorithms

Although there are several hashing algorithms, only a few have been adopted by the IT community and are actively used in real-life applications. In our attempt to evaluate a number of the most commonly used algorithms, we test the open source software implementations which have been developed for Linux, trying to identify their behavior on the cluster architecture, but also obtain a comparison among them. More precisely, despite of the fact that some of the algorithms have been successfully proven to be vulnerable to attacks, they still are applied in the real world and thus, are included in our performed benchmark. The popular *MD5*, *SHA-1*, the *SHA-2* family, *Tiger*, *Whirlpool* and the hybrid version of *Skein* algorithm are evaluated. This section provides a general background, discussing mainly

[1]http://www.open-mpi.org/

the levels of security for each algorithm. Based on the aims of our benchmark there is not a strict need for an in depth understanding of each algorithm and therefore, only a general background is provided. Our exploration remains at a quite *high-level* and it is application specific. Moreover, most of our research concerns involve issues relevant to the further optimizations that could be applied in the diskless model and an investigation of the benefits we can obtain by equally dividing a computational workload into the cluster nodes. Thus, the reader is redirected to the appropriate *standards* or *design documents* for further details with respect to the implementation of each hash algorithm.

### 1.2.1 MD5: Message Digest Algorithm 5

*Message Digest Algorithm 5 (MD5)* was introduced by Ronald L. Rivest, in April 1992 [4]. A predecessor called *MD4* had been proposed by the same author in 1990 and MD5 meant to be a *strengthen* version. Both the *produced output* of MD5 and its *internal state* consist of 128 bit. The MD5 *compression function* is using message blocks of 512 bit, which are expanded into *sixteen* 32 bit words. The most important feature of the compression function is the number of performed *rounds*. In the case of MD5, there are *64 rounds* organized in an unbalanced Feistel network each using a 32 bit word to update the internal state via a *non-linear* mix of boolean and arithmetic operations [1]. MD5 is considered as one of the innovative algorithms that influenced the further evolution of hash function designs. However, even though the algorithm was designed by the state of the art security principle of that time, flaws were discovered and today its usage is considered insecure. In particular, the first investigation which was conducted in 1996, revealed vulnerabilities for the compression function of MD5 and from 2004 a number of new attacks emerged. The existing complexity for mounting the collision attacks was decreased with optimizations that several researchers contributed [5] [6] [7] [8] and it was practically proved that MD5 algorithm is obsolete and it *must not* be used anymore in applications were collision resistance is of importance.

### 1.2.2 SHA-1: Secure Hash Algorithm 1

*Secure Hash Algorithm 1 (SHA-1)* was published by National Institute of Standards and Technology (NIST), in April 1995 [12]. The initial appearance of the SHA algorithm was made in an earlier submitted draft, in May 1993 [11]. However, due to a discovered vulnerability this algorithm version (aka SHA-0) was revised by NIST, in order to correct the flaw that reduced the level of security. The designers of SHA-1 borrowed some features from MD5, such as the same *padding algorithm* that breaks the message into 512 bit blocks. The SHA-1 algorithm produces a 160-bit output length and it uses the same number of bits for its internal state. It operates with *80 rounds* and it implements a more complex procedure for message expansion. Furthermore, the alternation of a single bit on the input message impose changes on almost half of the sub-blocks and that's something that was missing from the design of *MD5*. With respect to security, SHA-1 is considered to be theoretically insecure. In 2004, *collision attacks* were presented for the original version of *SHA-0* [14] and in later attempts that took place in 2005 [15] [16], SHA-0 was completely broken. In this way, the confidence for the security of SHA-1 was influenced and initiated further investigations for SHA-1. Even though that collisions has not yet been found for SHA-1, the provided *complexity* for finding collisions have been decreased below the *ideal security* level [17]. Therefore, the usage of SHA-1 is not recommended anymore and the transition to SHA-2 is proposed by NIST.

### 1.2.3 SHA-2: Secure Hash Algorithm 2

The family of *Secure Hash Algorithm 2 (SHA-2)*, was officially published as a standard by National Institute of Standards and Technology (NIST), in August 2002 [13]. Each of the included variation of the algorithm was named based on its digest bit-length, as follows: *SHA-224*, *SHA-256*, *SHA-384*, *SHA-512* and collectively all the algorithms are referred to as *SHA-2*. The SHA-256 function and the SHA-224, which was defined later in 2004 as a truncated version of SHA-256, use a *512 bit* block size for each message and operate at *64 rounds*. On the other hand, SHA-512 but also SHA-384 which is a truncated version, use a *1024 bit* block size for each message and operate at *80 rounds*. In reality, SHA-256 and SHA-512 have similar designs that are based on *32 bit* words and *64 bit* words, respectively. They make use of different shift amounts and additive constants, but their structure is almost identical. These algorithms bear also a yet strong resemblance to *SHA-1*, however they introduce a new message schedule and a higher security level. The SHA-2 family is not widely used and SHA-1 still remains popular, even if it is a matter of time to be deprecated due to its vulnerabilities. There are not any known *collision attacks* for SHA-2 algorithms and the intended level of their security is maintained. At the moment, it seems that the industry is waiting for the standardization of *SHA-3*, but it tries also to *migrate* as fast as possible to SHA-2. However, the transition to SHA-2 is not an easy task, since the majority of the existing security protocols and applications are using either SHA-1 *or* MD5.

### 1.2.4 Whirlpool

*Whirlpool* was introduced by Paulo Barreto and Vincent Rijmen in 2000 and it became a standard by the International Organization for Standardization (ISO) and the International Electro technical Commission (IEC) as part of the joint *ISO/IEC 10118-3*, in February 2004 [18]. Although Whirlpool is not focused on any particular architecture (32 bit, 64 bit processors), it is quite flexible in optimizations. It uses a block cipher which is very similar to the *Advanced Encryption Standard (AES)* algorithm and it operates on messages less than $2^{256}$ bit in length. Whirlpool outputs a *512 bit* digest and performs *10 rounds*. In 2009, an attack was presented for the case of reduced whirlpool [19]. However, at the moment there are not any serious *security weaknesses* and Whirlpool is considered to be quite reliable. Nevertheless, it seems to have a limited popularity in the commercial areas of security.

### 1.2.5 Tiger

*Tiger* was designed by Ross Anderson and Eli Biham in 1995 [20], however has not been officially standardized. The authors targeted mainly to increase the efficiency of tiger for the *64-bit* architecture, but also maintain the performance for the *32 bit*. All the computations are on *64 bit* words and each *512 bit* message block is divided into *eight* 64 bit words, producing a *192 bit*, *160 bit* or *128 bit* digest. There are 24 rounds, using a combination of operations such as *mixing with XOR*, *additions*, *subtractions*, *rotations*, *S-box lookups* and a fairly intricate *key scheduling algorithm* for deriving 24 round keys from the 8 input words. Tiger is fast in *software*, however the implementations in *hardware* or *small micro-controllers* are difficult due the S-box size. With respect to security, Tiger remains still secure and until now, only *pseudo-collisions* and *preimages attacks* on a reduced version have been discovered. [21] [22] Today, it is mainly used in the form of a hash tree and despite the fact that it has been in existence for almost 1.5 decade, it does not seem to constitute a first option for security implementations.

## 1.3 NIST Competition

In November 2007, the *National Institute of Standards and Technology (NIST)* initiated a public hash function design competition for defining a standard for *SHA-3*. The idea is similar to the one used for defining the *Advanced Encryption Standard (AES)* and the main objective is to replace SHA-1 and SHA-2. The first round has already been completed and *14 candidates* were successfully qualified to the *pre-final* round. The results for this round will be announced at the end of 2010 and the winners will continue in a final round. As it is stated by NIST, a new hash design will be published as the *SHA-3 standard* in 2012. For the majority of the 14 active candidates there are not many *available implementations* and only the original source code which was submitted in NIST, constituted a possible solution for our benchmark. However, the difficulties that we faced for compiling and successfully embedding these implementations into our cluster architecture, contributed to their *non-inclusion*, with the exception of the case of the *Skein* algorithm, for which we were able to find an implementation in a *python library* (PySkein 0.6.1) that allowed us to run the exact same tests. However, the *accuracy* and *trustworthiness* of the performance for this skein implementation is under evaluation, in a sense that the python library which we use, represents just an early attempt for implementing Skein algorithm. The library is still under development and has not been extensively tested, introducing a level of uncertainty with respect to performance. Therefore, we remain quite skeptical about the benchmark of the *python skein scripts*, compared to the authors' *optimized C* testing.

### 1.3.1 Skein

*Skein* is a hash function which was designed by Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas and Jesse Walker [23]. It is efficient on a variety of platforms and performs really well both in hardware and software implementations. The expectation is that this proposal would be able to *substitute* the complete SHA family of hash functions, since it provides similar and more advanced features. The authors submitted also an additional document containing a number of *provable secure properties* [24], increasing significantly the confidence in the algorithm. The primary proposal for using Skein is the *512-bit* version. However, it is feasible to define three different *internal state* sizes of *256-bit*, *512-bit* and *1024-bit*, with any output size.

Skein was tested under the *32 bit* and *64 bit* architecture for different message sizes, by using *assembly* language and *optimized C*. The complete results of the performance evaluation are stated in the design document [23] and in an overall assessment we can say that Skein was shown to be competitively fast. The benchmark was performed using a *cycles/byte* metric and the presented numbers at some cases yielded a better speed than the one of SHA-2 algorithms. In particular, the designers propose *Skein-512* as the best possible solution, since it is proved *slightly faster* than the equivalent SHA-512 and SHA-384. On the other hand, *Skein-1024* is almost *2 times slower* than SHA-512, but this is considered normal since the input block size for *Skein-1024* is double. Finally, *Skein-256* seems to be *1.5 times slower* than the equivalent SHA-224 and SHA-256.

With respect to security, at the moment there are not any known vulnerabilities for Skein and that makes it arguably a serious candidate for selection as SHA-3. In 2009, Jean-Philippe Aumasson *et al.* investigated the cryptanalysis of *Skein* algorithm and even if they could not directly extend an attack to the full Skein hash, they proved that a *minimum of 36 rounds* is required to validate the security proofs on Skein. The rounds of Skein used in the proposal document are *72* for the 256-bit and 512-bit version and *80* for the 1024-bit. Moreover, it is mentioned that the number of rounds it is possible to be modified without many changes, allowing in this way a certain flexibility to the implementation design of the algorithm.

# 2 Areas of Applicability

*Hash functions* find applicability in several real systems and in many security implementation the usage varies. As it is already known *public key cryptography* is usually the operation that introduces the most noticeable computational intensiveness. However, there are specific occasions on which the processing time for performing a hash computation is also important. Mostly, such a need appears on situations in which *cryptanalysis attempts* are applied or in cases that the *input data* for hashing have an enormous size. Security protocols treat hashing functions in a normal processing scale and thus, we do not consider our architecture an optimal solution for the hashing operations that occur in these protocols. Nevertheless, since the inclusion of hardware acceleration could deliver operations for all the cryptographic computations, it is argued that this feature could potentially help *off-load* the expensive operations of public key cryptography. Even if our research is focused only on hashing implementations, that does not necessarily restrict our cluster approach. Optimizations in the general area of cryptography could be obtained and some ideas for specific scenarios are described in section 2.1.

It is true that the latest experiments on *Graphics Processor Units (GPUs)* yielded amazing rates for intensive hashing computations, which are really hard to be beaten at the same *cost*, if not infeasible. Nevertheless, GPUs are specially programmed in order to perform only a simple hashing operation and they are not able to extend the usage options easily. In some occasions that is acceptable, but in some others further *flexibility* is required. The factor that makes our approach interesting is the *freedom* that provides to the implementers and the different opportunities for using hardware acceleration. We believe that at least for the *cluster structure* the limitations are defined strictly from the *economical cost* and in any other case, the complexity is left to the architect's imagination. For example, nothing stops an architect to install and use a number of GPUs at each cluster node or install cryptographic accelerators for delivering availability for the entire family of cryptographic operations.

The following sections will present a number of major areas in which hash functions appear. The intention is to provide an overall picture on where our cluster architecture could be adopted, highlighting the purposes that could successfully serve at real scenarios. We discuss the usage of hash functions in two commonly used *security protocols*, describing usability scenarios that could benefit from our proposed architecture. Next, the field of *digital forensics* is presented, demonstrating a clear need for additional computational processing power for hashing implementations. Finally, the popular area of *cryptanalysis tools* is explored, in an attempt to show that the cluster architecture could lead to additional ways for evaluating security. A concrete example (see 2.4) is analyzed in the area of wireless security protocols and it is briefly pointed out, how the *faster processing* could help exploit existing security flaws.

## 2.1 Security Protocols

*Transport Layer Security (TLS)* [26] and *Internet Protocol Security (IPSec)* [27] find applicability in very popular services and implementations and thus, their performance becomes an important requirement. They underline almost every secure *internet communication* by using cryptographic operations and that includes a computational cost. With respect to used hash functions, which according to their specifications would be the *MD5* and *SHA-1* algorithms, there are not noticeable performance issues. The hashing operations appear mostly as means of *authentication* and as a useful mechanism for the *initial key derivation.* The intensiveness that these hash operations introduce, is not major compared to the public key cryptography. Thus, we believe that the usage of our cluster model could not lead to any significant optimizations strictly only for hashing. If we investigate the case of these security protocols in a broader sense, we will realize that optimizing the speed on which *public key cryptographic operations* can be completed, maintains the most valid meaning for performance. TLS constitutes the classic way for securing *Web Transactions* and IPSec usually appears at the popular *Virtual Private Networks (VPNs).* From a client's perspective accelerating the operations of encryption and decryption does not seem to be a major concern. However, on the server's side cryptographic acceleration obtains a totally different meaning, since it could significantly increase the number of connections that are server could accept at the same time.

For example, consider the secure web connections that exist in many commercial websites. These connections are created with the usage of the TLS protocol, which aims to ensure the data transmission by adding *encryption*, but also provide a necessary *server authentication.* More precisely, each secure connection is initiated from a *TLS Handshake* between a client (e.g. web browser) and a web server. In the phase of the handshake, the cipher specifications are agreed and a certificate exchange occurs, accompanied with an authentication (optional for the client's side). Public key cryptography (e.g. RSA) is used on both sides, in order to establish a common symmetric key between the parties and start the encrypted data transmission. The most expensive part of computations lies in the public key cryptography that appears on this handshake and even if it does not seem so severe, this does not mean it cannot be further optimized. The designers of the protocol already predicted a possible performance drawback and they introduced a *session-id* re-usage. In the option of re-using a session-id the TLS handshake is avoided and the previously cryptographic keys that have been established for the specific session-id are used. Both client and server store the *sessions-id* in a cache memory and if they have not been expired based on *pre-defined timeouts*, the cryptographic keys related to these session-ids are successfully reused. The critical point is that these timeouts cannot be set to expire on a long time period, since we would decrease the provided level of security. Usually, timeouts are not more than some minutes long and that brings us again to renegotiation of the TLS handshake.

Suppose now the simple example of a bank's web server that receives thousands of secure connections simultaneously. Defining long timeouts is not possible due to the existing trade-off with security and at the same time, the intention is the bank's server to be able to serve as many connections as possible. Therefore, by using an architecture that could successfully *off-load* the public key computations by using hardware acceleration, the optimizations become meaningful and they add a significant value to overall performance of the protocol. At this point we should mention that since the nature of the handshake is *synchronous*, the overall time for completing the handshake would not change if the client does not support also acceleration. However, an accelerated web server in the same time frame would be able to process more requests and this in combination with the chance to accelerate also the symmetric cryptographic operations, makes the entire effort attractive. In particular, for the TLS Handshake and the speed of secure web transactions there are several interesting surveys [28] [29], that reveal an existing need for a better handling of expensive TLS computations.

For the case of *IPSec* and more precisely for the *VPNs* the objectives remains similar, even though that public key cryptography appears in a decreased rate. The traditional concept of establishing the connection first with the usage of *public key cryptography* (e.g. RSA) and then securely exchange a *symmetric key* for encrypting the transmitted data, finds applicability also in VPNs. Nevertheless, these connections could remain open for several hours without renegotiations that use public key cryptography and therefore, acceleration could only target on the symmetric cryptographic operations. This still hides a valid computational interest, but in the sense of *throughput.* The key point this time lies on the fact that in a VPN connection the communicated amount of data is significantly larger. Even if symmetric cryptography is faster, accelerating this kind of operations could still provide more connections per VPN server, as indicated in [30].

## 2.2 Digital Forensics

*Digital Forensics* refers to official investigations that are conducted aiming to present digital proofs for illegal actions that appear mostly in cyberspace. These investigations are part of the information security world and one of the fundamental tools they use for performing data analysis, is hashing operations. The concept is quite straightforward and it is based on the integrity features that a hash value offers. In a forensic examination of file systems, the hashing implementations are able to efficiently answer questions such as *data equality.* As it is mentioned in [33], the standard procedure that is followed in a forensic analysis is the creation of a targeted image for which the entire hash will be computed and recorded. Afterwards, this hash value is used in order to demonstrate the *non-alternation* of the working copy and verify the validity of the entire investigation. Most of the time, this process includes the storing of *finer-grain* hashes for a more in depth investigation, which typically targets at the block level. Another interesting approach is the filtering of known files from the examination process based on pre-computed libraries. An example of such library, is the extensive *reference library*[2] that NIST maintains with more than 50 million *MD5* and *SHA-1* hashes. These hashes have been computed over commonly used files that exist in operating systems and application packages. During a performed investigation, the hashes of the examined files are computed and compared to the ones contained in the library. All the successful matches are discarded as files of non-interest and the remaining files constitute the suspicious candidates, which will be carefully examined. *Intrusion Detection Systems (IDS)* and *Virus Signatures* constitute another case for which hashes provide the ability to perform successful integrity checks and verify the validity of the processed data. As it becomes obvious, the processing capabilities and the time that takes to deliver a hash operation, plays again a really important role. There is a number of special tools which are meant to help the forensic specialists. The simple *dd* linux command which is used for partitioning was modified by the *U.S. Department of Defense Computer Forensics Lab*, resulting into a powerful tool called *dcfldd*[3]. This tool offers a number of useful features that are really handy for the forensics analysts, such as the *on the fly* data hashing.

Except from the traditional cryptographic way of hashing, the idea of *fuzzy hashing* [34] is introduced in the area of forensic, in an attempt to minimize the false identifications during the performed examinations. The main problem is that regular hash implementations produce a totally different output, even if a single bit is changed at the given input. Sometimes this is desired, however imagine the example of two documents which in practice are identical and they differ just in only one character. This will yield a *non-association* between the two documents and will increase the population of files that investigators should examine. The concept of fuzzy hashing is based on the *traditional*

[2]http://www.nsrl.nist.gov
[3]http://dcfldd.sourceforge.net/

*hashes*, but it functions in *segments.* In a *high-level description*, a file depending on its size is fragmented in sections which will be hashed using traditional hashes and the results will be joined together for comparative purposes. A pre-processing takes place before fuzzy hashing is applied that aims to identify the trigger values based on which the segmentation will be performed. The strength of a fuzzy hash lies on its ability to locate similar files and match altered documents, such as *multiple* or *incremental* versions of the same document. A successful implementation of fuzzy hashing is a tool called *ssdeep*[4], which was designed based on the idea of a spam detector called *spamsum* developed by Dr. Andrew Trigdell[5]. Of course, fuzzy hashing does not come without drawbacks and even if text documents constitute an excellent candidate for processing, *special* and *complex* formats are failing to be matched.

The reason that we described the technique of fuzzy hashing is that it takes a *non-negligible* processing time to compute the sets of necessary information. In comparison to *regular MD5 hashes* the performance may be decreased by *7* to *10 times*, if fuzzy hashing is being used. Even if we do not conduct tests for *fuzzy hashing* in our cluster architecture, it is more than sure that our set up could optimize the processing time. In order to demonstrate a valid test case for the field of forensics, we use normal hashing implementations and we hash a database of image files. In this way, we are able to evaluate our architecture for a workload of large data input and argue about its capability deliver *faster* the desired results. The basic factor which comes under assessment into our cluster-based approach is the fact that these images files will be transfered and hashed *on the fly* through the network interfaces, making our experiment even more interesting to be analyzed from a performance perspective.

## 2.3 Cryptanalysis Tools

The area of *cryptanalysis* includes a variety of tools, which are commonly referred to as *cracking* or *password recovery* tools. The majority of these tools requires intensive hashing operations to be performed during the process of cryptanalysis. The needed computations constitute once again the key point for optimizing the performance of such tools and in practice an unavoidable connection with processing power exists. According to the processing capabilities that a system offers, the cryptanalysis attempts could be boosted to the maximum. Passwords have to be stored in a secure form and thus, instead of storing the password in *plain text* the hash value of the password is usually computed and stored. This makes *reversibility* impossible and it nominates *exhaustive search* as the only mean of attack. The main concept behind most of the cracking tools is to recover the secret password through *brute force attacks.* Of course, the attack and respectively the design of the cracking tool, is based always on the targeted data. Nevertheless, the basic techniques remain similar for most of the cases and the existence of *pre-computations* or the inclusion of *sophisticated word dictionaries*, maintain usually a valid role on such attempts.

A very distinct example is the one of the *LM Hashes*, which are implemented on the most popular *Windows OS* distributions. LM hashing algorithm was designed by Microsoft, in order to securely store user passwords. However, the algorithm included several weaknesses that were easily exploitable and forced Microsoft to admit the serious vulnerability by advising system administrators not to use LM hashes for password storage. At the moment, there are several cracking tools which were specifically developed for mounting successful attacks on windows platforms. Imagine that LM hashes are restricted only to the set of ASCII characters, passwords longer than 7 characters are divided into two parts and a separate LM hash is computed for each fragment, all the characters are converted into uppercases before the computation of the hash value and the hashes are not salted, making the successful usage of rainbow tables feasible. *Ophcrack* [6] is probably the best representer for recovering Windows passwords. The idea behind the design of ophcrack lies on the usage of *rainbow tables* and in a *time-memory trade-off* analysis [36] which was published by Philippe Oechslin, one of the original developers of *Ophcrack.* The wrong design decisions that Microsoft made for LM hashes, transformed the difficult task of building a Windows password cracker quite straightforward and inspired also additional programmer to develop variations of such cryptanalysis tools[7]. *RainbowCrack*[8] is another program developed by Zhu Shuanglei, targeting on the generation of rainbow tables that could be used in password cracking.

