TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

On the Speed of VSH

By
S.J.A. de Hoogh

Supervisors:

Scott Contini (Macquarie University Sydney)
Benne de Weger (TU/e)

Eindhoven, February 21, 2008

Abstract

Some important security schemes, such as Digital Signature Schemes, rely on cryptographic hash functions that are collision resistant. It is known that the most common hash functions, such as SHA1 and MD5, do not satisfy this property. Moreover, as cryptanalysts are advancing the methods for finding collisions for this type of hash functions, one might consider hash functions that are provable collision resistant. In 2005, Contini, Steinfeld and Lenstra introduced the Very Smooth Hash (VSH) that is the first provable collision resistant hash function (based on the hardness to factor large numbers) that is getting close to being practical. One of the problems of VSH is that it has a relatively long running time with respect to the common hash functions.

In this thesis, we show how to improve the speed of VSH by about 75% by means of smart implementations and by means of using a less conservative security assumption. In addition, we will present an implementation of VSH based on trapdoor information that makes VSH three times faster. Lastly, we show that advances in integer factorization are less of a problem for VSH than they are for RSA. Specifically, increasing the size of the modulus results in only a linear slowdown for VSH, as opposed to a cubic (in practice) slowdown for RSA.

Acknowledgements

First of all, I would like to thank those who made it possible to make a dream of mine to come true: doing my final project outside of Europe. In this respect I would like to thank Dr. Benne de Weger for contacting Stephan Overbeek, who I wish to thank for contacting Prof. Jozef Pieprzyk. Then, I wish to thank Prof. Jozef Pieprzyk for accepting me and helping me during my stay. I want to thank the Macquarie University Sydney for helping me doing this project by providing help and sufficient facilities.

Secondly, I would like to thank Dr. Scott Contini for suggesting this nice and interesting topic and for his support and helpful supervision during this research. I would also like to thank Dr. Benne de Weger, Dr. Ron Steinfeld and Prof. Igor Spharlinski for their support and helpful conversations.

And last but not least, I would like to thank those that gave me extra moral and/or financial support: The research institute EIDMA, Prof. Henk van Tilborg, my best friends, my family and of course my fiancee, who spend the whole year with me in Sydney.

Contents

Chapter 1

Introduction

As globalization is a fact, computer networks (such as the internet) are becoming more and more important to establish long distance communication in a reliable way. Since almost anyone can access such networks, security is an important issue. Cryptographic hash functions play a vital role in many such security systems.

As [7] describes, throughout the history people used several different and contradictory definitions for cryptographic hash functions. So it seems to be very difficult to formulate precisely what 'a' cryptographic hash function is and what properties 'it' should satisfy[1]. According to [22] a cryptographic hash function H is ideally a function that roughly satisfies the following properties:

- H maps an input x of arbitrary length to an output $H(x)$ of fixed length.
- It takes relatively little time to compute $H(x)$.
- (*pre-image resistance*) Upon receiving $H(x)$, it is computationally infeasible to find x.
- (*second pre-image resistance*) Upon receiving $H(x)$, it is computationally infeasible to find $x' \neq x$ such that $H(x') = H(x)$.
- (*strong collision resistance*) It is computationally infeasible to find an x and an x', where $x \neq x'$, so that $H(x) = H(x')$.

For a more detailed introduction to hash functions we refer to [22] or [11]

Scott Contini et al. also describe in [7] that the above definition is very imprecise. For example: What means "computationally infeasible" precisely? In addition to that they show that it is hard to formalize these definitions. We will, nevertheless, try to formalize above definition by giving a precise definition of "computationally infeasible". Because of the discussion of both Contini et al. in [7] and Phillip Rogaway in [24], we have to note here, that we don't claim this definition to be a very good one with respect to the general case. But it will give more insight in the definition of a cryptographic hash function and it will suffice with respect to VSH.