The list of tools that were designed to perform *cryptanalytic operations* is rather long and their detailed presentation is considered out of the scope of this document. The intention is to give to reader a flavor on the usability that the cluster can obtain in the field of cryptanalysis and argue about the efficiency that such tools can yield in our proposed architecture. In order to have more valid conclusion we perform a general test case by using a *dictionary file* and we hash the contained words with regular hash implementations, trying to investigate if there is any benefits for *balancing* the workload into the different cluster nodes. In an effort to make more clear the general concept of

[4]http://ssdeep.sourceforge.net/
[5]http://samba.org/ tridge/
[6]http://ophcrack.sourceforge.net/
[7]http://www.oxid.it/cain.html
[8]http://project-rainbowcrack.com/

cryptanalysis and to show the direct relation with the hashing operations, in section 2.4 we present a studied case for a wireless security protocol and we demonstrate how it could be compromised by using specific *cracking tool designs*.

## 2.4 A Case Study for Wi-Fi Protected Access (WPA)

Wi-Fi Protected Access (WPA) is one of the most popular security protocols for wireless data encryption [37]. As is already known, WPA-PSK which represents the *personal mode* of WPA is proved to be vulnerable to dictionary attacks [39] [41]. The fact that the initial *4-way handshake* which is performed between the *Access Point* and the *Client* is transmitted unencrypted, gives to the attacker all the needed information in order to start an off-line attack. Until now, there are several groups and companies that investigated this brute force attack and tried to optimize the entire process. The usage of hardware with additional computational power, such as the low-level *Field Programmable Gate Arrays (FPGAs)* [40] or the in parallel processing power that *Graphics Processor Units (GPUs)*[9] offer, revealed interesting approaches that could be followed with respect to the *hashing pre-computations* and the *key searching.* The idea of *rainbow tables* goes back in time [35], however it constitutes an efficient technique to significantly optimize time related constraints. More precisely, in the case of WPA-PSK these tables will be able to decrease the computational time and the required processing power. The *4096 HMAC-SHA1* iterations that are needed for the generation of the *Pairwise Master Key* could be performed beforehand.

Of course, the level of security that WPA-PSK offers is still relatively high for today's hardware capabilities. The population of all possible keys is salted by using the *Service Set Identifier (SSID)* of the wireless network and that makes the already difficult task of applying dictionary attacks, even harder. Imagine, that if someone wants to try every single password, a simple glance would reveal that computations needed are impossible to be performed in a logical time frame. There are *95 printable* ASCII characters and the pass-phrase could be 8-63 characters, covering only the case of one SSID. That leave us with something like $95^{63} \approx 2^{414}$ possible keys for a single SSID test and only for the case of the 63 characters pass-phrase.

Even if breaking completely the security of WPA-PSK is not feasible today, the form of the attack could be further optimized. As always, the technique of *off-line dictionary attacks* is based on the fact that some basic assumptions have high-probability to occur in a real-life scenario. In our case, the usage of a password more that *20 characters* can be considered rather unusual and thus the targeting population of keys can be significantly decreased. Moreover, the attack can use *specific captured SSIDs* or the *manufacturers' default SSIDs* and in combination with *cryptographic acceleration* and our *cluster-based architecture* a customized attempt for compromising security could be applied. The following sections gives a short overview of the attack and the exploitation of the actual vulnerability, by highlighting at the same time the points on which efficient hashing operations would be necessary.

### 2.4.1 The Key Hierarchy

A variety of different keys is generated and used within the WPA protocol family. There are *pairwise keys* which protect the transmitted normal traffic and *group keys* which protect broadband or multicast messages used for network association [42]. Our interest is focused on the top of the hierarchy of pairwise keys and more precisely, on the *Pairwise Master Key (PMK).* This master key is the start of the chain for generating a common cryptographic key for securing the connection between the involved parties. In WPA-PSK, the PMK is computed with the usage of a static *Pre-Shared Key (PSK)*, which is pre-defined between the *Access Point (AP)* and the *Client.* A method called *Password-Based Key Derivation Function (PBKDF2)* is used and a *256-bit master key* is generated. This function constitutes part of the *Public Key Cryptography Standards (PKCS#5 v2.0)* [38] and it takes as input parameters the *Pre-Shared Key (PSK)*, the *SSID*, the *SSID Length*, the *number of iterations* to be performed from the used hashing function and the *desired length* for the produced key. The function prototype is the following:

```
PMK = PBKDF2_SHA1(PSK, SSID, SSID_Length, Num_of_Iterations, Output_Bits) (1)
```

The created PMK is used to perform the authentication, but at the same time contributes to the generation of the *Pairwise Transient Key (PTK).* From a computational point of view, PBKDF2_SHA1 introduces multiple iterations and thus, it decreases the *efficiency* of guessing attacks. However, this function does not provide the final key used for securing the connection and it just generates an input key for another function. More precisely, a *Pseudo-Random Function (PRF)* defined in the 802.11i standard, computes a *transient key* every time a new connection is established. The *PMK* becomes an input parameter and in addition with the *MAC addresses* of the client and the access point, the *nonces* that these two parties will exchange and a *constant string*, the *Pairwise Transient Key* is generated. The inclusion of the *"Pairwise key expansion"* phrase is used by the authors of the standard, in order to cover the rare case of an identical output due to the same input parameters. The *PRM function family* is used elsewhere in the

[9]http://code.google.com/p/pyrit/

standard and in this way, by adding a static string they avoid any unwanted behaviors. The function prototype is the following:

```
PTK = SHA1_PRF(PMK, "Pairwise key expansion", AP_MAC, C_MAC, AP_Nonce, C_Nonce) (2)
```

After the creation of PTK has been completed, all the needed information for securing the wireless communication is contained inside its bits. Depending always on the chosen algorithm for data encryption (TKIP or AES-CCMP), different parts of the PTK will be used as encryption keys for the normal traffic, as encryption keys for the group keys and as keys for authentication. Therefore, it is quite obvious that if this information could be reproduced by an attacker the security of the wireless network will be completely compromised. The PTK is generated at the side of each party with the usage of the PMK, which respectively depends on the secret *Pre-Shared Key (PSK)* and thus, remains unknown to thirds parties. If we examine the required computations we will realize that if an attacker has access to adequate processing power, he could pre-generate a database of PMKs by calling the PBKDF2_SHA1 with input parameters retrieved from a dictionary. In this way, he creates the input needed for function of SHA1_PRF, for which the intensiveness is much less, since there are no iterations and the input is not more than *2 message blocks* of SHA-1 (1024 bits). Then, as the following section will present in detail, in order an attacker to take advantage of the existing WPA-PSK vulnerability, he needs only to identify the encryption algorithm and perform one additional hash operation by using part of the PTK key. However, this operation does not constitute an expensive computation, since again the input data represent a small number of *message blocks* as they refer to one of the transmitted packets of the 4-way handshake.

### 2.4.2 The 4-Way Handshake

The first step before any data transmission takes place between the *access point* and the *clients*, is the *4-Way Handshake*. This handshake will allow both parties to exchange information in order to compute the same *Pairwise Transient Key (PTK)* and moreover, perform a mutual authentication.

```
              +----------+                    +-----------------+
              |  Client  |                    |   Access Point  |
              +-----+----+                    +--------+--------+
                    |         /* A_nonce */            |
                    |<---------------------------------+
                    |                                  |
/* Computes PTK */  |       /* C_nonce, MIC */         |
                    +--------------------------------->| /* Computes PTK */
                    |                                  |
                    |    /* Install the key, MIC */    | /* Checks MIC */
/* Checks MIC */    |<---------------------------------+
                    |                                  |
                    |      /* Key Installed, MIC */    |
                    +--------------------------------->| /* Checks MIC */
                    |                                  |
                    |   /* Encrypting Data Started */  |
                    <---------------------------------->
```

Figure 1: 4-Way Handshake

As it is shown in the above figure, the access point will initiate the handshake by sending the *A_nonce*. The client on the successful reception of the A_nonce, will have all the needed information (e.g. MAC addresses, pre-shared key and nonces) and it will compute the PTK. After that, the client will reply by sending the *C_nonce* used to compute the PTK and a *Message Integrity Code (MIC)* for the transmitted packet, computed with the usage of PTK. Then, the access point will receive the C_nonce, it will compute also the PTK (which should be identical) and will verify that indeed the arrived MIC code is correct by *re-computing* the MIC value and *comparing* it with the one that has received. Finally, the access point will inform the client to install the computed key and the client will reply back acknowledging the message. These two last messages include also a MIC value, created with the usage of the PTK and based on the transmitted packets. At this point, the communicated data could start to be encrypted by using the common PTK.

The described message exchange that occurs on the 4-way handshake might seem secure, however the fact that it is not encrypted gives to the attacker all the required information to structure a successful *dictionary attack*. By eavesdropping the 4-way handshake, an attacker becomes holder of the exchanged *nonces*, the *MAC addresses*, and the network *SSID*. Thus, it only needs the *secret pass-phrase (i.e. PSK)* in order to generate the identical PTK key that the two parties possess. By using a dictionary the attacker is able to compute different possible PTK keys, however he still misses a way to verify that the generated PTK is indeed the one used by the two parties. The key point for overcoming the last security countermeasure, is found in the attached *Message Integrity Code (MIC)*. This value is computed by the usage of the PTK key and the transmitted packet. More precisely, it constitutes the output of one the following function prototypes:

```
MIC_TKIP = HMAC_MD5(key, 16, data) (3)

MIC_AES = HMAC_SHA1(key, 16, data) (4)
```

Therefore, one of the *captured* MIC values can be now compared with a MIC value that is computed with a *possible PTK* and the equivalent packet. If the two MIC values are the same, then the PSK *must be* the same and the password is found.

### 2.4.3 Performing the Attack

The process of exploiting the existing vulnerability on the WPA-PSK protocol, has a major computational drawback which induces serious time limitations. The *4096 SHA-1 iterations* which are needed in order to calculate a single possible *Pairwise Master Key (PMK)* for only one password, block the scaling of the attack. In practice, the intensive repeated hashing operation transforms the attack into such a slow procedure, which has no major chances of success at a realistic time frame. Considering also the fact that the computation of PMK is salted with the SSID of the network, that means that each pass-phrase generates a different PMK for a different SSID and thus, the targeting population of PMKs is extremely increased.

In order to overcome this computational drawback the *pre-computed lookup tables* appear to be an optimal solution. By generating beforehand PMK tables based on dictionaries and a list of potential SSIDs the attacker is earning a significant amount of time during the attack. Instead of performing 4096 hashing operations *on the fly*, he uses the already pre-calculated table of PMKs and feeds them as input parameter to the function that will create the PTK. In this way, by maintaining a large database of PMKs he is able to mount an attack on different networks, without having to perform again the needed computations. All he needs, is a complete captured handshake and then he could start a classic *brute force attack*. Of course the success of such an attack is purely based on the content of the database and there are no guarantees. However, the intensive computations are successfully *off-loaded* at a previous stage and this is something that has a great impact for optimizing the attack.

As the upcoming sections of this document will argue, the usage of a diskless cluster-based architecture could successfully scale the generation of such pre-computed tables and give an advantage to the attacker. The most interesting aspect of using this architecture is that it encloses a kind of general hardware independence and therefore, scaling the computations is mainly a matter of cost. In practice, there are different and more straightforward ways to generate the pre-computed tables and probably much more efficient. However, the extensibility that this architecture demonstrates, provides many pathways to the architect. Different directions could be followed and by including the most *optimal resources* for the desired cryptographic task, much better performance results could be achieved.

### 2.4.4 Previous Experiments

Several groups have investigated the *WPA-PSK attack* in depth. By using dedicated hardware for pre-computing WPA-PSK lookup tables and by developing open source software applications, they took advantage of the existing vulnerability to the maximum. *Renderlab*[10] is considered as one of the initiators for the attempts that followed. This unofficial high-skilled group succeeded to create *33 Gigabytes* of WPA-PSK lookup tables[11], involving 1.000.000 words associated with the 1.000 most commonly used SSIDs. According to the group, the tables were created by the usage of *Field Programmable Arrays* and the computational process that could take months, took only *3 days*. A tool called *CoWPAtty*[12] was developed and extended by *Joshua Wright*, in order to successfully mount the attack by using the pre-computed 33 Gigabytes of PMKs.

In an attempt to optimize the attack even more, the alternative approach of using *Graphics Processor Units (GPUs)* was introduced. An open source software called *pyrit*[13] was developed by *Lukas Lueg* and demonstrated

[10]http://www.renderlab.net/projects/WPA-tables/
[11]http://rainbowtables.shmoo.com/
[12]http://www.willhackforsushi.com/Cowpatty.html
[13]http://code.google.com/p/pyrit/

how it is possible to parallelize the processing of cryptographic operation on GPUs. The performance that this implementation yielded, was more than impressive. The achieved numbers revealed *10 times* faster processing than today's latest processors and in reality, left as only competitors to this task the dedicated *cryptographic accelerators*. As the author commented,

*"Every Pairwise Master Key is "worth" exactly 1 MB of data getting pushed through PBKDF2-HMAC-SHA1. In turn, computing 10.000 PMKs per second is equivalent to hashing 9.8 GB of data with SHA1 in one second."*

Imagine, that a combination of *4 GPUs* can reach a number of *89.000 PMKs per second*. Therefore, there is not any doubt that the *Graphics Processors* provide a very powerful hardware approach for the intensive hashing operations. Finally as a proof of a concept, in 2008 a Russian security company named *Elcomsoft*[14] created a cracking suite called *Distributed Password Recovery*. By using a number of GPUs as the main core of the suite, the company concluded that the security of WPA-PSK will soon become obsolete.

[14]http://www.elcomsoft.com/edpr.html

# 3 On the way of Benchmarking

This chapter provides all the required details for the *hardware*, the *software* and the *data* which were used for setting up the performed benchmark. Our intention is to enhance the ability of the reader to comprehend the upcoming sections and also present the precise actions which contributed to the steps for building the cluster architecture. Hereby, we give a *high-level description* of our entire effort, providing also a better insight in the structure which was followed. The idea of using hashing implementations was derived from the major areas in which hashes find applicability and mostly from the intensive computations which are required in some special cases, as previously described. We believe that the usage of a diskless cluster architecture will offer an alternative solution and it will bring to the surface several other interesting research opportunities. In section 3.1 we describe the general concept of our entire experiment and in section 3.2 we give the precise description of the hardware. Section 3.3 denotes the exact database of files which was used and the different reasons for which these files were chosen. In section 3.4, we present all the open source packages for hashing implementations and we discuss possible performance issues for Skein implementation. Section 3.5 explains the philosophy behind the shell scripts and it briefly describes the interesting parts of the source code. Finally, in the last section of this chapter we present some of our early research concerns with respect to the cluster model, allowing the reader to understand on which issues our investigation is mainly focused.

## 3.1 General Concept

By using a simple distribution of a *linux operating system* and a number of *open source tools* we successfully set up a *4-node cluster architecture*, which is controlled by a *main node*. This main node is a classic personal computer, on which 5 *Network Interface Cards (NICs)* and the proper software have been installed. All the cluster nodes are diskless and only the main node maintains a hard disk attached. Each *node* is connected directly to the main node without any in between network devices, such as *routers* or *switches*. In order to make the cluster nodes to boot from the network we use the *Preboot eXecution Environment (PXE)*, which is a standard environment found in almost every modern *Basic Input/Output System (BIOS)*. This environment provides the ability to boot an operating system using a network interface, without the need of an available *data storage device* at each node. In this way we are able to configure the 4 motherboards, which miss only the local hard disk, to *boot remotely*. More precisely, the PXE option is enabled on the BIOS of each cluster node and the operating system is booted *on the fly* form the main node through the network. The hard disk installed on the main node becomes the central *shared storage point*, on which all the necessary data are saved. Of course, the entire process hides a more complex configuration and several other steps are required, which will be further explained in chapter 4.

After the completion of the cluster setup and through the installed clustering tools, we are able to schedule a number of tasks *in parallel* and evaluate the *overall time* for submitting job requests. In practice, each of the node processors is capable of processing a different request at the same time and that make us quite confident that it will give us shorter *total time* for the completion of a task. However, the fact that we use only a single point of storage introduces several interesting research questions which are formulated in 3.6 and in practice, summarizes the *core* of our research. All *transferring*, *processing* and *storage* of data is done *on the fly* by using the ethernet interfaces and this has an *unavoidable impact* on the general behavior of cluster architecture. It may spare us from a *separate* operating system installation for each node and also provide us with a significant level of *freedom* and *flexibility* for the configuration. Nevertheless, it also induces an important *network latency* and an obvious *bandwidth dependability*, which must be further evaluated. The idea of performing a benchmark based on existing hashing implementations is considered quite interesting and a reliable path for putting our model under testing. By combining all the knowledge for the discussed areas of applicability of hashing algorithms and the critical performance points for the presented architecture, we are able to structure our test cases the best possible way and extract useful conclusions. We successfully install a number of open source hashing implementations used on linux distributions and we benchmark two different test cases. These tests aim to reveal the *independent performance* of each hash algorithm, but also try to explore any latency issues that will arise from the diskless architecture by using different data sizes. In particular, we choose a specific database of *input files* based on the needs of each benchmark and we write the proper *shell scripts* in order to run the desired tasks. By using the *resource manager* installed on main node, we schedule *in parallel jobs* at all cluster node and we produce an adequate amount of results for analysis.

Our main goal is to identify the *advantages* and *disadvantages* of the diskless cluster architecture, but also to get a feeling about the speed of the existing hashing algorithms. For accomplishing a better comparison and understanding for each corresponding node behavior, we introduce additional independent hardware. More precisely, we include the main node and two independent hardware architectures into our benchmark, as a different measure for comparison. In reality, we run all the identical test cases in systems with *faster processors* and *independent hard disk installations* and we record the noticeable differences. As a result of this additional testing, we obtain the ability to successfully argue about the benefits of a possible *processor substitution* in the cluster nodes. Demonstrating that simple and straightforward improvements could easily allow us to *scale* the achieved performance.

## 3.2 The Puzzle of Hardware

As we proceed with this document there is a need to remember and understand most of the hardware details that would be described here, since this knowledge will become a *pre-request* for the next chapters. We definitely advise the reader if it is possible to memorize at least the different architectures so that it could follow our upcoming data analysis, without having to return back to this chapter for further clarifications. In an attempt to make this process easier we provide an overall table (figure 2) with all the information for the hardware, correlating each of the systems with names derived from *Hellenic Mathematicians.* These names constituted the real hostnames of our systems and it was decided to be included as a helpful reference method for avoiding misinterpretations. After the description which follows for each of the systems, every architecture will be further referred by using the equivalent name and it is therefore recommended to show the proper attention.

**Pythagoras:** Is the *main node* of the cluster architecture and in practice is the most crucial system for the entire project. It facilitates every communication with cluster nodes and is the system that takes care of the successful *transferring* and *storage* of information *from* and *to* the shared hard disk. All the clustering tools are installed and configured on this system, making Pythagoras the initial point for every performed action. Pythagoras is equipped with *2.5 GB* of memory, an *Intel Pentium D* processor clocked at *3.2 GHz* and a hard disk that has *147 GB* of storage space. Moreover, in order to be directly connected with every cluster node and also be *remotely* controllable through an existing *intra-net*, Pythagoras provides *5 ethernet cards.* Three of them have a transfer rate of *1 Gbps* and the other two operate at the rate of *100 Mbps.* The first card of 100 Mbps is used for the intra-net connection, separating in this way the network bandwidth used for *administration duties* and excluding any unwanted latencies. The reason for including a second 100 Mbps card for establishing a connection with one of cluster nodes, is mainly to explore any major bandwidth related impacts and compare the performance with the other connections of 1 Gbps. Usually, in commercial architectures there is a *mix* of various hardware resources that have different capabilities. Therefore, we try when it is feasible to adopt a similar philosophy.

**Archimedes, Aristotle, Democritus, Thales:** These are the *4 diskless cluster nodes* which participate in our modeled architecture. They use identical motherboards with a small difference on the installed amount of memory and their setup has *no major* differences. More precisely, all the motherboards are equipped with an *Intel EP80579 Integrated Processor for Embedded Computing.* This family of processors maintains the ability to *accelerate* cryptographic operations. However for reasons that we will briefly discuss in chapter 5, this acceleration was not enabled. The cluster nodes are treated as normal processor units and the basic *Intel Pentium M* clocked at *1.2 GHz* which EP80579 is built on, constitutes the only processing power of the nodes. Archimedes and Aristotle have both *1.5 GB* of memory and a *1 Gbps* ethernet card. They represent the most powerful nodes on the cluster and Archimedes will become the cluster node for evaluating the *single remote* performance. Democritus has also *1.5 GB* of memory, but is connected to Pythagoras through the *100 Mbps* ethernet card. On the other hand, Thales has a *1 Gbps* ethernet card, but smaller amount of memory, since it has only *1 GB* installed in the form of two *512 MB* memory chips.

**Euclid:** Is an independent motherboard, which has the exact same processing capabilities with the hardware that resides on the diskless nodes. It is configured in a *personal computer* architecture and it is a complete and *stand-alone* system. Is also equipped with an *Intel EP80579 Integrated Processor*, but the *acceleration* is disabled. That leaves again the system with the *Intel Pentium M* clocked at *1.2 GHz* as the actual processor unit. Furthermore, Euclid has *1GB* of memory and a *160 GB* hard disk. The reason for introducing Euclid as an extra architecture for running our test cases, is quite obvious. Euclid has identical hardware with all the cluster nodes, but in comparison to the nodes it is not diskless. As a result, this will allow us to perform the same experiments and obtain valid conclusions about the *performance impacts* that are introduced from processing all data, entirely through the network.

**Plato:** Constitutes the last and most powerful system, on which we benchmark hashing implementations. In reality, it is an individual laptop that possesses an *Intel Core 2 Duo Processor* clocked at *2.4 GHz*, *2 GB* of memory and a *160 GB* hard disk. The processing power that this system offers, is of course superior from all the pre-mentioned systems. The main reason lies on the fact the Intel Core 2 Duo belongs to a later generation of processors and therefore, is based on a totally different architecture for integrated circuits. The decision for using a system such as Plato to the testing process was considered quite sensible, since this is a great way to introduce an additional metric for performance. By running the same test cases on Plato, we can successfully see how a *faster processor unit* behaves and record the *deliverable time* for the completion of each scheduled job.