With computationally infeasible, we mean the following:

Definition 1.0.1. *Let X be a problem, and let Y be a hash function. Let α denote the security parameter and let $t(\alpha, X, Y)$ denote the number of times that Y is called by the*

[1] see http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html

fastest algorithm known to solve problem X. $t(\alpha, X, Y)$ is also called the workfactor. We say that solving X is computationally infeasible for Y if the corresponding workfactor is super-polynomial in α.

For example: for common hash functions as SHA1, MD5 and HAVAL the security parameter is usually the bit-length of their output, say α. Initially, it is then assumed that the fastest algorithm to find either a pre-image or a second pre-image or to find a collision are brute force methods, i.e. exhaustive search for finding a first or second pre-image and a birthday attack for finding a collision. So it is assumed that the workfactor for these hash functions are respectively 2^α to find a first or second pre-image and $2^{\alpha/2}$ to find a collision. At Eurocrypt 2005 John Kelsey and Bruce Schneier showed in [17] that second pre-images can be found with a workfactor much less than 2^n, but still super-polynomial in n. With respect to VSH, the asymptotic complexity to find collisions is more or less equivalent to the complexity to factor an n-bit number using the Number Field Sieve, i.e. sub-exponential (but super-polynomial) in n.

Some important applications, such as Digital Signature Schemes (for example [1]), rely on hash functions which are strong collision resistant. It is known that the most common hash functions like MD5, HAVAL and SHA1 do not satisfy this property, see [28] and [27]. In [6] Scott Contini, Ron Steinfeld and Arjen Lenstra introduce VSH (*Very Smooth Hash*) a cryptographic hash function that is *provably* collision resistant. With *provably* we mean that it can be proven that finding a collision for VSH is as hard as solving some previously known and well studied mathematical problem which is assumed to be hard. Like RSA, this problem is based on the hardness to factor a large modulus $n = pq$, where p and q are secret primes of almost equal size.

However, this also results in some disadvantages. To provide proper security, n should be large (usually about 1024 bits at least). As the next chapter will show, this results in time-consuming modular arithmetic and a relatively large output length. The output of SHA1 is, for example, 160-bits, while the output of VSH is equal to the bit-size of n. Moreover, VSH provides collision resistance only. For certain outputs, it is easy to find a corresponding input. So in the sense of [22] VSH is not a hash function at all.

Still, VSH provides a provable collision resistant function, which is the most practical in the group of hash functions with provable properties, see also [26]. According to [6], previous hash functions that are also based on modular arithmetic and have provable properties (see for example [10]) require at least one modular multiplication per $O(\log \log n)$ message bits, whereas VSH requires only one modular multiplication per $\Omega(\log n)$ message-bits, without losing its provable collision resistance property. So indeed VSH is very efficient for this type of hash functions. Here, O and Ω denote the usual Big-O, Big-Omega notation, see also section **??** or section 2.4 of [12].

This thesis will describe VSH and how we improve the speed of VSH. We will build VSH and our solutions in the C programming language. The first chapter will give a detailed description of VSH. The second chapter will explore several techniques to perform fast arithmetic. The third chapter will describe some implementation ideas. The fourth chapter will discuss our main idea. The fifth chapter will discuss an idea with high performance, but with restricted applicability. The sixth chapter will discuss the influence of the speed of VSH when 1024-bit RSA moduli are not safe anymore. The final chapter will conclude this thesis.

Chapter 2

The Very Smooth Hash

Common cryptographic hash functions like SHA1, HAVAL and MD5 are usually very fast, but they have no security proofs behind them. However, there are hash functions (see for example [10]) that provide provable security. Unfortunately, these functions are usually very slow compared to the common hash functions. In [6] Scott Contini, Arjen Lenstra and Ron Steinfeld introduce VSH, which is provable collision resistant and –for this type of hash functions– very efficient. The following table illustrates this.

Year	Hash Function defined in	Image (hash) space	# Modular multiplications per message bit
2006:	[4]	$\mathbb{F}_{p^2}$	$2\log(p)$
2001:	[23, 25]	$\mathbb{Z}/n\mathbb{Z}$	1.5
1987:	[10]	$\mathbb{Z}/n\mathbb{Z}$	$O(1/\log\log n)$
2005:	VSH [6]	$\mathbb{Z}/n\mathbb{Z}$	$O(1/\log n)$

Table 2.1: Efficiency of several provable secure hash functions

VSH is based on modulo n arithmetic, where n is an RSA modulus. So $n = pq$, where p and q are distinct primes of almost equal size. Like RSA, the security of VSH is based on the difficulty to factor n. We will show in this chapter that if it is easy to create collisions, i.e. two distinct messages for which VSH produces the same output, then it is easy to factor n. Furthermore, we will show in this chapter how [6] relates the security of VSH to the security of factoring.

Moreover, VSH enjoys the property that in some sense its speed is linear in the bit-size of the modulus n. We will show this in detail in chapter 7. Due to the recent and quite successful attempts to factor large RSA moduli (see [21]) it may be desirable to increase the size of the modulus n to maintain a reasonable level of security. Because the speed of VSH is more or less linear in the size of n, increasing the size of n does not affect VSH's speed as much as it would affect the speed of RSA, which is cubic in the size of n.

Also, Contini et al. show in [6] that VSH can be very useful to speed up provably secure signature schemes (such as Cramer-Shoup) and to build a signature scheme where VSH is combined with RSA.

In order to give a complete introduction to the remainder of this thesis, this chapter will discuss VSH in detail. This discussion is based on [6]. The first section will introduce some important notation and describe the background of the security of VSH. The second section will present VSH and give its security proof. The third section will describe how [6] improves the speed of VSH and the last section will give some possible applications of VSH.

2.1 Background of the Security of VSH

This section will describe the background of the security of VSH and will conclude with a computational assumption. Based on this assumption the collision resistance property of VSH can be proved. But first we will introduce some important notations.

2.1.1 Notation

Let $c > 0$ be a fixed constant, p and q some primes and let $n = pq$ be a composite of bit-size S, which is hard to factor. Let $\phi(n)$ denote the Euler-phi function, i.e. $\phi(n) = (p-1)(q-1)$. Let $\mathbb{Z}_n$ denote the ring of integers modulo n represented by $\{0, 1, \cdots, n-1\}$. Finally let p_i denote the i-th prime, where $p_1 = 2, p_2 = 3, p_3 = 5, \cdots$. Let p_0 be equal to -1.

If for certain real functions f, g it holds that there exist constants $d > 0$ and $X \in \mathbb{R}$ such that $|f(x)| \leq |d \cdot g(x)|$, for all $x \geq X$ then we write $f(x) = O(g(x))$. And if it holds that $|f(x)| \geq |d \cdot g(x)|$ for all $x \geq X$, then we write $f(x) = \Omega(g(x))$.

Finally if $\lim_{x\to\infty} f(x)/g(x) = 1$ for certain real functions f and g, then we write $f(x) \sim g(x)$. If $\lim_{x\to\infty} f(x)/g(x) = 0$, then we write $f(x) = o(g(x))$.

The following definitions define several *smoothness* properties.

Definition 2.1.1. *Let $n = pq$ be an S-bit composite, where p and q are some distinct primes. Let c be some positive real constant. An integer x is called* p_k-smooth *if the largest prime factor of x is less than or equal to p_k.*

The integer x is called very smooth *if the largest prime factor dividing x is less than or equal to $(\log n)^c$.*

Note that by this definition $x = -1$ also counts as very smooth.

Definition 2.1.2. *An integer b is called a* very smooth quadratic residue modulo n *if b is very smooth and if there exists an integer x such that $b \equiv x^2 \mod n$. The integer x is called a* modular square root of b.

The integer x is called a trivial *modular square root if $b = x^2$. In other words: b is a square in $\mathbb{N}$ and x is the integer square root of b.*

2.1.2 Integer Factorization

This subsection will discuss a general approach that is used in most factorization algorithms. Based on this approach the security of RSA is usually evaluated. We will show that this approach is, therefore, also relevant to VSH.