At this point the reader should very wisely wonder, why we did not use latest processor units, such as the family of *Intel Core i7 processors.* Moreover, he can question the fact that we did not use processors with different capabilities on the cluster nodes, in order to record the behavior of our entire model under a *mixed architectural environment.* Indeed, these remarks find a valid ground for discussion, however they hide two main reasons for their appearance. Firstly, the cluster nodes were built with the intention to use the embedded acceleration features that the *Intel EP80579 Integrated Processor* offers for cryptographic operations. Unfortunately, in the end this was not feasible, leaving us the option of a normal use as the only possible solution. Furthermore, it was considered that a more powerful hardware on the nodes side would not result to major differences for the performance of the diskless architecture with respect to the *on the fly* processing and *network latency.* Nevertheless, in order to satisfy our curiosity, we try to sufficiently cover also this case. We perform the same tests at a faster processor *independently*, by using Plato and we argue about a possible node processor substitution. Realistically speaking, there are several alternative architectural directions that someone could follow and this should always depend on the exact problem that one aims to solve. It is our belief that the nature of the problem should be the priority for defining the most suitable hardware and only then a diskless architecture will be able to take advantage of its capabilities at maximum. In our case the hardware is chosen based on factors like *hardware availability*, *nature of the problem* (even if cryptographic acceleration was not finally used), *interesting research issues for the diskless approach* and *existence of suitable software.* The idea is to focus on the general behavior of the cluster model and identify the possibilities for improvement. Thus, we benchmark a number of open source hashing implementations *under different conditions* and we relate the cluster-based architecture with the intensive computational parts, proposing an alternative approach for achieving *in parallel* task scheduling.

```
+---------------------+---------------------------------------------+
|        Name         |                 Description                 |
+---------------------+---------------------------------------------+
|                     |             /* Main Cluster Node */         |
|                     |                                             |
|                     |           Intel Pentium D, 3.20 GHz         |
|     Pythagoras      |           2 GB + 512 MB, DDR2 667MHz        |
|                     |           Hard Disk: 147 GB                 |
|                     |           5 NIC Cards: 3 x 1 Gbps,          |
|                     |                        2 x 100 Mbps         |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |            /*  1st Cluster Node  */         |
|                     |                                             |
|                     |           Intel Pentium M, 1.2 GHz          |
|     Archimedes      |           1 GB + 512 MB, DDR2 400MHz        |
|                     |           Hard Disk: Diskless               |
|                     |           1 NIC Card: 1 Gbps                |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |           /*  2nd Cluster Node  */          |
|                     |                                             |
|                     |           Intel Pentium M, 1.2 GHz          |
|     Aristotle       |           1 GB + 512 MB, DDR2 400MHz        |
|                     |           Hard Disk: Diskless               |
|                     |           1 NIC Card: 1 Gbps                |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |           /*  3rd Cluster Node  */          |
|                     |                                             |
|                     |           Intel Pentium M, 1.2 GHz          |
|     Democritus      |           1 GB + 512 MB, DDR2 400MHz        |
|                     |           Hard Disk: Diskless               |
|                     |           1 NIC Card: 100 Mbps              |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |           /*  4th Cluster Node  */          |
|                     |                                             |
|                     |           Intel Pentium M, 1.2 GHz          |
|       Thales        |           2 x 512 MB, DDR2 400MHz           |
|                     |           Hard Disk: Diskless               |
|                     |           1 NIC Card: 1 Gpbs                |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |      /*  Independent Identical Board  */    |
|                     |                                             |
|       Euclid        |           Intel Pentium M, 1.2 GHz          |
|                     |           1 GB, DDR2 400MHz                 |
|                     |           Hard Disk: 160 GB                 |
|                     |                                             |
+---------------------+---------------------------------------------+
|                     |          /*  Personal Notebook  */          |
|                     |                                             |
|       Plato         |           Intel Core 2 Duo, 2.4 GHz         |
|                     |           2x 1GB, DDR2 667MHz               |
|                     |           Hard Disk: 160 GB                 |
|                     |                                             |
+---------------------+---------------------------------------------+
```

Figure 2: Description of the Hardware

## 3.3 Database

The *database* was chosen in order to fulfill two basic requirements. The first requirement is the variety on the file sizes for processing. By including CD image files which have size of approximately *630 MB* each, we are able to test a case on which the input data can be loaded entirely into the memory of each system. Note here, that the amount of installed memory for all the participated architectures is *1 GB* or *higher*. On the other hand, by including also a DVD image that has a size of *4.7 GB*, we are able to test the case on which the input data should be divided and loaded into the memory partially. As a second requirement, we set the inclusion of a test case that is related to intensive processing of input data that have a negligible size, but the workload is quite large. More precisely, this case is dedicated to the *dictionary files* and it tries to reveal the performance behind the hashing of *thousands of words* read from a file. The used dictionary file has a size of *84 KB* and it contains *10.200 words*, which give us an average of *8.5 bytes per word* as data input for the hash functions. The *exact files* which were included on our experiments and also their *size*, are summarized in the following figure.

```
+-----------------------------------------------+
|                  Database                     |
+-----------------------------------------------+
+-----------------------------------------------+
|  /*  1st Test Case:  Hashing Large Files */   |
+-----------------------------------------------+
|  1. CentOS-5.0-i386-bin-2of6.iso  (636 MB)    |
+-----------------------------------------------|
|  2. CentOS-5.0-i386-bin-3of6.iso  (627 MB)    |
+-----------------------------------------------|
|  3. CentOS-5.0-i386-bin-4of6.iso  (633 MB)    |
+-----------------------------------------------|
|  4. CentOS-5.0-i386-bin-5of6.iso  (636 MB)    |
+-----------------------------------------------|
|  5. Debian-504-i386-DVD-1.iso  (4011 MB)      |
+-----------------------------------------------+
+-----------------------------------------------+
|     /*  2nd Test Case:  Hashing Words  */     |
+-----------------------------------------------+
|  1. Dictionary.txt - 10.200 Words (84 KB)     |
+-----------------------------------------------+
```

Figure 3: Database

## 3.4 Open Source Hashing Implementations

The software which we installed for applying our benchmark is two existing *open source packages* that implement the most popular hashing algorithms and a *python library* which is used for the *skein* algorithm, all presented in figure 4. The *hashing commands* that these applications offer, are *easy to use* and they find applicability to most *linux distribution* enhancing the confidence for *stability* and *correctness*. The *GNU Coreutils* package constitutes the classic and most commonly used implementation for delivering regular hashing operations. In our benchmark we include the *md5sum*, *sha1sum*, *sha224sum*, *sha256sum*, *sha384sum*, and *sha512sum* commands. As an additional package we use *GPL md5deep*, which implements *tigerdeep* and *whirlpooldeep*. This package provides a number of commands that implement the same hashing algorithms with Coreutils, allowing us to test *sha1deep* and *sha256deep*. Finally, as already mentioned previously, we use the features of *GPL PySkein* library. By using this *python* library and by modifying an existing sample script for Skein-256, we successfully create the *Skein256sum*, *Skein512sum* and *Skein1024sum* scripts. Here we should mention, that *Python* is a very popular *interpreted scripting language*, as opposed to *C* which has as a base a *compiled* nature. Due to this significant difference, C is generally faster than Python. *Coreutils* and *md5deep* software packages are written entirely in C, however the performed tests for *Skein* use *python* scripts. Thus, we should keep in mind that this is likely to cause delays and slow down the *processing speed*. In particular, the authors of Skein algorithm used *optimized C* and *assembly language* to benchmark their design and the performance results discussed in 1.3.1 correspond to the optimized C testing. Our expectations for the Skein python scripts remain relatively modest compared to the C implementations. Nevertheless, there should still be a visible relation that would successfully link our analysis to the results that the authors reported in [23].

```
+--------------------+----------------------------------------+
|                    |                                        |
|     Commands       |           Additional Details           |
|                    |                                        |
+--------------------+----------------------------------------+
+--------------------+----------------------------------------+
|  1. md5sum         |                                        |
+--------------------|          (GNU coreutils) 5.97          |
|  2. sha1sum        |                                        |
+--------------------|        Free Software Foundation        |
|  3. sha224sum      |                                        |
+--------------------|      /*       Written by       */      |
|  4. sha256sum      |                                        |
+--------------------|           - Ulrich Drepper -           |
|  5. sha384sum      |            - Scott Miller -            |
+--------------------|            - David Madore -            |
|  6. sha512sum      |                                        |
+--------------------+----------------------------------------+
+--------------------+----------------------------------------+
|  7. sha1deep       |           (GPL md5deep) 3.6            |
+--------------------|                                        |
|  8. sha256deep     |        United States Air Force         |
+--------------------|                                        |
|  9. tigerdeep      |      /*       Written by       */      |
+--------------------|                                        |
|  10. whirlpooldeep |          - Jesse Kornblum -            |
+--------------------+----------------------------------------+
+--------------------+----------------------------------------+
|  11. skein256sum   |           (GPL PySkein) 0.6            |
+--------------------|                                        |
|  12. skein512sum   |     /*        Written by       */      |
+--------------------|                                        |
|  13. skein1024sum  |          - Hagen Frustenau -           |
+--------------------+----------------------------------------+
```

Figure 4: Hashing Commands

## 3.5 Shell Scripts

*Shell scripts* are used for *automating* the entire testing process. We create a number of bash script variations, however they all function by *feeding* each hashing command with an input file. The existing commands from each implementation package and the written python scripts for *skein*, are all called directly through these bash scripts. The scripts take *no external input* and during their execution are reading the input from the proper *directory* or *file*. As we will describe in chapter 4, the *cluster scheduler* does not accept job scripts with a *direct input* and therefore, we designed our scripts under this *prerequisite*. In particular, we wrote two different versions. The first version is able to read the *image files* from a directory and pass them as an input into the hashing implementation and the second one is able to open a file, read it per line and pass each word as an input to the hashing command. These two scripts appear in two basic forms. The first form is referred as *allhashes* and it is a script that includes all the hashing commands. *Allhashes script* computes all the hash values for every benchmarked algorithm and saves the output into a *text file*. The second form is referred as *single* and it corresponds to a script that will call only one specific algorithm. *Single script* produces the hash values only for this specific hash algorithm and it saves again the generated output in a *text file*. These scripts are executed for both cases of the *dictionary file* and *image file* hashing. In appendix A.1 we present extensively the *source code* for the *allhashes scripts* and the *single script* used for *md5sum* command. In total, we used *14 bash scripts* for each separate case. The *allhashes script* was able to give us the hash values for all hashing algorithms at once and the other *13 single scripts* produced the hashes for a specific algorithm, resulting in the exact same output as the *allhashes script*. In this way, we successfully divided the workload in an *easy* and *straightforward* way, that allowed us to parallelize the hashing operations and record the benefits. Of course, it is understandable that there are *more efficient* ways to perform parallelized processing, however the hashing operations are *state dependable* and additional techniques are required. Therefore, since our focus remain on the cluster architecture we decided not to proceed with such advanced test cases. However, we definitely recommend this kind of testing as a possible future work.

Inside the *source code* of every script, we also include a very straightforward mechanism to monitor the execution time. We use the *time* linux command and we are able to get detailed information about processing time for each hashing command. The first line of the following code presents the part of the source code that performs the hash for *large files* computing an *md5 hash* and the second line gives the equivalent operation for hashing the content of *a dictionary file* per line.

```
time -f "User:%U Sys:%S Time:%E CPU:%P" md5sum $file 1 >> ../allmd5hashses 2 >> ../allmd5hashes

echo $line | time -f "User:%U Sys:%S Time:%E CPU:%P" md5sum 1 >> ../allmd5hashses 2 >> ../allmd5hashes
```

On both lines of code the *md5sum* command will hash the input taken from the variable $file and $line, respectively. Simultaneously, the *time* command will start monitoring the execution of *md5sum*, until it finishes. The outputs of both commands, instead of being printed in the screen are redirected and appended into a file called *allmd5hashes*, leaving us with all the information we need for performing our data analysis. More precisely, we record the time spent in the *user space*, *kernel space*, but also the *total elapsed* time for completing the hashing operations. Furthermore, we monitor the *CPU utilization* that occurs during the processing time frame and we log the behavior of each algorithm with respect to the usage of the processor unit. In order to introduce timestamps that will allow us to accurately calculate the total time spent for each of the tasks, we use the *date* linux command. By appending the outcome of *date* command at certain points (see A.1) in the content of the output files, we successfully keep track of the total *completion time* of specific tasks that occur during the execution of the script. It is important to mention that the redirection feature for appending information into a file is *not expensive* when is done locally in a system and it does not occur often. This was a very interesting observation that we made when we tried to redirect information from the cluster nodes. The *thousands words* hashed for the dictionary case, yielded a significant inefficiency. The introduced latency from the operations of *opening*, *closing* and *saving* that redirection probably invokes, were *extremely costly*. The need for appending information again and again *remotely* to the shared hard disk was adding a *false delay*. Therefore, for the case of *hashing words* at the cluster nodes we excluded all the redirections, since the clustering tools offer an *adequate mechanism* that directly saves the results of a completed script at once. For the case of *large files* we did not observe any significant latency and thus, we used the *data redirection*. The reader can reasonably wonder why we did not use the saving feature that cluster manager provides, for all our test cases. The answer is rather simple and it lies on the fact that we wanted to structure our own controllable way for producing the output. The latency which we observed for the rest of our test cases was *negligible*, allowing us to proceed as initially planned. However, as a future recommendation we advise the usage of this feature, especially if the benchmark involves output which is related with time measurements. If avoiding customized designs for producing the output files is not possible, an investigation that will verify that the followed approach does not *affect the validity* of the results, is at least expected.

## 3.6 Research Questions

In our clustering project, there are several different research questions that could be formulated. However, it is out of the scope of this document and almost infeasible to cover every possible aspect. The scientific field of *high-performance computing* exists for many years and has a breeding ground for endless research quests. This section is dedicated to the *personal concerns* of the author, as they arise during the entire experiment. The most important investigation points are highlighted and through a short description we aim to inform the reader about the specific research issues on which our upcoming *data analysis* is focused. It was considered optimal to present these issues in a form of *10 basic questions*, accompanied with a brief concept analysis, as follows:

*1. Are there any significant performance impacts for using a remote board, rather using the exact same board as a normal personal computer architecture ?*

Archimedes constitutes one of the remote cluster nodes that is connected to Pythagoras. On the other hand, Euclid is an identical and *stand-alone* system that has its own separate hard disk. By running the test cases on both architectures, we expect that there should be a recorded latency for Archimedes. The processing is performed by transferring data through the network interface and especially for the large image files this should induce an obvious delay. The amount of time that is need for transferring the files over the network, in comparison with reading them directly from a hard disk, is definitely more. On this investigation there are two important points that must be taken into account. The first one is that Euclid has *512 MB* less memory than Archimedes and that is for sure a performance disadvantage for the large files. On the other hand, for the test case of dictionary file the network delay should be negligible, since the file size is close to *84 KB* and each of the contained words has an average size of *8.5 bytes*. Therefore, both systems are expected to report a similar processing time.

*2. Is the induced network latency from the on the fly processing, so severe?*

The question refers to a general conclusion that should be extracted from our data analysis. The diskless architecture is based on the fact that has no need for storage points at every node and network connections constitute the only way for transferring data to the nodes. If the transfer introduces a severe latency, then this could be a sufficient reason for rejecting the architecture. Of course, the word *severe* remains quite abstract, since the latency is still application specific. In a sense that it depends on the opportunities for parallelization, but also the size of the processing data that the application is using. By using in our case hashing implementations on which parallelization is limited and the data size varies, we will have an overall assessment about the network delays from every perspective.

*3. How do the different hashing implementations behave? Is there any major difference with respect to the input file sizes.*

The reason for benchmarking a number of different hashing commands with various input data sizes, is to get as much information as possible for their general performance. The hashing of a *CD image* that can be loaded at once into the memory and the hashing of *DVD image* that has a larger size that the existing memory resources, are expected to yield differences among the different hashing algorithms. Moreover, the usage of commands from different packages that implement the identical algorithm, should show *similar performance* on the same tasks. Finally, the dictionary file that contains a list of words for hashing, should not report an extreme difference on the delivery time.

*4. Do the Skein python scripts verify the performance that the authors have recorded?*

The authors reported that in some cases (e.g. 512-bit), the *Skein* algorithm is faster than the equivalent SHA-2 implementation. It is interesting to verify this claim, but also to obtain general performance for all the digest of this new hash algorithm. The test cases for *large files* and for the *dictionary file*, would be able to reveal additional aspects of the algorithm and give us the overall picture. Of course, the usage of an interpreted language such as python, could lead to unexpected results and this is another interesting point for investigation.

*5. How are the processing times influenced, with respect to the usage of different resources for each cluster node?*

The cluster nodes do not have the same amount of memory installed and Democritus has installed an ethernet interface with a lower data transferring rate of *100 Mbps*. We expect that these differences will affect in some extend the performance, especially for the large files. However, we are not quite sure if these difference would lead to a *negligible* or *not*, form of influence. We assume that the memory resources for the cluster nodes would not have a great impact, since the difference is *512 MB*. On the other hand, we have a confidence that the network interface of Democritus is expected to decrease the performance for the hashing of the image files, compared to the other nodes.

*6. Which is the performance for hashing a dictionary file for cryptanalysis reasons ?*

The test case of the dictionary file is directly connected to the field of cryptanalysis, since usually it is the base for off-line brute force attacks. We do not expect from our cluster model (without acceleration enabled), to be able to report processing capabilities as the ones that *Graphics Processor Units (GPUs)* yield. Our intention is to verify that the transferring of the words contained in the file does not cause any latency issues and also record the processing time for all the different hashing algorithms. Of course, we do not test our architecture on extreme situations, on which the dictionary file size is enormous. The reason is that we would have to wait for days or weeks for the results and it would probably not change anything with respect to the transferring of data, since the performance for the network can be anyway extracted from the image files case.

*7. How do the different processors perform on the same task ?*

We are mainly interested to compare Pythagoras and Plato, which possess a better processor unit. These architectures have also a greater and faster amount of memory installed and they have a separate hard disk from which the input data are retrieved. We should be able to record, a significant difference by comparing these systems to Euclid and Archimedes at all levels for the tested hashing commands. More precisely, we expect Plato to be the faster from all the used systems, since it has a later generation of a processor that includes *2 cores*. However, we still maintain a small doubt for the image files, because Pythagoras has *512 MB* more memory and represents an early approach for the *2 core* processors. Finally, from the results of this comparison we assume that we will be able to verify that a possible processor substitution at cluster nodes, would report an equivalent increase on performance.

*8. By splitting the task into the cluster nodes, what kind of time optimization will we achieve ?*

We split the workload for both test cases into the cluster nodes and in practice, we parallelize the processing in a really simple way. We expect that this division of tasks would yield a significant difference and will successfully verify the concept of clustering. We measure the overall time for completing the jobs needed for producing the same desired output and we expect to report that processing at the 4-node cluster is significant faster.

*9. What are the advantages and disadvantages of scaling ?*

In this case, scaling refers to the process of adding more cluster nodes to Pythagoras. In our experiment the boards were connected directly to the network interfaces that Pythagoras provided. However, there is the ability to support more cluster nodes at each ethernet card by using devices, such as *network switches*. This would indeed scale the processing capabilities of the cluster architecture, since it would add more end points. The main question is to investigate if the induced network latency makes such an approach inefficient. An early assumption is that in the case on which the amount of transferred data is not extremely large, the latency could be tolerable.

*10. What further optimizations and additional changes could be done in such an architecture in order to increase performance?*

After analyzing the collected data, we should be able to argue about the possible optimizations. Most of the critical potential points for performance are expected to be on the *latency* from the network interfaces and the *economical cost* for extending the capabilities of the cluster. However, our final evaluation is expected to come after the completion of the conducted benchmark, since only then we would be able to have a better insight on the *apparent trade-offs*.

# 4 Building a Scalable Cluster

The previous chapter gave a general overview for the way that we perform our benchmark and presented the basic structure of our approach for the cluster. However, the detailed information for building the cluster was not covered. This chapter guides the reader through the exact steps that are required, in order to successfully setup the entire system. The clustering tools are described to a sufficient extent and an opinion is formulated with regard to the scalability of the diskless architecture. It is impossible to exhaust all the options that this architecture provides within a *single chapter*, thus we refer only to the most relevant parts with respect to our research.

## 4.1 The Diskless Architecture

The ability to configure a *multi-node architecture* that has no need for a separate hard disk at each end-point is quite attractive [43]. As in any model choices, this reveals *advantages* and *disadvantages.* The avoidance of hard disks decreases the cost and the physical space needed for the final clustering system. In a *small scale* approach like ours, there is not an obvious benefit. However, in *large scale deployments* such as research data centers, the diskless feature could easily constitute a significant factor. Another interesting aspect is that the multiple storage points are moved into a single shared point. This enables *common access* to the same data for all nodes, but it also induces concerns about a *single point of failure.* Furthermore, the existence of only one storage device is closely related to the *transferring rate* that network interfaces can offer and it also depends on the *communication protocol* used for data transferring. Even though there are several *trade-offs* that must be considered when adopting this architecture, we believe that there is an opportunity for easy scalability. Each interface can support *up to 253* cluster nodes at the same network, by simply using network switches. As figure 5 shows, in our architecture the 4 network cards would be able to *theoretically* support close to *1000 diskless nodes.* Of course, this scenario comes with performance restrictions and it is mainly confined by the hardware capabilities. However, the fact that the operating system can be loaded *on the fly*, gives a great amount of flexibility.

```
+-----------------------------------+
|       /*   DRBL SERVER   */       |
|                                   |
|           [pythagoras]            |
+-----------------------------------+
+-----------------------------------+
|                          100 Mbps |
|+-- [intra]:10.243.18.98 +---------+-- [WAN]
|                                   |   /* Controlling pythagoras */
|                          1 Gbps   |
|+-- [eth0]:192.168.100.1 +---------+- [archimedes]: 192.168.100.2
|                                   |   /* Up to 192.168.100.254  */
|                          1 Gbps   |
|+-- [eth1]:192.168.101.1 +---------+- [aristotle]: 192.168.101.2
|                                   |   /* Up to 192.168.101.254  */
|                          100 Mbps |
|+-- [eth2]:192.168.102.1 +---------+- [democritus]: 192.168.102.2
|                                   |   /* Up to 192.168.102.254  */
|                          1 Gbps   |
|+-- [eth3]:192.168.103.1 +---------+- [thales]: 192.168.103.2
|                                   |   /* Up to 192.168.103.254  */
+-----------------------------------+
```

Figure 5: The Cluster Architecture

Imagine, that if the proper configurations are made, it is even possible to use a standard infrastructure for booting all the connected machines through the network. By just changing the *boot priority* at the BIOS of each system and by defining a central server that will be responsible for handling the setup process, a normal infrastructure could support a *second cluster nature.* The success for this approach lies on the ability to alternate between the *normal mode* and the *cluster mode*, by configuring the boot option from the local hard disk into the PXE environment and vice versa. In such a case an obvious availability issue arises for the nodes, especially for large institutions like *Universities.* Nevertheless, the *cost* for the hardware remains stable and the cluster mode does not necessarily imply that an administrator must enable every single machine on the system as a cluster node. This provides an adequate amount of *freedom* and allows even a mixed configuration. Although that such architectures are not common for a home usage, it is possible and relatively easy to set them up if you have the proper hardware. Assuming that many people have a number of old computers *or* usually end up with old computers, this could allow them to setup a small cluster system, with 2-4 nodes. Of course, several technical skills are needed. However, we consider that the potential users have all the required qualifications for building such a system. An obvious question would be the reason for setting up a cluster system with old processors, since this architecture targets mainly on additional computational power. The answer is hidden in the ability to obtain a parallelizable environment for dividing the workload. As we already mentioned in the previous chapter, *Graphics Processor Units (GPUs)* offer a powerful processing environment. [9]. So, if a motherboard is *compatible* and can support one or more GPUs, then the architecture could end up with nodes that have a *slow* main processor unit, but at the same time are equipped with powerful GPUs.