The idea to find factors of n is as follows: find x and $y \in \mathbb{Z}$ such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$. As the following claim will show, it follows that one can easily find factors of n from x and y.

Claim 2.1.3. *Let n be a composite in $\mathbb{N}$, and let $x, y \in \mathbb{Z}$ be integers such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$. Then, $\gcd(x \pm y, n)$ are proper factors of n.*

Proof. Let x, y and n satisfy the properties given in the claim. So for some $k \in \mathbb{Z}$ it holds that $x^2 = y^2 + k \cdot n$, so that

$$x^2 - y^2 = (x - y)(x + y) = k \cdot n. \tag{2.1}$$

Now suppose $\gcd(x+y,n)=1$. Because, by equation 2.1 it follows that $(x-y)(x+y)|n$, it follows by elementary number theory that $(x-y)|n$, i.e. $x \equiv y \mod n$, which is a contradiction. Therefore, $\gcd(x+y,n) \neq 1$ and similarly we find that also $\gcd(x-y,n) \neq 1$.

Hence $\gcd(x \pm y, n)$ are proper factors of n. □

A way of creating x and y that satisfy the properties of claim 2.1.3 is by using *relations* of the form:

$$v^2 \equiv \prod_{i=0}^{u} p_i^{e_i(v)} \mod n, \tag{2.2}$$

where $v \in \mathbb{Z}, e_i(v) \in \mathbb{N} \cup 0$ (for all i, v), and u is some fixed integer. The following claim shows that given $u+1+t$ relations, there exist least t different (x,y)-pairs for which claim 2.1.3 may hold.

Claim 2.1.4. *Let m, k be some positive integers, p some prime and A be an $m \times (m+k)$ matrix with entries in $\mathbb{Z}_p$. We call a vector $x \in \mathbb{Z}_p^{m+k}$ a* dependency *if $Ax \equiv 0 \mod p$. Then A has at least k linear independent dependencies.*

Proof. Let $\ker(A)$ denote the kernel of A. From matrix theory it follows that A has rank at most m and, therefore by the rank-nullity theorem (i.e. $m+k$=rank(A) + dim(ker(A))) the dimension of the kernel is at least k. Observing that $x \in \ker(A)$ implies that x is a dependency, completes the proof. □

So, according [6], given $u+1+t$ different relations of the form (2.2), using linear algebra, one is able to find at least t linear independent dependencies modulo 2 in the matrix $(e_i(v_j))_{ij}$, where $i = 0, \cdots, u$ and $j = 0, \cdots u+t$. Let $(d_j)_{j=0}^{u+t} \in \mathbb{Z}_2$ be such dependency. Then

$$\sum_{j=0}^{u+t} e_i(v_j) d_j \equiv 0 \mod 2, \qquad \text{for all } i = 0, \cdots, u$$

implying that

$$\left(\prod_{j=0}^{u+k} v_j d_j \right)^2 \equiv \prod_{i=0}^{u} p_i^{\sum_{j=0}^{u+t} e_i(v_j)} \mod n, \tag{2.3}$$

where all exponents in the right hand side are even by the choice of $(d_j)_{j=0}^{u+t}$. Obviously, from Equation (2.3) x and y such that $x^2 \equiv y^2 \mod n$ are easily deduced. If $x \not\equiv \pm y \mod n$ then it leads to a proper factor of n.

According to [6] finding a relation, where all exponents in the right hand side are even, is extremely rare, unless n is very small. Moreover, it claims that it can be safely assumed that for each relation of the form (2.2) found, at least one of the exponents $(e_i(v))$ is odd. Also, [6] claims that if the relations are random, then with probability at least $1/2$ the resulting (x,y)-pair satisfies the properties of claim 2.1.3 and, therefore, leads to a proper factor of n.