The main factor that allowed us to build our own diskless cluster, is found in the *open source* nature of the clustering tools. The existing research communities on the field of *High Performance Computing* provide and continuously evolve these tools, by publishing also helpful guidelines for their usage. Without these resources the task would not be infeasible, but it would be for sure way more expensive and probably more difficult. Even though this kind of software is not so popular in the majority of the IT community, it is in existence for several years. Many institutes, including *Universities* and *Scientific Centers*, have deployed their own grid computing systems based on the usage of such software. The following section describes 3 of the *clustering tools* that we installed and presents their main features.

## 4.2 Clustering Tools

*Diskless Remote Boot in Linux (DRBL)*[15] constitutes the *core* of the entire system and it is the fundamental component for enabling the diskless architecture. By bringing into mind our architectural setup, it is the tool that will create the needed environment for establishing the connections between Pythagoras and all the independent systems, namely Archimedes, Aristotle, Thales and Democritus. More precisely, DRBL is responsible for handling the boot process of the Operating System (OS) for every remote motherboard, configuring the network interfaces and making sure that the proper functionality for the data transferring is enabled. When the setup of DRBL has been completed, all systems are able to communicate with each other through Pythagoras, which is the *central point* of the cluster and has the role of *DRBL server.* At this point, our architecture is completely missing the clustering functionality. Therefore, there is a need for a *management system* that would control the resources, but also a need for a *scheduling system* that would be able to queue the desired jobs, in order to be processed from the independent units. *TORQUE*[16] is a resource manager that successfully co-operates with DRBL and it is the the tool that we installed for providing the clustering functionality. It is able to automatically identify the network interface and discover all the connected nodes. In this way, it provides details about the specific hardware installed at each end-point, but also gives the status of usage (e.g. free, running) according to the submitted workload for processing. Furthermore, TORQUE gives the ability to *submit*, *delete* and *distribute* the workload with extended freedom and it offers sufficient monitoring commands. Although TORQUE can perform job scheduling, we decided to install additional recommended software for obtaining more flexibility. *Maui*[17] is an advanced job scheduler that can be successfully integrated into TORQUE and it is designed to support and deliver even more administration options. DRBL, TORQUE and Maui, are all installed and configured on Pythagoras, which represents the main administration point of our architecture. As described previously, Pythagoras is accessible via a separate *intra-net* interface and thus, the entire monitoring can be performed also *remotely.*

### 4.2.1 Diskless Remote Boot in Linux (DRBL)

DRBL was developed by Steven Shiau, Kuo-Lien Huang, H. T. Wang, Ceasar Sun, Jazz Wang and Thomas Tsai. It offers a *centralized boot environment*, allowing client machines to function in a diskless environment. It works under all major linux distributions and it delivers many additional features such as an utility called *Clonezilla.* This utility enables *partitioning* and *hard disk cloning*, similar to the popular Symantec Ghost. Nevertheless, in our experiment we do not exhaust all the capabilities of DRBL, since it would be out of the scope of our investigation. We only take advantage of the diskless functionality and we perform the necessary steps, in order to transform Pythagoras into a *DRBL server.*

**Initial Steps:** Before we install DRBL software we complete the physical architecture and we connect all the systems into the future DRBL server. For our convenience we create a stack of motherboards that includes Archimedes,

[15]http://drbl.sourceforge.net/
[16]http://www.clusterresources.com/products/torque-resource-manager.php
[17]http://www.clusterresources.com/products/maui-cluster-scheduler.php

Aristotle, Thales and Demorcritus and through the ethernet interfaces we connect these systems to Pythagoras. All the remote systems have no hard disk and they have been enabled to boot from the network. On the other hand, Pythagoras has a linux operating system installed and it is equipped with all the peripheral devices, such as monitor, keyboard, mouse and a dvd-drive. After building the main physical structure, the network interface on Pythagoras should be properly configured, before we install DRBL. Therefore, we make our own custom configuration for each network interface and we create different networks, by assigning the chosen *IP addresses*. At this point we start the installation of DRBL on Pythagoras. There are two options for installing DRBL. The one is to manually download and install all the *pre-required software* and the other one is to let DRBL install all the needed packages through Internet. In our case, for simplifying the process and since Pythagoras has an access to an Internet connection through the intra-net interface, we use directly DRBL. By executing the command: `/opt/drbl/sbin/drblsrv -i`, DRBL starts to download and automatically install all the proper software.

**Setup Process:** After all the initial steps have been completed, we are able to start the *setup process*. We ensure that all the systems are *powered on*, even if they have not an operating system installed. Moreover, we check that the small light on each ethernet card is blinking, as an indication that all physical links operate successfully. This is quite important, since DRBL during the setup process will perform an automatic collection of *Media Access Control (MAC) addresses* for each interface. By executing the command: `/opt/drbl/sbin/drblpush -i`, a configuration guide begins in the form of *questions* and *answers*. In practice, it is an intelligent DRBL script that covers all the possible aspects of configuration that an administrator will need to complete. By answering *step by step* each question, Pythagoras is transformed into a functional DRBL server. During our setup, we disable the options that are not applicable to our scenario, such as the *Clonezilla* utility tool and the *existence of hard disks* at the nodes side. The *three* most important settings that appear on this script, is the setup of a *Dynamic Host Configuration Protocol (DHCP)* server, the creation of the *boot image* and the *allocation of space* at the shared hard disk for the *root filesystem* of each node. DHCP is used by hosts to dynamically retrieve an IP address and other configuration information. When PXE environment runs, it is looking for a DHCP server that will provide information, such as an *IP address*, *a name server* and the location of the boot image to be loaded. DRBL will identify the unique *MAC addresses* of each powered on network card and it will setup a DHCP server that binds these addresses with specific IP addresses. In this way, when the PXE of each motherboard will request information for booting an Operating System (OS), the DHCP server which is configured on Pythagoras, will successfully respond and OS will be loaded *on the fly*. At an earlier stage, there are another two operations that have been already performed by the DRBL setup script. A boot image is created based on the Linux OS installed on Pythagoras and it is placed into the directory of a *tftpboot server*. The tftpboot server is running on Pythagoras and it is one of the pre-required software that have been installed before the setup process starts. Its purpose is to transfer the boot image at the nodes and allow PXE environment to start the remote booting procedure of OS. The fact that the *boot image* is generated based on the already installed OS on Pythagoras, it does not limit the options of the administrator for alternative boot images. In our case, we decided to proceed with the *standard* boot image that DRBL script produces, since there was not any specific reason for building our own customized version. Finally, during this setup we decide if the root filesystem of each node would exist *separately* or it would be *identical* for all the cluster nodes. More precisely, after the OS has booted on each node, there is a need for filesystem access as it would happen in a normal OS installation that uses an attached hard disk. In order for DRBL to handle this vital need, it uses the *Network File System (NFS)*, which allows the nodes to access files over the network, in a way similar to the access that is done when a local hard disk is present. DRBL offers a configuration flexibility, since it is able to use the same *root filesystem* for each node or allocate space for a separate one. Apparently, the dedicated root filesystem offers additional freedom and therefore, in our small scale cluster we decided to generate a distinct filesystem for Archimedes, Aristotle, Thales and Democritus. However, this configuration requires significant *free disk space* and in a larger system, it is recommended to use an identical root filesystem, when that is feasible. After the completion of the DRBL script the setup process has finished and Pythagoras is a functional DRBL server that waits for requests. At this point, we can reboot the remote nodes connected to Pythagoras and the OS will be automatically booted *on the fly* through the network. The basic infrastructure of our diskless cluster architecture is ready and the next final step is to install the *resource management tools*.

**Final Configurations:** Another very useful feature that DRBL offers, is the ability to make the software that resides on the DRBL server available for every remote node. In practice, when we install a new software on Pythagoras the directories on the separate root filesystems of each node are updated accordingly. Sometimes there is a need to *re-execute* the DRBL setup script, since only then the newly installed software will be visible at every remote system. However, DRBL provides the ability to use a configuration file that is created the first time that we execute the setup script. This file contains all of our settings and therefore, running again the setup script for updating all the root filesystems with new software, becomes a quite trivial task. As a final stage, we install TORQUE and Maui on

Pythagoras and we also compile all the chosen hashing commands and the PySkein library. Afterwards, we re-run the DRBL script by using the *initial configuration file* and in this way, we successfully finish the setup of our cluster architecture. We reboot all the systems and after every node is *up and running* we perform some final configurations. By using *Secure Shell (SSH)*, we connect to each remote node and we start the *client daemons* needed, for allowing communication of resource management tools that reside on Pythagoras and the cluster nodes. Finally, we verify by using TORQUE, that all the systems have been successfully identified and are ready for use.

### 4.2.2 TORQUE & Maui

Both *TORQUE* and *Maui* are derivatives of OpenPBS technology that is actively developed, supported and maintained by Cluster Resources[18]. *OpenPBS* is an open source version of the *Portable Batch System (PBS)*, which was developed for NASA. PBS was software that was able to perform job scheduling and it's main purpose was to allocate computational tasks among existing computing resources. These computational tasks, are usually referred as *batch jobs*. Today, PBS is offered as a commercial version by Altair Engineering[19], under the name *PBS Professional*[20]. Nevertheless, TORQUE and many other free clustering tools have successfully integrated similar scheduling features, based on the initial version of PBS software. TORQUE is responsible for providing control over *batch jobs* and *distributed compute nodes*. On the other hand, Maui is capable of supporting an array of *scheduling policies*, *priorities*, *reservations* and *fairshare capabilities*. Together, they constitute a very efficient way to manage clustering resources and also control effectively the various scheduled tasks.

When all systems are booted and are in a running state, the needed software processes should be initiated for both TORQUE and Maui. In particular, on Pythagoras we start a daemon called `pbs_server`, which will accept all the *PBS requests*. In practice, this daemon according to the received requests will identify the *existence*, but also the *state* of every remote cluster node. When the `pbs_server` is up and running a client daemon called `pbs_mom`, is started on every remote node. This makes all the systems communicate with each other and allows TORQUE to collect the desired information. In order to verify that everything has been completed without any unexpected errors, we execute the command: `pbsnodes -a` on Pythagoras. As figure 6 presents, the 4 cluster nodes have been successfully identified and additional details with respect to the memory resources, the number of processors and the availability state of each remote system, have been retrieved. Finally, for enabling Maui on every system it is sufficient to start a daemon called `maui`. Then, all the additional *monitor commands* that Maui offers, such as the `showq` command, can be executed as normal from each environment.

```
[spyros@pythagoras ~]$ pbsnodes -a

node_aristotle02
     state = free, np = 1, ntype = cluster, status = opsys=linux,
     uname=Linux node_aristotle02 2.6.18-8.el5 i686, idletime=4495861,
     totmem=1554188kb, availmem=1483948kb, physmem=1554188kb, ncpus=1,

node_thales02
     state = free, np = 1, ntype = cluster, status = opsys=linux,
     uname=Linux node_thales02 2.6.18-8.el5 i686, idletime=4495893,
     totmem=773912kb, availmem=706124kb, physmem=773912kb, ncpus=1,

node_archimedes02
     state = free, np = 1, ntype = cluster, status = opsys=linux,
     uname=Linux node_archimedes02 2.6.18-8.el5 i686, idletime=4495917,
     totmem=1489156kb, availmem=1419244kb, physmem=1489156kb, ncpus=1,

node_democritus02
     state = free, np = 1, ntype = cluster, status = opsys=linux,
     uname=Linux node_democritus02 2.6.18-8.el5 i686, idletime=4495874,
     totmem=1294100kb, availmem=1224232kb, physmem=1294100kb, ncpus=1
```

Figure 6: State of Cluster Nodes

[18]http://www.clusterresources.com/
[19]http://www.altair.com
[20]http://www.pbsworks.com/

## 4.3 Job Scheduling & Scripts

In our benchmark the main functionality for scheduling our batch jobs is handled by TORQUE and we use Maui mostly for extending the monitoring capabilities for our job queue. The commands that TORQUE offers are quite powerful and they offer a number of advanced features, which cover adequately our scheduling needs. More precisely, by using the `qsub` and `qdel` commands, we are able respectively to *submit* and *delete* jobs from the queue. Furthermore, by enabling the proper options on the `qsub` command, we define the *desired executional time* and the usage of a *specific node* for processing. As it was already mentioned in section 3.5, the submitted scripts do not allow a direct data input and therefore, our bash scripts are written based on this condition. In order to *simplify* and make *easier* the submission of our hashing scripts, we create a general type of scheduling script, presented in A.3. We use the `walltime` option to define a maximum time of *24 hours* for the execution of each hashing script and in practice, we submit all the jobs in a default queue, which we named *batch*.

```
[spyros@pythagoras ~]$ qstat

Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
354.pythagoras            md5sum           spyros          00:00:31 C batch
355.pythagoras            sha256sum        spyros          00:00:34 C batch
356.pythagoras            sha256deep       spyros          00:00:35 C batch
357.pythagoras            sha224sum        spyros          00:00:35 C batch
358.pythagoras            sha512sum        spyros                 0 R batch
359.pythagoras            sha384sum        spyros                 0 R batch
360.pythagoras            sha1sum          spyros                 0 R batch
361.pythagoras            sha1deep         spyros                 0 R batch
362.pythagoras            tigerdeep        spyros                 0 Q batch
363.pythagoras            whirlpooldeep    spyros                 0 Q batch
364.pythagoras            skein256sum      spyros                 0 Q batch
365.pythagoras            skein512sum      spyros                 0 Q batch
366.pythagoras            skein1024sum     spyros                 0 Q batch
```

Figure 7: Monitoring with qstat - TORQUE

The above figure presents the created queue for all the hashing scripts, after executing our scheduling script. This output is produced by the `qstat` command that TORQUE provides and it allows us to easily monitor the status of each submitted job. Each job is distinguished with a precise *unique ID number* and the state for each hashing script is reported, by using the starting letters of the following words : *C(ompleted)*, *R(unning)* and *Q(ueued)*. When a job is completed, TORQUE generates automatically two output files. An *error file* that contains potential errors that occurred during execution time and a *result file* that has the produced output after processing. These files are created separately for every job and they are named, based on the individual *ID number* and the script name of the job. In the case on which there are *no errors* or there are *no output* results, these files are still created, but they are empty. They are located in the pre-defined *working directory* and they are generated *at the end*, when the relevant job has been *completed* or *failed*.

The list of *command options* that both TORQUE and Maui provide, is quite long. There are several different options that could be used and there is also the ability to define many different processing queues. An administrator can write dedicate *pbs scripts*, enabling a significant scheduling flexibility. There are even mechanisms that could enable *email notification* after the completion of a specific job, informing in this way directly the *owner* of a scheduled script. In our case, there was not any particular reason for using a more complex approach and therefore, the benchmark of the hashing implementations is performed, by using only *one default queue* and a rather *simple* configuration. Nevertheless, we make use of a complementary command of Maui and we are able to better control and monitor the scheduled jobs. In particular, by using the `showq` command, we get a detailed version of the submitted jobs with respect to all the available resources of our system. As figure 8 demostrates, the queue is divided into *active*, *idle* and *blocked* jobs and there is the *remaining* and *starting time* as additional information. Another important monitoring feature that this command offers, is the percent of processor usage on the total cluster system. Again, in our project this does not have a major monitor benefit, since we have only *4 processors* one in each cluster node. However, in a *high-scale* cluster architecture, this kind of command could constitute an important monitor advantage for the system administrator. Especially, when using the `nodes` option with which you can define precisely the cluster node on which a script will be processed (see A.3), this command obtains a significant meaning. It allows the administrator to know the *exact percentage* of resource usage and in the case on which a single motherboard has *more than one* processor units, it is able to identify the number of processors that are in use.

```
[spyros@pythagoras ~]$ showq

ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

354                  spyros    Running     1 01:00:00:00  Mon Jul  5 21:21:56
355                  spyros    Running     1 01:00:00:00  Mon Jul  5 21:21:56
356                  spyros    Running     1 01:00:00:00  Mon Jul  5 21:21:56
357                  spyros    Running     1 01:00:00:00  Mon Jul  5 21:21:56

     4 Active Jobs       4 of    4 Processors Active (100.00%)

IDLE JOBS----------------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME

358                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
359                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
360                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
361                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
362                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
363                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
364                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
365                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55
366                  spyros       Idle     1 01:00:00:00  Mon Jul  5 21:21:55

9 Idle Jobs

BLOCKED JOBS----------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


Total Jobs: 13   Active Jobs: 4   Idle Jobs: 9   Blocked Jobs: 0
```

Figure 8: Monitoring with showq - Maui

In an overall assessment we could say that TORQUE and Maui covered the resource management of our diskless cluster, in a very *efficient* and *effective* manner. Of course, there are also other resource management tools that could allow someone to have a similar functionality. It is worth to mention that in our initial planning we were considering the usage of tool called *Condor*[21], mainly in order to avoid the installation of two different tools. Condor is a complete workload management system that brings the features of TORQUE and Maui into one application. In practice, it is a *full-featured* batch system that provides job queueing mechanisms, scheduling policies, priority schemes, resource monitoring and resource management. Condor was developed under a research project which was conducted at the *University of Wisconsin-Madison.* Nevertheless, it is only the last years that the software is distributed without any *license restrictions.* A registration is still necessary for downloading the software, but its usage is free and it does not imply any license fees. The fact that Condor seems quite more complex and the registration process for obtaining the software, discouraged its usage in our project. However, Condor co-operates successfully with DRBL and it constitutes a recommended alternative, which could be used when the installation of TORQUE and Maui is not possible.

[21]http://www.cs.wisc.edu/condor/

# 5 Hardware Acceleration for Cryptography

*Hardware acceleration* is a technology that has evolved in the IT industry for quite a long time. It refers to the ability to transfer the processing of a number of desired operations from the *software level* to the *hardware level.* That usually brings significant performance benefits, since the dedicated circuits are able to process faster than the equivalent software implementations. Hardware acceleration can target into many different areas and cryptography is one of them. As the need for security continuously increases, most of the software applications start to consider cryptographic operations as an integral security component. For an individual user the impact on performance is relatively small and there is not a vital need for accelerating cryptographic operations. On the other hand, from a server's perspective there is a fundamental need for performing as fast as possible the expensive cryptographic operations, by maintaining also a *low CPU utilization.* By enabling acceleration the *overall processing time* is optimized and a server is capable to serve more *requests per second.* That enables support for even *more clients* and transforms the feature of acceleration in a productive mechanism for *off-loading* intensive cryptographic computations. In this way, the processing balance that the software is failing to provide in *large scale* infrastructures, can be effectively delivered by *special hardware designs.*

In 2008, Intel released into the market a new family of *integrated processors* for embedded computing[22]. In particular, the Intel EP80579 processor was presented as a promising solution for optimizing cryptography at the hardware level. By embedding a *system on-a-chip (SOC)* technology, the processor was able to process individually the operations related to cryptography and thus, perform better than the equivalent software designs. Intel EP80579 was proved faster with a *low power consumption* and delivered a promising direction for security, with respect to *performance* and *cost* [30]. Of course, Intel is not the only corporation that develops dedicated hardware for cryptography and there are several other companies that released similar products. Another common approach instead of embedding the functionality inside the processors, is to design separate *cryptocards*[23], which can be installed on a variety of motherboards as a common *Peripheral Component Interconnect (PCI)* card. Each of these cards constitutes an independent *cryptographic coprocessor* and therefore, it is able to successfully process intensive cryptographic computations, by providing also an obvious *portability.*

Even though that hardware acceleration for cryptography constitutes an attractive solution for many applications, it still remains on the background of the IT market. The main reason is that except from the companies that target to offer better *quality* services at the *lowest* possible cost, this dedicated hardware does not reveal any major usage advantages for the individual users. If we consider also the fact that this kind of hardware is not so *cheap*, then it is more than obvious that these designs become *company* oriented. By examining the hardware implementations from a clients perspective, we identify that the needs of an every day user with respect to cryptographic operations can be sufficiently covered from any powerful processor. Thus, there are not any important performance reasons that would lead an individual user to buy and use dedicated hardware for cryptography. The introduced embedded technology seems a promising way for offering the benefits of hardware acceleration to a broader community of users, but at the moment the adoption of these products covers mostly needs for *specific* infrastructures.

In our project the initial idea was to include the *Intel EP80579 processor* for enabling its cryptographic acceleration to all the remote nodes. In this way, we could scale the processing of hashing, but also measure the benefits of performance for the individual areas of applicability. Unfortunately, we were not able to use acceleration for several reasons and therefore, we used the Intel Pentium M processor for benchmarking. The main reason for which we decided not to enable the acceleration feature, was the need for developing a small *shim* for each hashing applications. A *shim* is a small library that is used to integrate *changes* or *function call redirections* into an existing *Application Programming Interface (API)*, focusing mainly on *compatibility* purposes. As we will present in the following section, a shim is the only way to enable the acceleration of Intel EP80579 processor in an application that has already been built based on its *own customized* API. By embedding a shim into a customized API the necessary cryptographic function calls can be properly transformed, enabling in this way the *accelerated* processing. The difficulty of developing a shim depends on the application needs and it is strictly related to the complexity of cryptographic computations that must be accelerated. In our case, the difficulty of developing a shim for each hashing command was not so complex. Nevertheless, the required amount of time for *developing* and *testing* a shim for each of the supported hashing commands was considered a reasonable obstacle. Imagine, that for enabling the acceleration into each of the hashing commands we would have to modify the original source code of every implementation and then recompile each command. Furthermore, algorithms such as *Skein*, *Whirlpool* and *Tiger* were not supported by the hardware acceleration and thus, they could have not been accelerated, even if we had chosen to develop the required shims.