Remark 2.1.5. The previous discussion shows that relations of the form (2.2) lead to the factorization of n. The opposite, i.e. given a factorization, one can create relations of the form (2.2), is also true. This can roughly be done as follows:

1. Choose an exponent vector $(e_j)_{j=0}^{u}$.

2. Calculate $x = \prod_{j=0}^{u} p_j^{e_j}$.

3. Check whether x is a square modulo p and q, the prime factors of n.

4. If not then start over, else calculate the square roots of x modulo the prime factors of n. Let $\sqrt{x} \equiv a \mod p$ and $\sqrt{x} \equiv b \mod q$.

5. Apply the Chinese remainder theorem to find a relation of the form (2.2).

Because checking whether a number is a square modulo a prime, via Euler's criterion, and taking its square root can be done very efficient (see corollary 5.7.3 and the beginning of chapter 7 of [12]), it follows that one can find relations of the form (2.2) efficiently if the factorization of n is known. We will demonstrate this with the following example.

Example 2.1.6. Suppose $n = 143$, so that $p = 11$ and $q = 13$, and suppose that $u = 2$. We choose $(2, 1, 2)$ as exponent vector, so that $x = 2^2 \cdot 3 \cdot 5^2 = 300$. It follows that $x \equiv 3 \mod 11$ and $x \equiv 1 \mod 13$. As $5^2 \equiv 3 \mod 11$ it follows that $\sqrt{x} \equiv 5 \mod 11$ or $\sqrt{x} \equiv 6 \mod 11$, and $\sqrt{x} \equiv 1 \mod 13$ or $\sqrt{x} \equiv 12 \mod 13$. We take $a = 5$ and $b = 1$. As 11 and 13 are co-prime it follows from the Chinese Remainder Theorem that there exists a solution for $\sqrt{x}$ to

$$\begin{cases} \sqrt{x} \equiv a \mod p \\ \sqrt{x} \equiv b \mod q \end{cases},$$

which is unique modulo n. Using the Extended Euclidean Algorithm, we find that $6 \cdot 11 - 5 \cdot 13 = 1$. So $\sqrt{x} \equiv 5 \cdot (-5) \cdot 13 + 1 \cdot 6 \cdot 11 \mod n$, which is equivalent to $27 \mod n$. Hence,

$$(27)^2 \equiv p_1^2 \cdot p_2 \cdot p_3^2 \mod n.$$

2.1.3 The VSSR assumptions

It follows from the previous subsection that finding relations in the form of (2.2) is closely related to factoring n, i.e. finding such relations is easy if and only if factoring n is easy. As nowadays methods to factor n are still sub-exponential in the size of n, one can perhaps assume that finding relations of the form (2.2) is sub-exponential in the size of n and, therefore, hard. This is roughly the assumption on which VSH is based.

To make this precise we will cite the following definition and assumption from [6].

Definition 2.1.7. ***(VSSR)*** *Let n be the product of two unknown primes of approximately the same size and let $k \leq (\log n)^c$. The Very Smooth number nontrivial modular Square Root (VSSR) problem is as follows: Given n, find $x \in \mathbb{Z}_n^*$ such that $x^2 \equiv \prod_{i=0}^{k} p_i^{e_i} \mod n$ and at least one $e_0, \cdots, e_k$ is odd.*

VSSR Assumption: *The VSSR assumption is that there is no probabilistic polynomial (in $\log n$) time algorithm which solves VSSR with non-negligible probability.*

From this definition and assumption it is not clear how the complexity of factoring n and solving VSSR are related. Contini et al. use in [6] the available factoring algorithms to make this clear. One of the most powerful tools to factor n is the *Number Field Sieve* (see Chapter 6 of [9] for an introduction to factoring algorithms). Let $L(X)$ denote the expected time that the NFS (Number Field Sieve) requires to factor an X-bit number. Consider again the fixed integer u in equation (2.2) and recall that n has bit-length S. According to [6] with

the current state of the art, asymptotically, a relation of the form (2.2) cannot be found with the available methods in less than $L(S)/u$ time. Now [6] concludes that with the current state of the art, the security of a system that is based on VSSR using an S-bit modulus is at least as difficult as factoring an S'-bit modulus, where S' is the smallest integer such that

$$L(S') \geq \frac{L(S)}{u} \tag{2.4}$$

This is the so called **Computational VSSR Assumption**.

Remark 2.1.8. As [6] remarks, this computational VSSR assumption is conservative and leads to relatively large n. In chapter 5 we will show how we can improve the computational VSSR assumption so that $S = S'$.

2.2 Classic VSH

This section will describe the basic version of VSH and give its security proof, i.e. a proof that VSH is collision resistant under the VSSR assumption. We will refer to this version of VSH as *Classic VSH*. Finally, we will discuss some advantages and disadvantages of Classic VSH.

2.2.1 Description and the Proof of Security

Let k denote the block length. Choose k as the largest integer such that $\prod_{i=1}^{k} p_i < n$. Let m be a ℓ-bit message and let m_i denote the i-th bit of m. Suppose $\ell < 2^k$. Classic VSH is given by the following algorithm:

Algorithm 2.2.1. Classic VSH

Input: An ℓ-bit message m and an S-bit RSA modulus n.

Output: An S-bit hash of m.

Procedure:

Initialization:

(Initial Value): Let $x_0 = 1$.
(Number of iterations): Let $\mathcal{L} = \lceil \frac{\ell}{k} \rceil$.
(padding): Let $m_j = 0$ for $\ell < j \leq \mathcal{L}k$.
(Appending length block): Let ℓ_j be bits so that $\ell = \sum_{j=1}^{k} \ell_j 2^{j-1}$ and let $m_{\mathcal{L}k+j} = \ell_j$ for $j = 1, \cdots, k$.

Iteration:

For $j = 0, \cdots, \mathcal{L}$ compute

$$x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_i^{m_{j \cdot k+i}} \mod n.$$

Finalization:

Return $x_{\mathcal{L}+1}$.

Theorem 2.2.2. *Classic VSH is collision resistant under the VSSR assumption. In other words: Finding a collision for algorithm 2.2.1 leads to a solution to an instance of VSSR.*

Proof. The proof follows the lines of the proof given in [6]. The proof shows that if one can find colliding messages m and m', then one can use them to find a solution for VSSR. Let x_j and x'_j denote the j-th iterated values of algorithm 2.2.1 applied to m and m' respectively. Let $\ell, \mathcal{L}$ and $\ell', \mathcal{L}'$ denote the bit-length and the number of blocks of respectively m and m'. As m and m' collide it holds that $m \neq m'$ and $x_{\mathcal{L}+1} = x_{\mathcal{L}'+1}$.

Firstly, consider the case in which $\ell = \ell'$. Let $m[j]$ denote the j-th k-bit block of m, $m[j] = (m_{j\cdot k+i})_{i=1}^{k}$, for $j = 0, \cdots, \mathcal{L}$, and we define $m[\mathcal{L}+1] = \emptyset$. Let $t \leq \mathcal{L}$ be the largest index such that $(x_t, m[t]) \neq (x'_t, m'[t])$, but $(x_j, m[j]) = (x'_j, m'[j])$ for $t < j \leq \mathcal{L}+1$. From the choice of t it holds that

$$(x_t)^2 \times \prod_{i=1}^{k} p_i^{m_{t\cdot k+i}} \equiv (x'_t)^2 \times \prod_{i=1}^{k} p_i^{m'_{t\cdot k+i}} \mod n. \tag{2.5}$$