By presenting this chapter, we aim mainly to present the opportunities that arise for optimizing further the cluster model. The intensive processing of cryptographic operations can be optimized by simply embedding acceleration on

[22]http://www.intel.com/design/intarch/ep80579/index.htm
[23]http://www-03.ibm.com/security/cryptocards/

each desired implementation. For areas such as *cryptanalysis* and *digital forensics*, scalability and computational power hold an important role for overcoming time constraints. Therefore, if the proposed diskless architecture is proved to be efficient *without* the acceleration, then it is quite possible that with the acceleration enabled will yield a *noticeable* performance difference. Considering also the fact that the number of cluster nodes needed for achieving great processing power could be decreased when acceleration is present and thus, such an infrastructure becomes sufficiently attractive. Even for implementations that are focusing mostly on commercial usage, such as secure *Web Servers* or *Virtual Private Networks (VPNs)*, this approach is believed that could still contribute with the proper modifications. The following sections present the *high-level model* of the Intel EP80579 processor and they discuss the most important concerns about hardware accelerators. Finally they reveal some of the existing trade-offs, between *performance* and *cost*.

## 5.1 Intel EP80579 Processor

As it is fully described in [31], the Intel EP80579 processor is based on an embedded technology which is integrated successfully in a *single* chip. In particular, in the case of the Intel EP80579 that chip is an Intel Pentium M processor that co-exists with an embedded technology, called *QuickAssist Technology*. In a high-level description, QuickAssist is combined with a set of *Intel Architecture (IA)* instructions and a number of other *on chip* hardware capabilities, delivering in this way mechanisms for accelerating cryptography. In particular, QuickAssist constitutes the highest layer that it is visible to the end-user and in practice, it is the *dedicated API* for calling cryptographic acceleration [32]. Below this layer there is an acceleration driver, which is responsible to handle all the *received requests* from the QuickAssist API and pass them directly to the hardware for processing and vice versa. The acceleration driver is probably the most important component, since it represents the *intermediate connection* between the actual hardware and the software. It enables the *message exchange* and provides the necessary functionality for configuring the hardware according to the generated requests.

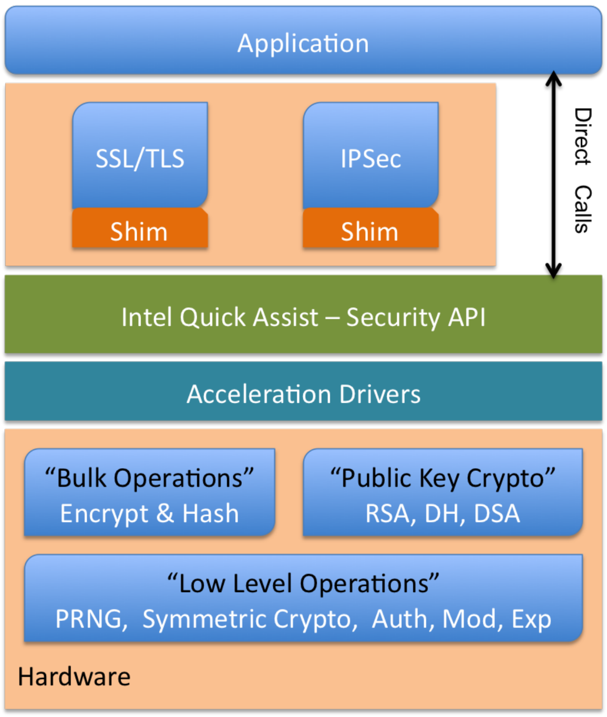

Figure 9: High Level Model

As the above figure demonstrates, the hardware of Intel EP80579 supports several cryptographic operations, such as *symmetric* and *asymmetric* cryptography, *random* number generation, *hash* algorithms, *modular* and *exponential* arithmetic. All these operations are performed directly on the specially designed circuits, decreasing significantly the processing time. However, in order for an application to generate the proper requests for the hardware acceleration, it should integrate the necessary QuickAssist API calls. As we mentioned earlier this API constitutes the only link for communicating with the hardware and its usage is mandatory. In practice, there are two possible ways that could be followed for making the needed hardware calls. The first solution would be to implement the entire application based on the provided *QuickAssist API* and integrate directly every function call for acceleration. Nevertheless, this method is *quite restrictive* for general purpose applications and usually customers tend to have their own versions of software, which are not willing to re-implement. Therefore, most of the time a *shim* is developed and integrated into the general security protocol stacks, such as *TLS/SSL* or *IPSec*. This shim enables the acceleration for every application that makes use of the protocol and thus, provides a more flexible way to use the QuickAssist API. Nevertheless, even this approach could lead to a relatively *limited freedom* for configuration. Usually, most of the companies are using their own *customized* protocol stacks and therefore, there could be cases on which significant changes are needed in order to embed a shim. For avoiding any incompatibility issues, there is continuous communication between the client company and the manufacturer, during the development phase of the API for acceleration.

It is obvious that hardware acceleration is not a *plug and play* feature and further efforts are required, in order to obtain the desired processing benefits. However, the concept remains quite straightforward and involves a dedicated Application Programming Interface (API), which as a final deliverable becomes the base for enabling acceleration. Then, it is up to the user's skills to make it suitable for serving his own needs. Our belief is that in particular occasions, when the operations are quite simple, such as the multiple hashing iterations, the usage of the dedicated API does not introduce any significant complexity. Small implementations like *cryptanalysis* and *digital forensics* tools (e.g. CoWPAtty, dcfldd) are expected to be able to embed the proper API calls easily enough.

Of course, hardware acceleration does not come without limitations. In the case of Intel EP80579, acceleration performs much better to an *asynchronous* packet processing, but significantly slow down in particular cases on which there is a need for *synchronous* requests. With the term *asynchronous requests* we refer to a number of continuous acceleration requests to the hardware, which do not require an immediate response. This kind of requests obtain the *maximum advantage* from the capabilities of the hardware acceleration. On the other hand, *synchronous requests* are requests which require directly a response after processing and this in some cases, turns to be a quite expensive operation for hardware. The key point that might cause a synchronous request to be processed even slower from software, is the *packet size* for processing and the *lack of computational intensiveness.* Imagine, the example of the dictionary file that contains a list of words and suppose that we want to perform an MD5 operation for each of the words contained inside the file. The size of data for hashing is almost negligible and thus, it does not introduce a very intensive computation. If the dictionary file contains *10.000 words* and the results are asked synchronously, that means that for every word that is hashed the digest should be returned directly back to the user. That introduces a *transfer latency* and it is probable that the hardware acceleration would be processed *slower* than software. The reason lies in the fact that the time spent for *generating* and *transferring* the requests *from* and *to* the lowest level is much more than processing the request in a higher software level. On the other hand, if these 10.000 words are hashed asynchronously and the results are returned to the user after the entire processing is finished, then in that case, the hardware acceleration would remain *faster* than software. Intel EP80579 was designed to perform best under intensive processing and significant amount of workload. If there is a need for processing *non-intensive* cryptographic operations that include a *small data* size for processing in a *synchronous mode*, then the hardware acceleration is not considered a very efficient way and software is recommended. This behavior can also be related to the *time-memory* trade-offs when the computational task is referring to cryptanalysis purposes. However, it finds a much broader meaning on the area of security protocols that perform *synchronous* message exchange. At the moment, it is considered a point for further optimization and it is believed that the upcoming generation of hardware will offer better performance even for synchronous requests, but without entirely eliminating the problem.

## 5.2 GPUs vs CPUs

As we already discussed in section 2.4.4, the performance that could be achieved by using *Graphics Processor Units (GPUs)* instead of *Central Processor Units (CPUs)* is quite noticeable. Nevertheless, that does not mean that GPUs are always able to successfully substitute a main processor, since they cannot perform similarly under any computational task. If we compare the acceleration features of Intel EP80579 with today's GPUs, we will probably conclude that GPUs are faster. However, we consider this comparison not valid, since the Intel EP80570 is based on an old CPU generation. On the other hand, if we compare *GPUs* with latest and fastest processor units, even then, we will find that GPUs are quite faster and that is expected. Graphics cards are designed to support in parallel processing, for delivering adequate power for handling graphics. Moreover, they include their own independent memory and that increases even more the processing speed. In our opinion, GPUs should not be considered as a hardware that could be used for the same purposes as CPUs. Indeed, GPUs might be faster for *non-complex* and *repeatable* intensive operations, however there were not initially designed for serving demanding cryptographic operations and that is considered a quite valid point for using hardware accelerators, especially for commercial applications. Do not also forget the fact that for running an implementation on a GPU, a user still needs to modify the application for supporting the proper *low-level calls* to the graphics card. This is not an easy task and it involves several source code modifications. We will of course agree that introducing GPUs as means for cryptanalysis purposes, reported at least *10 times* faster processing from the latest CPUs. This is very attractive for people that are willing to develop the required implementations for obtaining the benefits of parallelized processing. By considering also an existing ability to use *more than one* GPUs simultaneously and the fact that at the same time the main CPU is *free for usage*, this approach introduces an excellent way for obtaining the maximum processing capabilities in a single motherboard. However, an obvious drawback for using GPUs in a regular base, is the lack of a structured API for off-loading cryptographic operations, such as the one that Intel EP80579 provides. Moreover, there are many graphics cards which are not programmable for tasks out of the initial designed scope and this is another obstacle that induces serious limitations to the choices that exist for independent customized GPU programming.

Another interesting structure for *high-performance computing* is the usage of *PlayStation 3 (PS3)* game consoles. Ecole Polytechnic[24] has already created an entire cluster system based on PS3 devices, which constitutes an integral part of the *Laboratory for Cryptographic Algorithms (LACAL)*. This system is used to conduct research on the field of *cryptanalysis* and it mainly takes advantage of the *Cell processor*, on which PS3 consoles are built. In practice, Cell is a microprocessor architecture which was jointly designed by IBM, Sony and Toshiba and aimed to accelerate *multimedia* and *vector processing* applications. The key factor of this architecture is its ability to efficiently perform *in parallel* processing, enabling in this way several other hardware features that could be used for processing intensive computations, such as cryptographic operations.

### 5.2.1 Performance vs Cost

An important factor that we did not discuss yet, is the appeared *trade-offs* between performance and cost. By looking strictly only into the cost for buying the hardware, the majority of GPUs will be proved less expensive compared to the latest generation of CPUs and definitely cheaper from the equivalent hardware accelerators. If someone take also into account that GPUs come with a desired high processing power, then he will reasonably argue that buying and using a GPU when the task allows it, will be a *sensible* and *efficient* approach. Especially, for computational tasks that target in particular intensive operations (e.g. cryptanalysis), that claim might find a valid ground. However, there is an important and not so obvious drawback, that hides additional concerns with respect to the overall usage cost. Graphic cards offer great processing power, but at the same time operate with significant *per watt* power consumption, which increases the cost in an indirect way. The modern GPU models demand between *110* and *270 watts* from a power supply and this is almost as much power as all the other components of the system require, with CPU included. While CPUs designs have focused on improving *performance per watt*, GPUs continued to increase their energy usage. Imagine, that the latest CPUs generation consume at least *2 times less* energy from GPUs, introducing cost benefits which are quite visible when the hardware should operate in a continuous way. On the other hand, the dedicated hardware acceleration aims to *optimize* even more the per watt consumption that CPUs offer, by maintaining similar processing capabilities with the ones that GPUs provide. Therefore, there is a *non-negligible* trade-off that appears in the form of performance per watt and should be seriously considered before any further decisions are made. In our opinion, if the cost is not of a major importance, then the potential usage of GPUs is attractive. There are cases such as research projects on which maintaining a low cost infrastructure is not the highest priority and other objectives define the selection of hardware. However, in any other case, the cost that GPUs introduce due to energy needs, makes them an inefficient option for cryptographic computational tasks and better hardware solutions should be preferred.

### 5.2.2 Uprising Technology

The newest generation of hardware accelerators aims to change completely the capabilities and the performance of the provided acceleration. The upcoming releases will include additional features that target in broader security areas than cryptography and will implement mechanisms that enable *Network Intrusion Detection* and advanced *Compression Algorithms*. Moreover, the existing cryptographic operations will be updated to support the latest algorithms and the new security standards, such as the ones that implement *Elliptic Curve Cryptography (ECC)*. The main goal is to *decrease* the per watt consumption even more, but at the same time increase the processing performance, by using the latest generation of CPU technology. Moreover, there is an effort to make the integration of acceleration as easy as possible for the end user, by optimizing the delivered Application Programming Interface (API). It is almost sure that this technology will still serve mainly *large-scale* infrastructures, but there is a belief that if the cost of obtaining the hardware is not overcome the standard commercial limits, this technology will have chances to enter the houses of normal users. In any case, we believe that hardware acceleration is evolving in a promising way and it is a technology that could certainly serve the need for fast intensive computations at an affordable cost. If someone also decides to combine this technology with a cluster diskless architecture, he could end up with a quite powerful system that would be able to provide sufficient in parallel processing for several different security tasks.

[24]http://www.lacal.epfl.ch

# 6 Evaluating the Diskless Model

By using all the systems described in section 3.2, we are able to run *2 basic* test cases. These test cases are based on all the installed hashing implementations and they allow us to collect sufficient data for our analysis. The inclusion of *timestamps* gave us all the needed information for the processing times and introduced a certain amount of *flexibility*. Archimedes, Euclid, Pythagoras and Plato represent all the different hardware and architectures that are used in our experiments. We recall that Archimedes is one of the *cluster nodes*, Euclid is a *stand-alone system* with capabilities identical to Archimedes, Pythagoras is the *main node* of the cluster and Plato is a laptop equipped with a *faster processor* compared to all the other systems. Both test cases are executed on all 4 systems and a special case for parallelized processing into the cluster nodes is defined for each test.

In the *first test case*, we compute hashes for the entire database of image files that was described in section 3.3 and we try to investigate the reported processing time, for a CD image, a DVD image and the entire database of image files. The results are obtained by running the *allhashes* script written for image files (see Appendix A.1). We insert the proper timestamps and we calculate the processing time for each of the image files separately, but also the total amount of time for hashing the entire database. Afterwards, we use the *13 single* scripts which were written for the image files and we divide the hashing of the entire database by *individualizing the execution* of hashing commands and running every command for the entire database, individually. In this way, we are able to *parallelize* all the hashing commands to the 4 cluster nodes in a very simplified manner, but without splitting the workload. In practice, this is not quite common and usually the workload should be divided in order to perform an optimal parallelization. However, in our case we decided to proceed with and *easy* and *straightforward* approach, in order to avoid developing complex software that would be able to parallelize the hashing of image files. This will affect to some extent the obtained overall times for the overall in parallel processing of the database, but it would give us an idea about the performance benefits. More precisely, the way that the hashing commands are scheduled on the different cluster nodes influences significantly the overall time and therefore, we expect a latency. If we assume that the *SHA-2 family* and *Skein* are the most expensive hash algorithms and we schedule the equivalent commands together in a cluster node, we could end up waiting for a node to finish the hashing of the database, while the other ones have already completed their task and they are *free for usage*. In practice, the chosen method for scheduling introduces an obvious *balance problem* with respect to the expected processing cost of each hash algorithm. In our test, we tried to resolve this problem as efficiently as possible, by equally distributing the expensive algorithms into the 4-cluster nodes and thus, maximizing their in parallel usage. Although that our scheduling approach for the image files is not considered optimal, we believe that we will be able to identify the advantages for running *in parallel* the hashing commands. Further optimizations are proposed in this area and we briefly discuss some of our recommendations for future work in chapter 8.

The *second test case* refers to the dictionary file and more precisely, to the hashing operations which are performed for the *10.200 words*, by using all the installed hashing commands. Again, the idea is similar to the one that we applied in the first test case. We use a special written *allhashes* script that reads the dictionary file *word by word* and we compute all the hashes for each of the implementations. For parallelizing the processing, we split the word list of the dictionary file into the number of the provided cluster nodes. More precisely, we create *4 independent files* with 2.550 words each and thus, we successfully divided the *10.200 words* of the initial Dictionary.txt into *4 equal* data inputs. In comparison to first test case, this time we *divide the workload* instead of *individualizing the execution* of hashing commands, obtaining the maximum advantage of parallelization from our model, without writing any special software. In order to satisfy our curiosity, but also to prove that our assumption for the fist case was correct, we perform a simple test which is not included in our data analysis. By individualizing the execution of hashing commands and maintaining a single file with 10.200 words for all the cases, we identify that indeed the way that the hashing commands scheduling is performed when the workload is not split, influences the total overall time.

During our entire experiment we monitor and record also the *CPU utilization* for each of the hashing commands. Even though that we do not provide any statistics for the CPU utilization, there are cases in which we refer to the *percentage* of the processor's usage, since it was the cause of some unexpected results. Note here, that the Operating System (OS) is responsible for *controlling* and *defining* the CPU utilization for each process. In our case, Linux is responsible for handling the CPU utilization for each of the installed hashing implementations. Of course, the way that an implementation is written affects also the utilization and if it is not well implemented that could lead to improper usage of a processor. However, from our benchmark results we have only some doubts about the implementation of *md5sum* and we consider all the other implementations sufficiently stable with respect to the mechanisms for CPU utilization. Another important aspect that we discovered, which is related to utilization, is a difference between the way that the utilization is performed between *a dual core* processor and a *single core* processor. From our systems, Plato and Pythagoras are the only machines equipped with a dual core processor and in that case the utilization gets even trickier. The OS is responsible for defining the usage on both cores and decides how the implementation will be processed in the two available cores. In our benchmark, at most of the tested cases, the percentage of processor usage remained over 90%. Nevertheless, there were specific cases such as the one of *md5sum* command that yielded higher processing times, due to a CPU utilization at 35%. Moreover, Plato at some points demonstrated *stable processing times* for a majority of hashing commands, while the CPU utilization remained at a level lower than 45%. Although we tried to control the factor of CPU utilization *by adjusting* the scheduling priority of each process, this seemed to have *no effect*. As the reader will notice, in the analysis of the graphs that follows, variances are introduced and sometimes these variances are specifically related to the CPU utilization of each system. In an attempt to clarify the exact points on which the low usage of processor caused higher processing times, we discuss in detail the obtained results by arguing also for the reasons of appearance. In section 6.1 we present the results obtained from the test case of *image files*, by analyzing individually the case of the *CD image*, the *DVD image* and the benefits of parallelization for hashing an *entire database* of image files. Section 6.2 describes the test case of *dictionary file* and in particular the hashing of the *10.200 words*. Moreover, it demonstrates the advantages of dividing the workload equally into the number of available number of cluster nodes and makes a comparison with respect to the processing capabilities of the other systems. Finally, in section 6.3 we answer most of our research questions which we have formulated in 3.6, to the extent that our results allow us.

## 6.1 Image Files

Image files are considered an interesting way to measure both performance of the network and the behavior of hashing implementations under a large data input. Moreover, image files and especially the hashing of an entire database of image files can be successfully linked to the area of digital forensics. More precisely, there are cases on which a forensic investigator needs to compute a single hash of an entire hard disk image, in order to prove that the examined hard disk was identical to the one which contained the initial evidence. Therefore, we believe that this test case could have an additional interest for forensic investigations, if our *4 node cluster* could be proved significantly faster for hashing entire image files. The interesting aspect of this specific benchmark is how the different systems will behave, based on the installed *amount of memory* and what would be the *impact of reading* from a remote shared hard disk.

### 6.1.1 Hashing a CD Image

The following graph shows the performance that *Archimedes*, *Euclid*, *Pythagoras* and *Plato* yield for hashing a CD image (630 MB). At a first glance, independently from the used system, there is a number of hashing algorithms that report more time consuming processing.

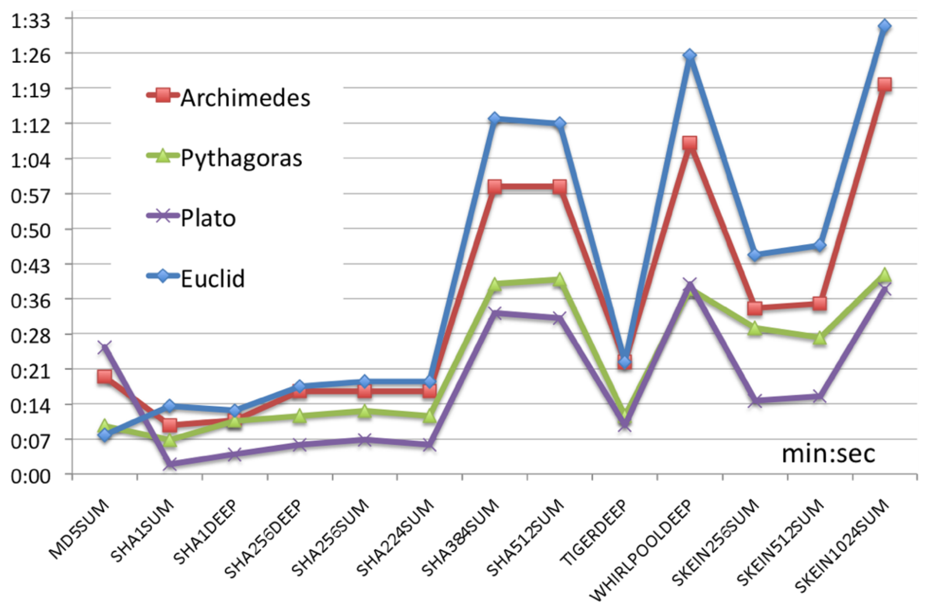

Figure 10: Hashing a CD Image (630 MB)

In particular, *sha384sum*, *sha512sum*, *whirlpooldeep*, *skein256sum*, *skein512sum* and *skein1024sum* seem to be the implementations with most expensive computations. The *highest time* for hashing the CD image is reported by *skein1024sum* executed on Euclid, close at *1.5 minutes* and the *lowest time* is reported by *sha1sum* executed on Plato, close to *3 seconds*. The general performance of hashing implementations provides a *performance behavior* that is maintained independently from the used system and only in some rare cases this behavior is not preserved. Especially for the hashing commands that do not introduce a great computational intensiveness, such as *md5sum*, *sha1sum*, *sha1deep*, *sha256deep*, *sha256sum* and *sha224sum*, we can argue that there are no significant differences for hashing on the same system. By examining individually the systems, we find that Plato constitutes the *faster system* for all the hashing implementations, with only exception of *md5sum* which is the slowest due the CPU utilization that appears on Plato and remains close to 35%. Pythagoras follows as the next most powerful system with small difference compared to Plato and with an exact match for the cases of *tigerdeep* and *whirlpooldeep*. Although that at this point we expected that the next faster system would be Euclid, since it represents a *stand-alone* system with an attached hard disk, Archimedes seems to be *faster* for hashing commands that are computationally expensive. On the other hand, for commands that are not so computationally expensive Archimedes is proved *slower* than Euclid. The only way that we could reason about the fact that Archimedes is faster, is the additional 512 MB of memory with which it is equipped. In previous sections we mentioned that we expect that the CD image would be able to be directly loaded entirely into the memory, however it seems that this is not the case. The file probably is read remotely from the cluster node and the data are buffered, while processing takes place. This means that when a cluster node wants to process a file, then it initiates the proper read calls to the remote hard disk and by using the *Network File System (NFS)* protocol it starts receiving the data through the network, in a continuously streaming way. If we assume that both hashing implementations and the clustering tools have mechanisms to control the buffer size based on the available memory resources, then it might be acceptable for Archimedes to perform better than Euclid. However, any further assumptions at this point are considered *non-valid* and further investigation is recommended, with respect to the data transferring mechanisms which are implemented by the clustering tools and the NFS protocol. Finally by taking a look into the performance of the *Skein* algorithm compared to the *SHA-2 family*, we clearly see that *skein512sum* is almost *1,5 times* faster than *sha384sum* and *sha512sum*, while *skein256sum* is almost *1,5* times slower than *sha224sum* and *sha256sum*. These results are maintained similar on every used system and they are quite close to the performance numbers that the authors of the skein algorithm reported. Finally, *skein1024sum* is indeed proved more expensive than *sha512sum*, which was expected since skein1024sum uses a message block of *1024 bits*. In particular, in the case of Archimedes and Euclid our results verified a performance almost *2 times slower* than *sha512sum*, but on Pythagoras and Plato skein1024sum reported a performance close to the one of *sha512sum*. We assume that the reason that *skein1024sum* yields similar processing time, is based on the fact that Pythagoras and Plato have *multi-threading* capabilities, on which the skein design probably behaves better.

### 6.1.2 Hashing a DVD Image

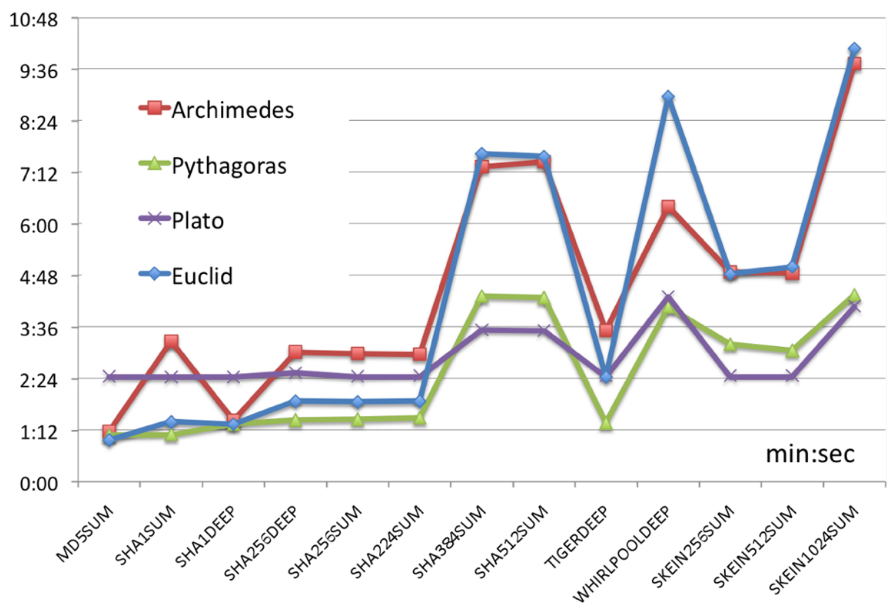

Figure 11: Hashing a DVD Image (4 GB)

Graph 11 presents a similar test that we performed for the CD image, but instead it uses a DVD image which is *7 times* larger (4011 MB). Again, we use *Archimedes*, *Euclid*, *Pythagoras* and *Plato* as our platforms for benchmarking. By examining carefully the graph we can easily see that we have an *expected increase* of the processing time for every hashing implementation, due to the larger data input we provide. The symmetry which was identified among the different systems at the case of the CD image is still preserved for most of the hashing commands. The *highest* processing time for hashing a DVD image file is reported again by *skein1024sum* executed on Euclid and it is approximately *10 minutes* and the execution of the same command on Archimedes reports a similar performance. On the other hand, the *lowest* processing time is reported by executing *md5sum* on Archimedes, Euclid and Pythagoras and it is close to *1 minute*. An interesting remark is that Plato was restricted due to the CPU utilization, which remained below 45% for all the *less-expensive* hash algorithms. This resulted in a worst performance, even if the commands were executed on a faster processor. The main cause lies in the hashing implementations and the OS, which failed to make *efficient usage* of the dual core processor and thus, the processing time remained at unexpectedly high levels. In the case of the *more-expensvie* hash algorithms Plato takes advantage of its powerful processor and at most of the cases reports better processing times, but without again utilizing the CPU more than 70%. Pythagoras reports the *lowest* processing times for almost every hashing command and this is expected since Plato was using in the best case only the $\frac{2}{3}$ of its processing power. Finally, by comparing Archimedes with Euclid, we identify that both systems for the *expensive* hash algorithms report results that *almost match*. On the other hand, for the *less expensive* algorithms Euclid is proved faster than Archimedes and that's probably due to the introduced *network latency*. The system calls for *reading* and *transferring* the DVD image file, in combination with the fact that the amount of memory on both systems is about *3-4 times* smaller than the image size, causes a reasonable performance difference. A noticeable remark is that the execution of *sha1deep* on Archimedes yields a *lower* processing time than *sha1sum*, which reaches the processing time that Euclid reports. After investigating this behavior, we found out that the reason lies again in the *CPU utilization*. More precisely, Archimedes maintains an utilization close to 75% for all the *less-expensive* hashing commands, but in the case of *sha1deep* the utilization reaches 96%, which in practice give us the lower processing time appeared on the graph.

### 6.1.3 Hashing the Database

The following graph demonstrates how our *4 systems* behaved when they were asked to hash an entire database of image files.

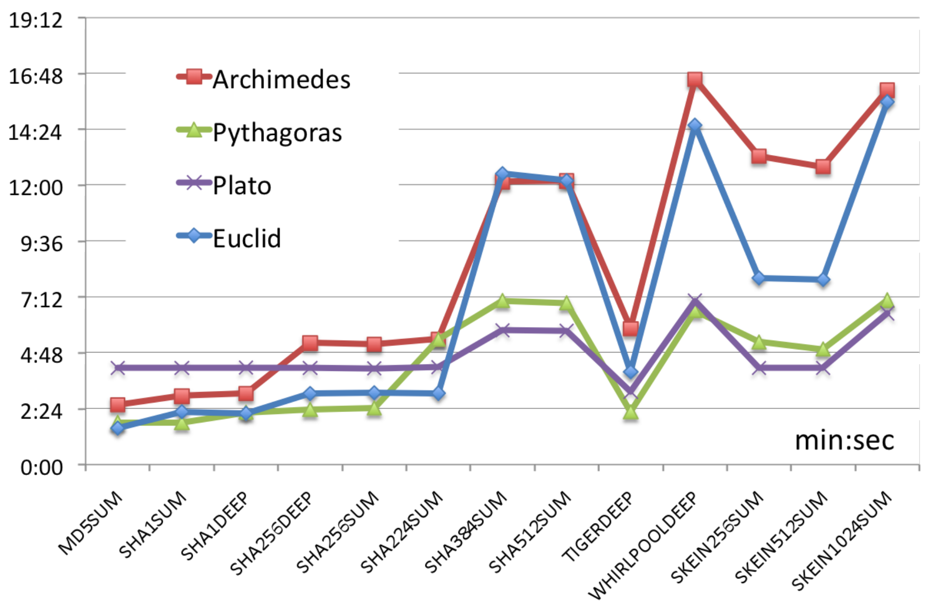

Figure 12: Hashing the Database (6,5 GB)

The database is formed by all the image files presented in figure 3 and more precisely, the database contains *4 CD images* and *1 DVD image*, which give us approximately a *total size* of 6,5 GB as a data input for each of the hashing commands. Figure 12 reports the *overall time* for hashing the entire database with each of the benchmarked commands and allow us to make some valid conclusions with respect to a more realistic scenario. By analyzing the graph we see a lot of similarities with the case of the *DVD image* file and this is not awkward, since both refer to the hashing of a large data input. Of course, in the case of the database the total processing times have been increased accordingly to the given workload. The highest processing time is reported by *whirlpooldeep* executed on Archimedes and it took almost *16,5 minutes* to complete the entire hashing operation. Even though that *skein1024sum* does not give the *highest* processing time, it remains very close to the *whirlpooldeep* performance and we believe that this occurred again, due to the *instability* of CPU utilization. The differences between Archimedes and Euclid become more obvious and *sha1deep* behaves normal, without introducing any further unexpected results. Pythagoras performs the faster processing for the *less-expensive* hash algorithms and it competes with Plato for the performance on *expensive* hash algorithms. An interesting remark on this graph is that *skein256sum* and *skein512sum* report an obvious difference between Archimedes and Euclid, which can be partially explained by the *CD image* graph. In practice, even if the DVD image does not introduce any difference, the *4 included* CD images seem to introduce a latency for Archimedes, which in the end is summed up into the difference that is presented on the database graph. This remark give us an ability to argue that most probably, there are *two major issues* that influence the *remote* performance compared to the *local.* The first factor is the file size that must be hashed and whether this file is larger than the amount of memory which is installed on each of the benchmarked systems. An equally important factor is how *efficiently* a hashing implementation was designed and what is the behavior of the mechanisms that are used for *transferring* and *buffering* the streamed data. Both of these issues are considered vital points for performance and therefore, additional research is required.

### 6.1.4 Cluster vs Units

The following graph represents the main part of our evaluation for this specific test case and it refers to the benefits that someone can obtain by parallelizing the hashing commands. It reports the total time for completing the hashing of the entire database of image files, by using all the benchmarked hashing implementations. It uses *all the available* cluster nodes and gives a comparative assessment of performance with respect to the rest of the architectures. We recall that the cluster nodes refer to *Archimedes*, *Aristotle*, *Thales* and *Democritus*, which are identical systems with *slightly different* resources, as presented in 3.2.

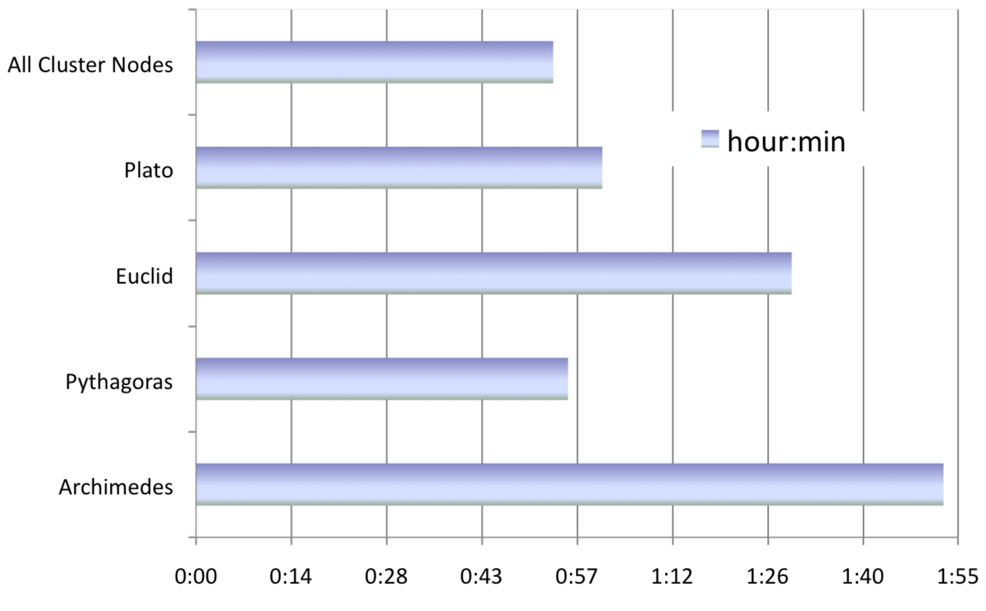

Figure 13: Cluster vs Units (entire database - 6,5 GB)

The results that are presented in the above graph, they do not introduce any surprises regarding performance, with exception the performance of Plato which remains quite poor, because of the imbalance in the CPU utilization. Archimedes appears approximately *1,2 times* slower than Euclid and that proves that indeed the *remote diskless processing* induces a certain level of *latency.* Pythagoras and Plato report significantly faster processing than Euclid and Archimedes, and they verify that the installation of a faster processor leads to better processing times. The *highest* processing time is reported by Archimedes and it is close to *1 hour* and *50 minutes*, while the *lowest* processing time is reported by the cluster system and it is close to *52 minutes*. Parallelization on *4 cluster nodes* is proved to be almost *1,7 times* faster than Euclid and slightly faster than Plato and Pythagoras. At this point we should mention that Plato is probably expected to be *faster* than the 4 cluster nodes, since its CPU is not fully utilized. However, we should take also into account that the parallelization is not performed in the most optimal way and thus, we believe that the processing time reported from the nodes could be also decreased.

## 6.2 Dictionary File

The usage of a *dictionary file* constitutes the second test case we conducted with our cluster architecture. The idea behind the hashing of thousands of words is to explore the *network latency* from reading a file remotely, but also see the capabilities of each hashing command for a *small data input*. Another interesting point for investigation is the behavior of each processor, since it is known that for small packet sizes a processor that performs an intensive repeatable operations (e.g. in a form of loop) at some point will reach its *highest* CPU utilization point, even if the nature of a single hash computation is not so intensive. This point will signify the maximum of capabilities that a processor can achieve and give an overall picture of the maximum benefits that someone can obtain with respect to the processing times. However, in our test we are mostly interested in analyzing the benefits that we can obtain for parallelizing the workload and therefore, this kind of analysis will not be presented in this document. As it was mentioned in 3.3, the average data size for processing is maintained approximately at *8.5 bytes per word* and this constitutes the input for each of the hash functions for a single call. This test case aims to examine the performance of the *cluster model* for transferring this negligible data size on a continuous way. This performance could later be related with *off-line brute force attacks* or *generation of pre-computed tables*, such as the ones discussed for the WPA-PSK in section 2.4. We decided to proceed without conducting any cryptanalysis attacks for two main reasons. First of all the amount of cluster nodes that we provided is not sufficient for reaching the necessary amount of processing power for performing brute force attacks and thus, it would not be possible to obtain interesting results in a realistic time frame. The second and maybe the most important factor, was the fact that hardware acceleration could not be enabled and the most promising chances for significant additional power were excluded. Nevertheless, we believe that the following sections will give to the reader enough information to formulate a personal opinion, about the general capabilities of the diskeless architecture.

### 6.2.1 Hashing 10.200 Words

The following graph presents the performance of all systems for hashing the *10.200 words* by using each different command. A first noticeable remark, is the *quite poor* performance of skein implementation.

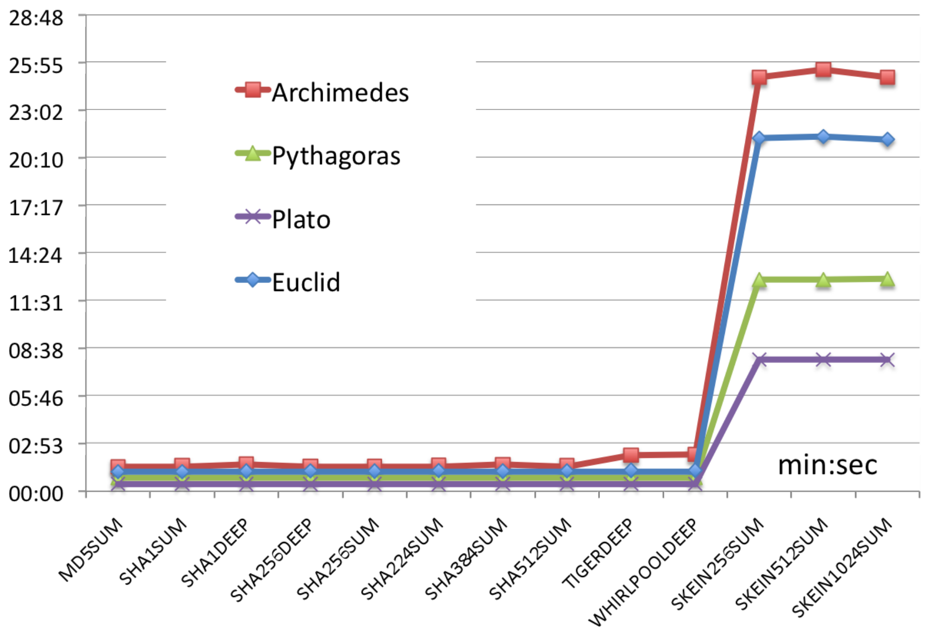

Figure 14: 10.200 Words

This unexpected behavior initiated a further investigation and after several tests, we successfully identified that the reason of appearance lies in the *interpreted nature* of the skein python scripts that we used. As we already mentioned at several points in this document, python is an interpreted programming language and that can introduce significant delays on processing. By performing a simple test for a single hash on the word *hello*, we are able to successfully identify that the cause of the poor processing that skein yielded, is due to the *python scripts*.

```
[spyros@node_archimedes02 ~]$ time skein256sum hello

c620e63c2c148e1e4836ede136801bfecd587f9f8d53d902d4e81d1a2deb2816  - hello

real    0m0.144s
user    0m0.104s
sys     0m0.031s

[spyros@node_archimedes02 ~]$ time sha256sum hello

5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  - hello

real    0m0.004s
user    0m0.001s
sys     0m0.003s
```

Figure 15: Skein - Hashing a single word on Archimedes

The above figure shows that the processing time which Archimedes needs in order to to compute the *skein256sum* hash of a single word is *144 milliseconds*, while for *sha256sum* implementation takes only *4 milliseconds*. The difference is enormous and when we try to compute the hashes for 10.200 words, this performance flaw becomes visible. If we calculate the total time that Archimedes will take to hash 10.200 words by using the *skein256sum* implementation, we will have the following result: $144 \times 10.200 = 1.468.800$ milliseconds. This is equivalent to approximately *24.4 minutes*, which is similar to the processing time reported for *skein256sum* and Archimedes, in figure 14.

```
[spyros@pythagoras ~]$  time skein256sum hello

c620e63c2c148e1e4836ede136801bfecd587f9f8d53d902d4e81d1a2deb2816  - hello

real    0m0.073s
user    0m0.065s
sys     0m0.008s

[spyros@pythagoras ~]$  time sha256sum hello

5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  - hello

real    0m0.002s
user    0m0.000s
sys     0m0.001s
```

Figure 16: Skein - Hashing a single word on Pythagoras

In order to exclude any further doubts and verify that indeed our statement for the behavior of the skein python scripts is valid, we conducted the same test on Pythagoras. Again, the *skein256sum* implementation took *73 milliseconds* to compute the hash of a single word, while *sha256sum* needed only *2 milliseconds*. By calculating the total time for the 10.200 words we obtained the following result: $73 \times 10.200 = 744.600$ milliseconds. This number is almost equal to *12.4 minutes*, which is really close to the total processing time presented in figure 14. Someone could easily wonder why we did not get the same kind of *latency* in the case of image files. The answer is rather simple and it is based on the fact that each time that an image is hashed, we have only a *single call* to the python interpreter. This call introduces a *negligible* time processing cost in comparison to the time that it takes to hash the entire image file and thus, there is not an obvious *performance cost*. It is obvious that the python language and any other interpreted language is possible to introduce performance drawbacks, which apparently seem that does not exist on languages such as C. A guaranteed way to get more reliable performance for the hashing of words would be to use a different implementation of skein, which is written in another programming language that is not interpreted. A good proposal would be the *optimized C* code that the authors provided.

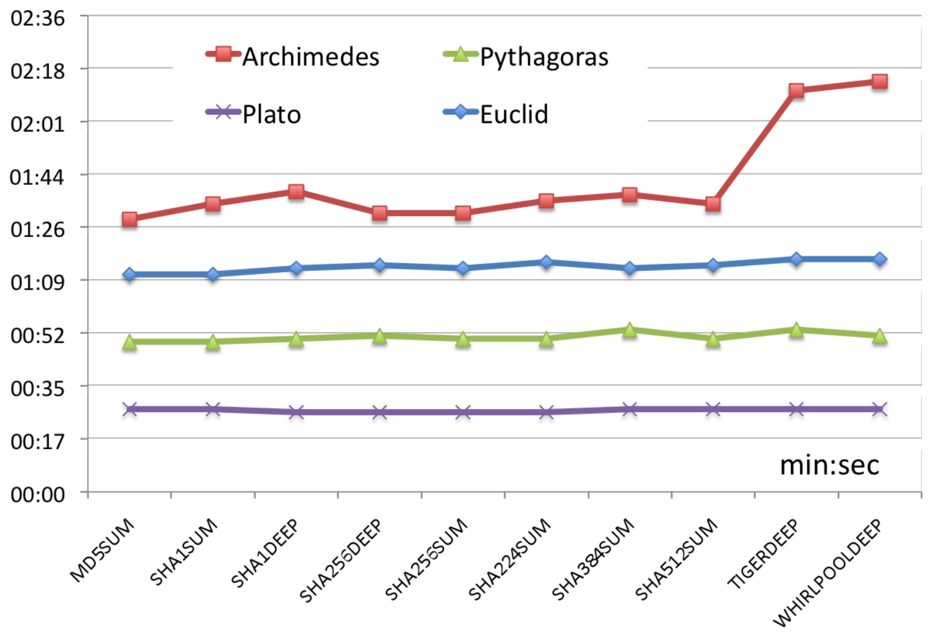

Figure 17: 10.200 Words - Zoomed Graph

In an overall assessment, we could see that all the hashing commands except *Skein* behave in a very *stable manner* in the same system. The following graph demonstrates a *zoomed version* of the previous graph, that excludes the skein commands. An obvious observation is that on this test case the difference on the capabilities of the processors is more than visible. Archimedes reports the *slowest* processing times and Plato is proved as expected, the *fastest* processor. In particular, Archimedes is proved approximately *1,3 times slower* than Euclid, except for the *tigerdeep* and *whirlpooldeep* commands, for which it reports even slower performance with a ratio that reaches almost *1,7 times* slower processing. It is worth to mention that Archimedes appears to have a small scaling among the different commands that becomes bigger for the *tigerdeep* and *whirlpooldeep* implementations. This is believed to be caused by the remote calls needed to access the dictionary file that lives in the *shared hard disk*. Finally, Plato is proved almost *2 times faster* than Pythagoras, verifying in this way our prediction that a faster processor will result in faster processing.