Let $\Delta = \left\{ i \in \{1, \cdots, k\} : m_{t\cdot k+i} \neq m'_{t\cdot k+i} \right\}$ and
$\Delta_{rs} = \left\{ i \in \{1, \cdots, k\} : m_{t\cdot k+i} = r \text{ and } m'_{t\cdot k+i} = s \right\}$. As the primes p_i are coprime to n for all $i = 1, \cdots, k$ it holds that all primes p_i are invertible modulo n, and by the iteration step of algorithm 2.2.1 it holds also that $(x_t)^2$ and $(x'_t)^2$ are invertible modulo n. So equation (2.5) is equivalent to:

$$\begin{aligned}
(x_t)^2 \times \prod_{i \in \Delta_{10} \cup \Delta_{11}} p_i &\equiv (x'_t)^2 \times \prod_{i \in \Delta_{01} \cup \Delta_{11}} p_i \mod n && \Leftrightarrow \\
(x_t)^2 \times \prod_{i \in \Delta_{10}} p_i &\equiv (x'_t)^2 \times \prod_{i \in \Delta_{01}} p_i \mod n && \Leftrightarrow \\
(x_t)^2 \times \left(\prod_{i \in \Delta_{10}} p_i \right)^2 &\equiv (x'_t)^2 \times \prod_{i \in \Delta_{01} \cup \Delta_{10}} p_i \mod n && \Leftrightarrow \\
\left[\frac{x_t}{x'_t} \times \prod_{i \in \Delta_{10}} p_i \right]^2 &\equiv \prod_{i \in \Delta} p_i \mod n && (2.6)
\end{aligned}$$

Obviously if $\Delta \neq \emptyset$ then equation (2.6) solves VSSR as stated in Definition 2.1.7. If $\Delta = \emptyset$ then also $\Delta_{10} = \emptyset$ and it follows from equation (2.6) that $(x_t)^2 \equiv (x'_t)^2 \mod n$. Recall that $x_0 = x'_0 = 1$. As $\Delta = \emptyset$ implies that $m[t] = m'[t]$ it follows from the choice of t and $m \neq m'$ that $t \geq 1$. If $x_t \not\equiv \pm x'_t \mod n$ then it follows from remark 2.1.5 that VSSR can be solved by factoring n. If $x_t \equiv \pm x'_t \mod n$, then $x_t \equiv -x'_t \mod n$, since $m[t] = m'[t]$ implies by the choice of t that $x_t \neq x'_t$. It follows that

$$\begin{aligned}
x_t &\equiv -x'_t \mod n && \Leftrightarrow \\
(x_{t-1})^2 \times \prod_{i=1}^{k} p_i^{m_{(t-1)\cdot k+i}} &\equiv -(x'_{t-1})^2 \times \prod_{i=1}^{k} p_i^{m'_{(t-1)\cdot k+i}} \mod n && \Leftrightarrow \\
\left[\frac{x_t}{x'_t} \right]^2 &\equiv -1 \times \prod_{i=1}^{k} p_i^{m_{(t-1)\cdot k+i} - m'_{(t-1)\cdot k+i}} \mod n,
\end{aligned}$$

a solution to VSSR as $p_0 = -1$ has an odd degree. Thus for the case $l = l'$ it holds that the colliding messages m and m' can be used to find a solution for VSSR.

Next, consider colliding messages m and m', where $l \neq l'$. Since $l \neq l'$, it holds for at least one $i \in \{1, \cdots, k\}$ that $l_i \neq l'_i$. Similar to equation (2.5), we find from $x_{\mathcal{L}+1} = x'_{\mathcal{L}'+1}$ and the fact $m_{\mathcal{L}\cdot k+i} = l_i$ (see initialization part of algorithm 2.2.1) that

$$(x_{\mathcal{L}})^2 \times \prod_{i=1}^{k} p_i^{l_i} \equiv (x'_{\mathcal{L}'})^2 \times \prod_{i=1}^{k} p_i^{l'_i} \mod n. \tag{2.7}$$