### 6.2.2 Cluster vs Units

The following graph presents the benefits of parallelization that someone can obtain by dividing the workload to the cluster nodes and it performs a comparison with all the other systems. Our intention is to show that by splitting a word list of 10.200 words into *4 equal* parts of 2.550 words each, we are able to process faster and take the maximum advantage of parallelization, without developing any dedicated software based on the *Open Message Passing Interface (OpenMPI)* library. An important remark is that the performance flaw which was introduced by the skein python scripts will affect the total processing time. More precisely, the times reported on the graph are much higher than the ones that someone should obtain in reality. By avoiding to use a python implementation, a significant difference should be reported for every system. Even though that the numbers are not accurate, the ratio between the different systems should remain the same. The latency that skein introduced, is present in every system and therefore, that affects similarly the total processing times on all architectures. Based on this observation we can still analyze the produced results presented on the following graph and formulate valid conclusions about the general performance.

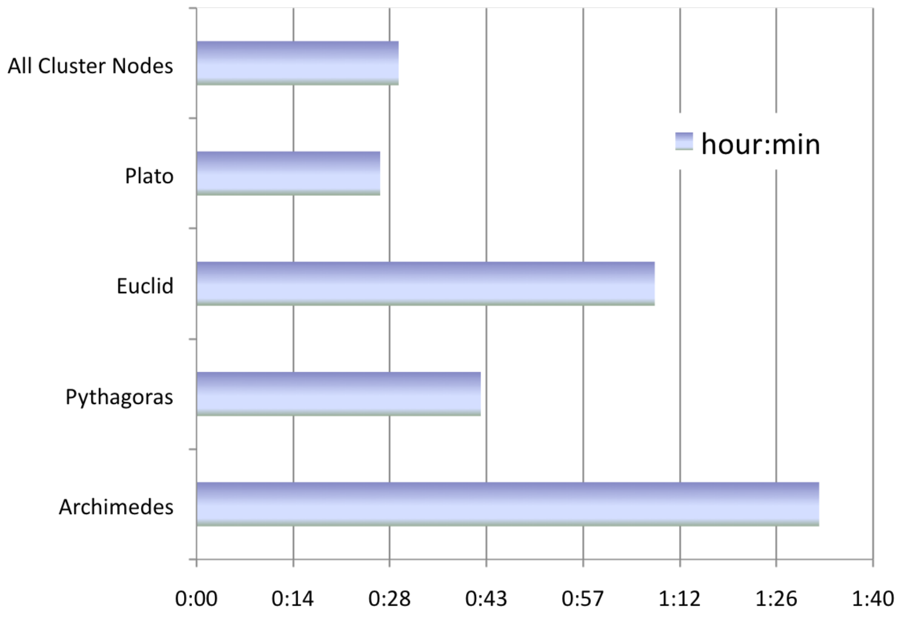

Figure 18: Cluster vs Units (10.200 words)

Archimedes reports almost *1,4 slower* processing than Euclid and Plato is proved approximately *1,6 times faster* than Pythagoras. The parallelized processing on the cluster nodes reveals *2,2 times* faster processing than Euclid and the nodes are able to successfully compete with Plato and Pythagoras. In particular, they are proved slightly slower than Plato and *1,3 times faster* than Pythagoras. This graph shows clearly the benefits that arise by using latest generation of processor units and proves that the CPU utilization does not introduce any imbalances, similar to the one that was identified in the case of image files.

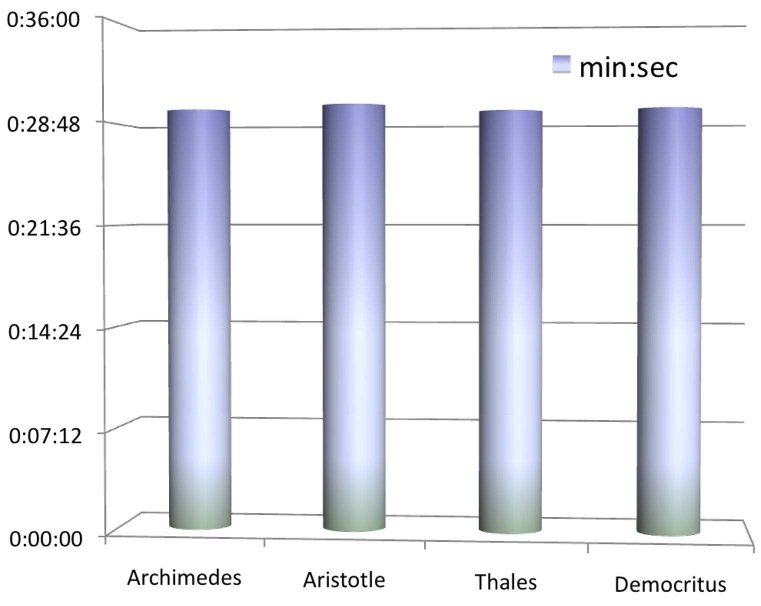

Figure 19: Cluster Nodes ( 4 x 2.550 words)

In an attempt to clearly show that the division of workload (4 x 2.550) represents a simple and an effective way to parallelize word hashing, we present an additional graph that demonstrates the overall processing time for each of the cluster nodes. More precisely, in figure 19 we present the performance for each of the cluster nodes, namely

*Archimedes*, *Aristotle*, *Thales* and *Democritus*. Again, the reported overall processing times are incorrect and they should be in much lower levels than reported. The latency which was introduced by our inefficient bash script must be excluded, in order to obtain the real processing times. However, this latency is present for all nodes and therefore, we could successfully argue that identical boards will offer almost the same processing times. In practice, in a *large-scale* deployment of our cluster model the *slowest* reported time will define the total time required for completing the hashing of the entire scheduled workload. This of course assumes that each scheduled job and the divided workload are *independent* and the scheduled processes are not related. In any other case, on which the cluster nodes have processes that introduce *dependences* among the nodes, the task of defining the total required time depends on the nature of relations that have been introduced.

## 6.3 Performance & Diskless Architecture

In general, the diskless cluster offered an interesting performance through parallelization and proved that better processing times are possible with the usage of nodes which lack a local hard disk. The advantage that comes from the additional processing units is promising, however the way on which processors will be used could accordingly change the level of performance. By comparing the diskless approach to a *regular cluster* architecture that maintains local hard disks, we identify several existing *trade-offs* with respect to *cost*, *performance*, *configuration flexibility* and *storage capacity*. The diskless cluster architecture constitutes an alternative option for building a cluster architecture with an easy and straightforward way, however it comes with its own limitations. The introduced *network latency* from the remote accessing on the hard disk is definitely present and the *intelligence* of the implementations for reading and transferring data *from* and *to* the nodes is a fundamental factor. In our opinion, as in every cluster system, the performance benefits are closely related to the *available resources*, the *nature* of the performed tasks and the *capabilities* of the used hardware. The experiments that we conducted on the different hashing commands yielded a promising performance, but also revealed issues that we have not initially considered, such as the *remote transferring/buffering* of the data, the *balance* of CPU utilization and the *delays* that can be caused by scripting languages. Our two test cases gave us the ability to evaluate the architecture under different processing workloads and allowed us to answer a number of research questions, but also reveal additional research paths that should be further investigated. The following sections present some of our conclusions with respect to the early questions we have formulated in section 3.6 and analyzes the several involved aspects that appear on the cluster model, when it is used for hashing operations.

### 6.3.1 Remote Processing vs Local Processing (Q. 1)

By recording the performance that Archimedes and Euclid reported for our two test cases, we identify the existence of an obvious delay from the remote processing. This was of course expected, but we could not define the precise performance for the different hashing operations without additional testing. Our conducted experiments demonstrated that, depending on the data size and the intensiveness of the performed task, the induced latency changes form. In the case of image files, the transferred data are quite large, but on the other hand the *read* and *write* calls to the remote hard disk are maintained in a scale that does not affect the overall processing times. Moreover, the amount of installed memory seems to influence the processing of image files and there were points on which Archimedes yielded slightly better performance than Euclid. We are not quite sure if memory resources are the only factor that led to these results. The CPU utilization which is controlled by the OS, but also the fact that the better performance appeared mainly on the hashing algorithms that are considered more intensive, introduce new factors for investigation and further testing is required. On the other hand, for the case of the dictionary file the performance is affected for different reasons. The file size which is transmitted through the network is negligible for any serious delays, however the performed *read* and *write* calls are proved very expensive. Characteristic is the example of the *output redirection into files*, which was excluded from our bash scripts because of the introduced latency. In an overall assessment we could say that remote processing is not as efficient as local and someone needs to explore in depth the involved components. In our case, a suggestion would be to analyze the functionality that the *NFS protocol* offers and identify any optimizations that could be made with respect to the remote *read* and *write* calls to the shared hard disk. If someone could balance effectively the access calls to the remote hard disk, then it is believed that this will probably increase performance, in tasks such as the hashing of words.

### 6.3.2 Network Latency (Q. 2)

By examining strictly the performance of the *network interfaces* and their transfer rates, we are not able to identify any major performance issues. From the two conducted benchmarks only the one of image files could constitute a possible bottleneck for the network, but the fact that we used *direct one-to-one* connections gave us a serious advantage. In larger scale deployments, where network devices will be introduced in between in order to increase the number of cluster nodes, the performance constraints for large data sizes will become more obvious. Assuming that the access calls for the case of image files do not introduce a great performance impact and the fact that Archimedes already reported a noticeable delay for hashing remotely the entire database of image files, it makes us believe that network connection will be one of the most important factor for further scaling. In the case of hashing image files, splitting each file in parts in order to parallelize the processing in an optimal way, will become an important *prerequisite* for balancing network latencies. However, hashing operations have a *statefull* nature and therefore, advanced programming methods are needed and a proper synchronization during processing. In the case of the dictionary file the size of a single word does not cause major delays. But in a case on which brute force attacks are applied by using a database of dictionary files which include millions of words to be hashed into the different cluster nodes, a performance question with respect to the behavior of the network interface arises. We are not in a position to make any specific conclusions about this case, since we did not perform any relevant tests. However, we are quite sure that the access calls to the remote hard disk would be much more expensive from the reported network latency.

### 6.3.3 Hashing Implementations (Q. 3)

The open source hashing implementations which we used in our tests did not reveal any surprises with respect to performance. Except from *one* or *two* individual cases on which we had an unexpected behavior due to third factors, such as the CPU utilization and the implementation language, all hashing implementations performed as expected. The family of *SHA-2* algorithms together with the implementations of *Whirlpool* and *Skein* algorithms, yielded the highest processing times and the most intensive computations. On the other hand, the implementations for *MD5*, *SHA-1* and *Tiger* algorithms reported faster processing times and less intensive processing. More precisely *md5sum* was proved the fastest implementation, when the CPU utilization did not affect the hashing operation. On the other hand, *skein1024sum* was proved the slowest implementation, with *sha512sum* and *sha384sum* to follow. A remark for all the hashing implementations is that in the case of the word hashing the CPU utilization remained 0% and the reported differences among the algorithms on the same system were not significant. Finally, the implementations for the same algorithms, namely *sha1sum*, *sha1deep*, *sha256deep* and *sha256sum* behaved almost the same, enhancing in this way our confidence for the results we obtained from our benchmarks.

The reader at this point could easily wonder why we did not use *eBASH* [25] *(ECRYPT Benchmarking of All Submitted Hashes)* for *measuring* or *comparing* the performance of the involved hashing implementations. In our opinion, eBash is a powerful benchmarking tool for measuring the performance of hash algorithms on different processor units, but our project targeted to record the behavior of specific hash implementations mainly on our cluster system. Someone could argue that we could at least correlate our results with the one that eBash produces, however we believe that this is not a valid approach. The results that we obtained for the performance of the open source hash implementations remain at a quite high level in comparison to the analysis that eBash offers and they are not targeting on *cycles/byte* performance. Any attempt for comparing our results by using eBash as a metric would be probably inappropriate and could lead us on the wrong conclusions. Our system is affected by several third factors that should first be identified and investigated in depth, before we use additional tools to strictly measure the performance of hash algorithms. We consider eBASH as a great *stand-alone* way to benchmark hashing implementations, but as in any valid benchmark the structure and the goals should remain stable, without involving external metrics which were not included from the start. Thus, we will agree that eBASH could constitute an extra approach for testing the open source hashing implementations. However, this should be done under the proper testing and by focusing only on the performance of the hashing implementations, after all the existing performance barriers have been successfully excluded.

### 6.3.4 Performance of Skein Algorithm (Q. 4)

If we exclude the dictionary file test case, the *Skein algorithm* reported performance similar to the one that its authors presented. *Skein1024sum* was proved slower than *sha512sum* and *sha384sum*, but *skein512sum* was proved faster than both of them. On the other hand, *skein256sum* reported slower processing than *sha256sum* and *sha224sum* and verified the numbers reported in [23]. Even though, that the implementation of python library for skein introduces a small interpretation latency, it achieves the expected performance.

### 6.3.5 Better Processors (Q. 5, 7)

The systems that include latest generations of processor units, reported significant processing benefits. Pythagoras and Plato were proved faster than Euclid and demonstrated that processors that have *multi-threading* capabilities

[25]http://bench.cr.yp.to/ebash.html

and more than one *core*, perform quite faster. That allow us to argue that by substituting the processors on each of the remote nodes, our cluster system could obtain an additional advantage with respect to the processing power. At this point we should mention that in order for someone to take the maximum advantage of such processors should design his implementation accordingly. As it was reported at some points in our analysis, skein design seemed to have an advantage on the latest generation of processors, because it was using the processors to a better extent.

### 6.3.6 Processing on the Cluster Nodes (Q. 6, 8)

The *in parallel* processing on the 4 cluster nodes offered an additional power that allowed us to divide our workload accordingly. On both of our test cases the processing times were decreased compared to the processing times that the identical single system offered. The parallelization of the tasks yielded an advantage, which could be further extended by adding more nodes. Although that the form of parallelization that we applied is the simplest and in the case of the image files is not even efficient, we successfully showed that *in parallel* processing on Archimedes, Aristotle, Thales and Democritus, is faster than processing on Euclid. Moreover, the reported results for the cluster nodes demonstrated a performance that could compete with even faster processors. In particular, for the case of image files cluster nodes were proved *faster* than Plato and Pythagoras, while for the case of dictionary file reported a *slower* processing.

At this point reasonably the reader could argue that the performance that the cluster nodes yielded is poor, since someone expects a scaling on the performance of *factor n*, where *n* would be the number of cluster nodes. However, this is not entirely true and this is considered to be the ideal scaling even for cluster architecture that have hard disks on each node. In our case, the remote access calls to the hard disk and in some occasions the network latency from the on the fly processing, lowered indeed the performance of our 4 cluster nodes. Nevertheless, *the way* that the parallelization is applied and the *efficiency* of the implementation to handle in the best possible way the remote access calls, play a significant factor for further improving the performance. Imagine, that even in a regular cluster architecture small differences on the *configuration* of the Operating System that runs on the cluster nodes, could potentially lead to completely different performance numbers.

In our opinion, the *diskless cluster* is not faster than a *regular cluster* system, but this is an expected *trade-off* that comes from the lack of hard disks. However, as a result of this exclusion of storage devices, there is a cost mitigation which could be quite important for large scale deployments. The performance of our diskless model is lower than the *ideal n* factor for scaling. Even though, we could successfully argue that by increasing the number of cluster nodes we could obtain better performance, however we are not in position to speculate about the factor of scaling without further testing. Until now, our experience with the cluster system showed us that there are factors which could influence performance and these factors are not always visible. Therefore, it would be *meaningless* to make a prediction about the factor of scaling that could be obtained by adding more cluster nodes.

### 6.3.7 Scaling the Diskless Cluster Architecture (Q. 9)

In our opinion, scaling the diskless architecture is not a trivial process and it requires an architectural plan that will take into account all the involved factors. This document identified several different considerations that must be made during scaling and it presented that except from the obvious *network latency* there are many other important issues that should not be neglected. If we assume that there are already network cards that offer transfer rates equal to *10 Gbps* and that can be installed on the main node, the inclusion of extra cluster nodes without a great impact on network connection become feasible. However, the concept of network performance escapes the strict bounds of data transmission and it is related to a number of different aspects, such as the *nature of the processing task*, *the access calls to the remote hard disk*, *the installed memory resources*, the *control* over the CPU utilization and the way that the used implementations have been *designed* to function under different processor units.

In general, we can say that scaling the diskless model is possible, but it should be done in a very careful and structured way. Different dependencies are introduced based on the processing goals that must be achieved and that changes the form of optimizations. The two distinct test cases that we benchmarked presented individual bottlenecks on performance and showed that the weight for optimizations must be given in different areas. In particular, the dictionary file reported a need to minimize the *expensive access calls* to the remote hard disk, while the image files yielded mainly a need to *control* the CPU utilization and maintain *adequate* bandwidth and memory resources, for transferring the data through the network.

### 6.3.8 Optimizations (Q. 10)

We believe that there are *two main directions* for optimizations. The first one is quite straightforward and refers to the upgrade of all the installed hardware. If we assume that cost is not an issue, then someone could *substitute* the processor units with latest generation of processors, could *change* or *add* memory resources in order to provide faster access on the data, could *replace* the shared hard disk with one that offers the highest transfer rates and the faster access and could provide network cards with a transfer rate of *10 Gbps.* However, this is an extreme ideal case and it is considered *non-realistic*, because usually the cost constitutes an important priority. Under this *requirement* the second direction for optimization would be to use the *best possible hardware resources* and then focus on optimizations at the *software level.* As we already, demonstrated through our research there are several different areas on which someone could optimize the software. The way that the used implementations have been written, but also the protocol which is used to transfer the data through the network, are considered the most valid points for starting optimization efforts. Of course, someone should not forget that optimizations usually are based on the nature of targeted tasks and thus, there is not a standard procedure for approaching software optimizations. Nevertheless, we think that the presented cluster model has several potentials to be further optimized and it is a promising alternative.

# 7 Conclusions

In this document we presented an alternative system for obtaining additional processing power, in the form of *parallelization*. We described how it is possible to build a diskless cluster and divide the processing for hashing operations into all the available nodes. More precisely, we tested a number of open source hashing implementations and focused mainly on two areas on which hashes appear to be computational expensive, namely on the fields of *cryptanalysis* and *digital forensics*. Although we did not perform testing on either of these two areas, we structured our test cases in such a way, that allowed us to obtain valuable results for the behavior of the diskless cluster with respect to general hashing operations. In particular, we identified several factors that could affect performance on our architecture and we discussed ways for a more optimal use. As an additional comparison metric for the performance of the diskless model, we used *3 supplementary* and *independent* systems that maintained a local hard disk. In this way, we were able to *explore* and *compare* the performance of our cluster architecture from different perspectives. By using the open source hashing implementations we successfully benchmarked a number of well known hash algorithms, including also a new algorithm called *Skein*, which is considered a promising candidate for the SHA-3 standardization. In particular, we reported the performance for different *input data sizes* and we discussed their behavior with respect to the diskless cluster system. Our entire experiment allowed us to identify the main *advantages* and *disadvantages* that appear for executing hashing operations on a diskless cluster model and it gave us the chance to bring to the surface a number of critical points that should be further investigated. Finally, we discussed the emerging technologies of *Hardware Acceleration* and *Graphics Processors Units (GPUs)* and we argued that they could possibly help to further *boost performance* on our cluster model.

After our conducted experiments we concluded that in a diskless cluster model there are *several different factors* that could affect the general performance. Our initial assumption that the *on the fly* processing will introduce a significant network latency was proved valid, however it seems that this is not the only factor that could influence such a system. Depending always on the nature of the performed task and the size of the data to be processed, the factors that could influence performance are transformed accordingly. As it became obvious from our data analysis for the case of the *image files* and for the case of *word hashing*, usually a combination of different issues defines the final achieved performance. Factors such as the *CPU utilization*, the *programming language* on which the application is written, the *protocol* used for transferring the data through the network, the *memory resources* installed on each of the nodes and the limitations that arise from the *remote access calls* to the shared hard disk, are only some of the *performance barriers* that we highlighted through this research.

Even so, our system was proved sufficiently fast to give us better processing times, when *in parallel* processing was applied. The *4 cluster nodes* were powerful enough to minimize the required time for performing the scheduled hashing tasks and at some points, they reported a performance that could compete with faster processor units. If we consider also the fact that we did not perform parallelization with the most *optimal* manner, that leave us with significant space for obtaining even *lower processing times*. With respect to the scaling of our model, there is no doubt that any attempt will be strictly related to the limitations which we described during our analysis. In our opinion, the diskless model offers adequate potential for scalability and it is a matter of choice, if someone is willing to accept all the introduced *trade-offs* in order to enjoy additional processing power. As in any other cluster infrastructure the acquisition of the hardware becomes and extra drawback. However, the diskless model could be considered a bit more *flexible*, since it allows to *boot* the operating system *on the fly* and thus, it could use existing network infrastructures, mitigating in this way the cost. In conclusion, we believe that the diskless architecture is not a flawless system, but it provides specific benefits which could be used accordingly. The remote access calls to the shared hard disk is probably the most important performance barrier for the cluster nodes and if someone wants to achieve the maximum speed for processing, it should be successfully faced. Our proposal is to conduct further research with respect to this architecture, since only then, the existing limitations could be analyzed in depth. We are quite optimistic that the proper optimizations could lead to a better behavior for the proposed model.

# 8 Future Work

Even though it is not possible to cover every aspect of future work within this section, we give a concrete research direction based on the most important performance factors which we identified during this research. We believe that there are several issues that should be additionally explored for such systems, especially for *off-loading* intensive cryptographic operations. In a high-level description, we propose the following areas for further investigation:

**Software:** The results of our experiment revealed that there is a number of significant optimizations that can be applied at the software level. In our attempt to test the capabilities of our cluster system we used specific open source hashing implementations in two basic test cases. However, we consider that additional implementations should be tested. By benchmarking tools from areas such as *Cryptanalysis* and *Digital Forensics* someone could evaluate the performance of our system at different levels and get also a better evaluation about the capabilities of the architecture. In chapter 6 we mentioned that the *in parallel processing* was not applied in an optimal way. Our suggestion is to develop the appropriate software, which will be using the *Open Message Passing Interface (OpenMPI)* library and thus, perform parallelization at all available nodes in the most efficient way. Moreover, it is considered of fundamental importance to try to stress CPU at maximum for all the performed benchmarks, since this is the only way to reach the highest possible performance. Finally, we believe that future experiments should test applications that have *embedded shims* for *Hardware Acceleration* and low-level support for *Graphics Processors Units (GPUs)*, taking in this way advantage of the extra processing power.

**Hardware:** With respect to hardware optimizations we recommend the substitution of *Pentium M processors* either with the latest generation of *i7 family of processors* or with processors that have *embedded hardware acceleration*, such as the Intel EP80579 processor. Concerning the peripheral hardware, such as the *type* and *amount* of memory, the *size* and the *speed* of the shared hard disk and the *transfer rate* that the installed network cards support, we propose a complete upgrade. In practice, we advise the future researchers to increase the available resources for the cluster as much as possible and therefore, obtain a better idea about the influence that this upgrade could have on the entire performance of the system. Finally, during this document we discussed a possible *combination* of hardware acceleration with GPUs and a potential *mix of hardware* with various capabilities at all cluster nodes. Most of the time, real cluster systems use a variety of different hardware and it would be wise to include tests, which will be performed under a *mixed environment.* An important remark is that the researcher should have always in mind that the *main node* of the cluster system should remain powerful enough to support any hardware upgrade, otherwise there is a great chance to introduce a general performance bottleneck on the DRBL server.

**Network:** The scaling of the diskeless architecture is based on its ability to support *multiple* cluster nodes at the same ethernet interface of the main node. In order for this to happen, network devices such as switches should be introduced between the cluster nodes and the main node. This is an important aspect of the system and performance that could be achieved by such a setup has great value for scaling. Therefore, we definitely recommend such experiments. These experiments will report the performance of the network, but also they will provide a better overview for the *NFS protocol* in relation with *remote access calls* to the shared hard disk.

**Architecture:** An interesting task for investigation would be the *integration of a DRBL server* to a common network infrastructure. As we briefly discussed at some point in our document, the diskless architecture could *co-exist* inside a classic network infrastructure, if the proper configurations are made. In particular, by setting up a DRBL server that could control all the other systems inside the network, someone could create an infrastructure that has *two natures.* By alternating the *boot option* at the BIOS of each system, it would be possible to maintain a cluster system for intensive processing tasks. The existence of hard disks at each of the independent systems of a common network infrastructure, add a very promising way to *mitigate* the remote access calls to the remote shared hard disk, since the operating system that would be booted on the fly, will be also aware of the local storage device. Even if this case is *extreme* and *not very common*, we propose at least an attempt for implementing such an approach, as it will help to investigate the performance and the general behavior of this setup.

**Hashing Operations:** Concerning our attempt to benchmark a number of hashing implementations, we certainly encourage further research. In our experiments we remained at a *high-level investigation* and we did not examine thoroughly the design of the tested hashing implementations. Therefore, we recommend the usage of additional tools such as *eBASH* for measuring the performance of different hashing algorithms. Especially for the *Skein* algorithm, we suggest a benchmark that will be conducted with a *non-interpreted* programming language.

## References

[1] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Handbook of Applied Cryptography*, Chapter 9, Pages 321-383, ISBN: 978-0849385230, CRC Press, 1st Edition, (December 1996).

[2] Iya Mironov, Hash Fuctions: Theory, attacks, and applications, Microsoft Research, Silicon Valley Campus, (November 2005).
`http://research.microsoft.com/pubs/64588/hash_survey.pdf` (Accessed on: June 2010)

[3] Arjen K. Lenstra, Eric R. Verheul, *Selecting cryptographic key sizes*, Journal of Cryptology, Vol. 14(4), Pages 255-293, Springer Berlin / Heidelberg, (December 2001).

[4] Ronald L. Rivest, *RFC 3174 - The MD5 Message-Digest Algorithm*, MIT Laboratory for Computer Science and RSA Data Security,(April 1992).

[5] X.Wang, D. Feng, X. Lai, H. Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint, Report 2004/199, (August 2004).

[6] A. Lenstra, B. de Weger, *On the Possibility of Constructing Meaningful Hash Collisions for Public Keys*, Information Security and Privacy, LNCS, Vol. 3574/2005, Pages 267-279, Springer Berlin / Heidelberg, (July 2005).

[7] M. Stevens, A. Lenstra and B. de Weger, *Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities*, Advances in Cryptology - EUROCRYPT 2007, LNCS, Vol. 4515/2007, Pages 1-22, Springer Berlin, (June 2007).

[8] Yu Sasaki, Kazumaro Aoki, *Finding Preimages in Full MD5 Faster Than Exhaustive Search*, Advances in Cryptology - EUROCRYPT 2009, LNCS, Vol. 5479/2009, Pages 134-152, Springer Berlin, (April 2009).

[9] Marc Bevand, *MD5 Chosen-Prefix Collisions on GPUs*, Black Hat Conference 2009, Las Vegas, Nevada, (July 2009).
`http://www.blackhat.com/presentations/bh-usa-09/BEVAND/BHUSA09-Bevand-MD5-PAPER.pdf` (Accessed on: June 2010)

[10] P. Jones, *RFC 3174 - US Secure Hash Algorithm 1 (SHA1)*, (September 2001).

[11] National Security Agency (NSA), *Secure Hash Standard - FIPS 180*, National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS), (May 1993).

[12] National Security Agency (NSA), *Secure Hash Standard - FIPS 180-1*, National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS), (April 1995).

[13] National Security Agency (NSA), *Secure Hash Standard - FIPS 180-2*, National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS), (August 2002).

[14] E. Biham, R. Chen, *Near-Collisions of SHA-0*, Advances in Cryptology - CRYPTO 2004, LNCS, Vol. 3152/2004, Pages 199-214, Springer Berlin / Heidelberg, (December 2004).

[15] X. Wang, H. Yu, Y.L. Yin, *Efficient Collision Search Attacks on SHA-0*, Advances in Cryptology - CRYPTO 2005, LNCS, Vol. 3621/2005, Pages 1-16, Springer, (August 2005).

[16] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, W. Jalby, *Collisions of SHA-0 and Reduced SHA-1*, Advances in Cryptology - EUROCRYPT 2005, LNCS, Vol. 3494/2005, Pages 36-57, Springer Berlin / Heidelberg, (May 2005).

[17] V. Rijmen, E. Oswald, *Update on SHA-1*, Topics in Cryptology - CT-RSA 2005, LNCS, Vol. 3376/2005, Pages 58-71, Springer Berlin / Heidelberg, (February 2005).

[18] International Organization for Standardization/International Electrotechnical Commission, *ISO/IEC 10118-3:2004*, Information technology - Security techniques - Hash-functions - Part 3: Dedicated hash-functions, (February 2004).

[19] F. Mendel, C. Rechberger, M. Schläffer, Søren S. Thomsen, *The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl*, Fast Software Encryption, LNCS, Vol. 5665/2009, Pages 260-276, Springer Berlin / Heidelberg, (July 2009).

[20] R. Anderson, E. Biham, *Tiger: A fast new hash function*, Fast Software Encryption, LNCS, Vol. 1039/1996, Pages 89-97, Springer Berlin / Heidelberg, (January 1996).

[21] F. Mendel, V. Rijmen, *Cryptanalysis of the Tiger Hash Function*, Advances in Cryptology - ASIACRYPT 2007, LNCS, Vol. 4833/2008, Pages 536-550, Springer Berlin / Heidelberg, (November 2007).

[22] T. Isobe, K. Shibutani, *Preimage Attacks on Reduced Tiger and SHA-2*, Fast Software Encryption, LNCS, Vol. 5665/2009, Pages 139-155, Springer Berlin / Heidelberg, (July 2009).

[23] N. Ferguson, S. Lucks , B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker, *The Skein Hash Function Family*, Version 1.2, (September 2009).

[24] M. Bellare, T. Kohno, S. Lucks, N. Ferguson, B. Schneier, D. Whiting, J. Callas, J. Walker, *Provable Security Support for the Skein Hash Family*, Version 1.0, (April 2009).

[25] J.P. Aumasson, C. Calik, W. Meier, O. Ozen, Raphael C., W. Phan, K. Varici, *Improved Cryptanalysis of Skein*, Advances in Cryptology ASIACRYPT 2009, LNCS, Vol. 5912/2009, Pages 542-559, Springer Berlin / Heidelberg, (December 2009).

[26] T. Dierks, E. Rescorla, *RFC 5246 - The Transport Layer Security (TLS) Protocol*, Version 1.2, (August 2008).

[27] S. Kent, K. Seo, *RFC 4301 - Security Architecture for the Internet Protocol*, BBN Technologies, (December 2005).

[28] L. Zhao, S. Makineni, L. Bhuyan, *Anatomy and Performance of SSL Processing*, Proc. 2005 IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS 2005), Austin TX USA, Pages 197-206, IEEE, (May 2005).

[29] V. Gupta, D. Stebila, S. Fung, *Speeding Up Secure Web Transactions Using Elliptic Curve Cryptography*, 11th Network and Systems Security Symposium, Pages 231–239, (February 2004).

[30] Intel Corporation, *Accelerating a Security Appliance*, Intel EP80579 Integrated Processor with Intel QuickAssist Technology, (February 2009).
`download.intel.com/design/intarch/ep80579/320124.pdf` (Accessed on: June 2010)

[31] Intel Corporation, *Intel EP80579 Integrated Processor Product Line*, Platform Design Guide, (May 2010).
`http://download.intel.com/design/intarch/ep80579/320068.pdf` (Accessed on: June 2010)

[32] Intel Corporation, *Intel EP80579 Software for Security Applications on Intel QuickAssist Technology*, Cryptographic API Reference, (May 2009).
`http://download.intel.com/design/intarch/ep80579/320184.pdf` (Accessed on: May 2010)

[33] Vassil Roussev, Yixin Chen, Timothy Bourg, Golden G. Richard III, *md5bloom: Forensic filesystem hashing revisited*, Digital Investigation, Vol. 3, Pages 82-90, (June 2006).

[34] Jesse Kornblum, *Identifying Almost Identical Files Using Context Triggered Piecewise Hashing*, Digital Forensic Research Workshop (DFRWS), Conference 2006, (June 2006).
`http://dfrws.org/2006/proceedings/12-Kornblum.pdf` (Accessed on: April 2010)

[35] M. E. Hellman, *A Cryptanalytic Time - Memory Trade-Off*, Information Theory 26, Pages 525-530, IEEE Transactions, (July 1980).

[36] Philippe Oechslin, *Making a Faster Cryptanalytic Time-Memory Trade-Off*, Advances in Cryptology - CRYPTO 2003, LNCS, Vol. 2729/2003, Pages 617-630, Springer Berlin / Heidelberg, (October 2003).

[37] Wi-Fi Alliance, *Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks*, (April 2003).

[38] B. Kaliski, *RFC 2898 - PKCS#5: Password-Based Cryptography Specification*, RSA Laboratories, (September 2000).

[39] Alexander Krenhuber, Andreas Niederschick, *Advanced Security Issues in Wireless Networks*, Johannes Kepler University of Linz.
`http://www.fim.uni-linz.ac.at/Lva/SE_Netzwerke_und_Sicherheit_Security_Considerations_in_Intercon_Networks/semD.pdf` (Accessed on: April 2010).

[40] K. Jarvinen, M. Tommiska, J. Skytta, *Comparative survey of high-performance cryptographic algorithm implementations on FPGAs*, Information Security, IEEE Proceedings , Vol. 152, Pages 3-12, IEEE, (October 2005).

[41] Johnny Cache, Vincent Liu, *Hacking Exposed Wireless: Wireless Security Secrets & Solutions*, Chapter 3, Pages 84-88, ISBN: 978-0072262582, McGraw-Hill Osborne Media, (March 2007).

[42] Matthew Gast, *802.11 Wireless Networks: The Definitive Guide*, Chapter 7, Pages 160-167, ISBN: 978-0-596-00183-4, O'Reilly Media, ( April 2002).

[43] Chao-Tung Yang, Wen-Feng Hsieh, Hung-Yen Chen, *Implementation of a Diskless Cluster Computing Environment in a Computer Classroom*, APSCC 2008, Pages 819-824, IEEE Asia-Pacific Services Computing Conference, (December 2008)

# A Appendix: Sample Scripts

## A.1 Benchmarking scripts

```
#!/bin/bash
# ----------------------------------------------------------------------------------
# The following script was used for hashing the entire image database for every
# algorithm.
#
# Author: Spyridon Antakis
#
# Notice: The redirection of the output to the "allhashes" file was excluded from the
#         cluster benchmark, due to the existing functionality of the cluster to
#         automatically save the produced output.
# ----------------------------------------------------------------------------------

function StartHashing {

cd /home/spyros/database/

date >> ../allhashes
for file in `dir -d *` ;
do
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" md5sum "$file" 1>> ../allhashes        \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha1deep "$file" 1>> ../allhashes      \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha1sum "$file" 1>> ../allhashes       \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha256deep "$file" 1>> ../allhashes    \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha256sum "$file" 1>> ../allhashes     \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha224sum "$file" 1>> ../allhashes     \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha384sum "$file" 1>> ../allhashes     \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha512sum "$file" 1>> ../allhashes     \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" tigerdeep "$file" 1>> ../allhashes     \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" whirlpooldeep "$file" 1>> ../allhashes \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein256sum "$file" 1>> ../allhashes   \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein512sum "$file" 1>> ../allhashes   \
2>> ../allhashes
/usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein1024sum "$file" 1>> ../allhashes  \
2>> ../allhashes
done
date >> ../allhashes
}

rm -fr /home/spyros/allhashes
StartHashing
```

```
#!/bin/bash
# ----------------------------------------------------------------------------------
# The following script was used for hashing the entire dictionary file for every
# algorithm.
#
# Author: Spyridon Antakis
#
# Notice: The redirection of the output to the "allhashes" file was excluded from the
#         cluster benchmark, due to the existing functionality of the cluster to
#         automatically save the produced output.
# ----------------------------------------------------------------------------------


function StartHashing {

cd /home/spyros/database/

date >> ../allhashes
for file in `dir -d *` ;
do
while read line
do
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" md5sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha1deep 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha1sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha256deep 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha256sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha224sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha384sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" sha512sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" tigerdeep 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" whirlpooldeep 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein256sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein512sum 1>> ../allhashes \
2>> ../allhashes
echo "$line" | /usr/bin/time -f "User:%U␣Sys:%S␣Time:%E␣CPU:%P" skein1024sum 1>> ../allhashes \
2>> ../allhashes
done < $file
done
date >> ../allhashes
}

rm -fr /home/spyros/allhashes
StartHashing
```

```
#!/bin/bash
# -----------------------------------------------------------------------------------
# The following script was used for hashing the entire image database for a single
# algorithm.The sample code corresponds to the md5sum function. Identical scripts
# were used for the other hashing algorithms.
#
# Author: Spyridon Antakis
#
# Notice: The redirection of the output to the "md5sum" file was excluded from the
#         cluster benchmark, due to the existing functionality of the cluster to
#         automatically save the produced output.
# -----------------------------------------------------------------------------------

function StartHashing {

cd /home/spyros/database/
date >> ../md5sum
for file in `dir -d *` ;
echo "MD5SUM" >> ../md5sum
/usr/bin/time -f "User:␣%U␣Sys:␣%S␣Time:␣%E␣CPU:␣%P" md5sum "$file" 1>> ../md5sum \
2>> ../md5sum
done
date >> ../md5sum
}

rm -fr /home/spyros/md5sum
StartHashing
```

```
#!/bin/bash
# -----------------------------------------------------------------------------------
# The following script was used for hashing the entire dictionary file for a single
# algorithm.The sample code corresponds to the md5sum function. Identical scripts
# were used for the other hashing algorithms.
#
# Author: Spyridon Antakis
#
# Notice: The redirection of the output to the "md5sum" file was excluded from the
#         cluster benchmark, due to the existing functionality of the cluster to
#         automatically save the produced output.
# -----------------------------------------------------------------------------------


function StartHashing
{

cd /home/spyros/database/
date >> ../md5sum
for file in `dir -d *` ;
do
while read line
do
echo "$line" | /usr/bin/time -f "User:␣%U␣Sys:␣%S␣Time:␣%E␣CPU:␣%P" md5sum 1>> \
../md5sum 2>> ../md5sum
done < $file
done
date >> ../md5sum
}

rm -fr /home/spyros/md5sum
StartHashing
```

## A.2 Python script for Skein algorithm

```
#!/usr/local/bin/python3
# ---------------------------------------------------------------------------------
#   skeinsum.py
#   Copyright 2008, 2009 Hagen Frustenau <hagen@zhuliguan.net>
#
#   Demonstrates Skein hashing with PySkein.
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program.  If not, see <http://www.gnu.org/licenses/>.
# ---------------------------------------------------------------------------------


import sys
import os
import skein
from io import DEFAULT_BUFFER_SIZE

# For the cases of skein512sum & skein1024 the following numbers were changed into 512
# and 1024, respectively.

HASH = skein.skein256
DIGEST_BITS = 256


def printsum(f, name):
    h = HASH(digest_bits=DIGEST_BITS)
    buf = True
    while buf:
        try:
            buf = f.read(DEFAULT_BUFFER_SIZE)
        except KeyboardInterrupt:
            print()
            sys.exit(130)
        h.update(buf)
    try:
        print("{0}␣␣{1}".format(h.hexdigest(), name))
    except IOError as e:
        if e.errno != 32:
            raise


if len(sys.argv) < 2:
    printsum(sys.stdin.buffer, "-")
else:
    for filename in sys.argv[1:]:
        if os.path.isdir(filename):
            print("skeinsum:␣{0}:␣is␣a␣directory".format(filename),
                  file=sys.stderr)
            continue
        with open(filename, "rb") as f:
            printsum(f, filename)
```

## A.3 Job Scheduling scripts for Cluster Nodes

```
#!/bin/bash
# --------------------------------------------------------------------------------
# The following scripts were used for scheduling all the scripts for hashing into
# the clustering queue.
#
# They represent just a sample code and variations of these scripts were used for
# scheduling the performed test cases.
#
# Author: Spyridon Antakis
# --------------------------------------------------------------------------------

qsub -l walltime=24:00:00 md5sum
qsub -l walltime=24:00:00 sha256sum
qsub -l walltime=24:00:00 sha256deep
qsub -l walltime=24:00:00 sha224sum
qsub -l walltime=24:00:00 sha512sum
qsub -l walltime=24:00:00 sha384sum
qsub -l walltime=24:00:00 sha1sum
qsub -l walltime=24:00:00 sha1deep
qsub -l walltime=24:00:00 tigerdeep
qsub -l walltime=24:00:00 whirlpooldeep
qsub -l walltime=24:00:00 skein256sum
qsub -l walltime=24:00:00 skein512sum
qsub -l walltime=24:00:00 skein1024sum


# --------------------------------------------------------------------------------
#  The following script was used in order to define the specific cluster node on
#  which each hashing script would be executed.
#
#  This script was used mostly for equally spliting the hashing of the dictionary
#  file into the 4 cluster nodes.
#
#  Author: Spyridon Antakis
#
#  Where: arch_dict_split, arist_dict_split, tha_dict_split, dem_dict_split
#         Bash scripts for hashing the 1/4 (2550 words) of the dictionary file.
# --------------------------------------------------------------------------------

qsub -l nodes=node_archimedes02,walltime=24:00:00 arch_dict_split
qsub -l nodes=node_aristotle02,walltime=24:00:00 arist_dict_split
qsub -l nodes=node_thales02,walltime=24:00:00 tha_dict_split
qsub -l nodes=node_democritus02,walltime=24:00:00 dem_dict_split
```

## B Appendix: Photographs

**Notice:** In the photographs appear *5 or 6 cluster nodes*, but at end only *4* of them were used. The reason was some complications with the network cards, which had been installed on *Pythagoras*.

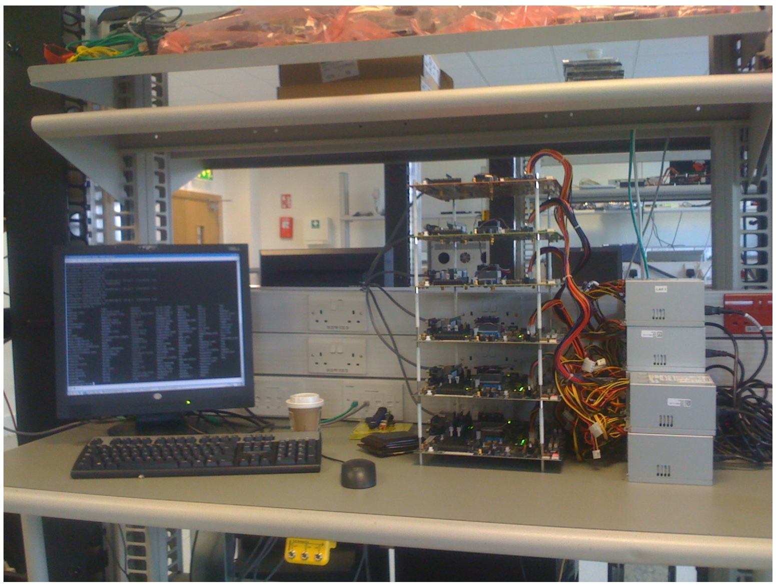

Figure 20: The Diskless Cluster

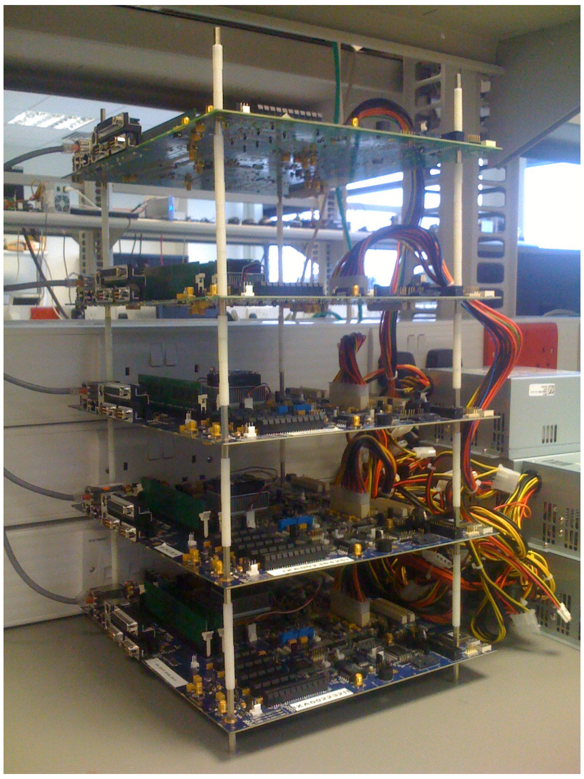

Figure 21: Cluster Nodes