Now letting $\Delta_l := \{i \in \{1, \cdots, k\} : l_i \neq l'_i\}$ and $(\Delta_l)_{rs} := \{i \in \{0, \cdots, k\} : l_i = r \text{ and } l'_i = s\}$, we derive similar to equation (2.6) an immediate solution to VSSR, since $\Delta_l \neq \emptyset$:

$$\left[\frac{x_{\mathcal{L}}}{x_{\mathcal{L}'}} \times \prod_{i \in (\Delta_l)_{10}} p_i \right]^2 \equiv \prod_{i \in \Delta_l} p_i \mod n, \tag{2.8}$$

which concludes the proof. □

If the factorization of n is known then collisions are easily generated. Using the fact that $p_i^{a+b\cdot\phi(n)} \equiv p_i^a \mod n$ we can create collisions as follows. Take a message m of bit-size at least $k \cdot S$. Define $e_i = \sum_{j=0}^{\mathcal{L}} m_{j\cdot k+i} 2^{\mathcal{L}-j}$. Observe that VSH will output $x_{\mathcal{L}+1} \equiv \prod_{i=1}^{k} p_i^{e_i} \mod n$. Calculate a message m' such that the corresponding exponents satisfy $e'_i = e_i + t_i\phi(n)$, where t_i are some integers, for all $i \in \{1, \cdots, k\}$. Then m and m' will collide.

The output $x_{\mathcal{L}+1}$ has in general bit-size S, which is more than 1024 bits to provide the same security level as factoring an 1024 bit modulus, according to the computational VSSR assumption. With respect to common hash functions this output is very long. SHA1 outputs hashes of length 160-bits for example.

Finally, modular arithmetic using a large modulus is very time consuming compared to bitwise operations on which the common hash functions are based. To illustrate this, we note that this version of VSH, using $S' = 1024$ in (2.4), is about 75 times slower than SHA1. Subsection 2.3.3 will give an improved version of VSH, which is still 25 times slower than SHA1. We will show in this thesis how we can make a version, which is 15 times slower than SHA1 and, if one has access to the trapdoor information, a version which is only 2.5 times slower than Sha1.

However, as we noted in the introduction, VSH is -compared to other hash functions with provable properties- very efficient. The fastest hash function based on modular arithmetic, with provable properties (see [10]) requires a multiplication per $O(\log \log n)$ message bits. The next subsection will show that Classic VSH requires a multiplication per $\Omega(\log n / \log \log n)$ message bits.

2.2.2 The Efficiency of Classic VSH

This subsection will show that Classic VSH requires a multiplication modulo n per $\Omega(\log n / \log \log n)$ message bits. The next chapter will give a proper introduction to multiplication algorithms, here we only note that a multiplication of two integers A and B can be computed in $O(\log(A) \cdot \log(B))$ time. Consider implementation 2.2.3 of the iteration function of Classic VSH, given as C-code. Because a multiplication of two numbers that are reduced modulo n can be done in $O((\log n^2))$ time, we will say that some calculation is equivalent to a multiplication if it can be done in $O((\log n)^2)$ time. This straightforward implementation consists of three parts: PoP, Sq and Mul. Because x_i is at most as large as

Pseudo-code 2.2.3. Iteration of Classic VSH (straighforward)

Input: The iteration number j and x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$.
Product of Primes (PoP):
for$(i = 1; i \leq k; i{+}{+})\{$
if$(m_{j \cdot k+i} == 1)\{$
$t = t * p_i;$
$\}$
$\}$
Squaring (Sq):
$x_j = x_j^2 \mod n;$
Multiplication (Mul):
$x_{j+1} = x_j * t \mod n;$
Return x_{j+1}.

n it follows that the calculation of x_i^2 can be done in $O((\log n)^2)$ time. And also from the definition of k in algorithm 2.2.1 it follows that t is at most as large as n. Hence, also the calculation of $x_i * t$ can be done in $O((\log n)^2)$ time.

We will show that PoP is equivalent to a multiplication as well. In order to do this we need some basics from number theory. See appendix B for a summary of these basics. Recall from algorithm 2.2.1 that k is chosen such that

$$\prod_{i=1}^{k} p_i < n < \prod_{i=1}^{k+1} p_i. \tag{2.9}$$

Claim 2.2.4. *Given an integer n, let k be an integer such that Equation (2.9) holds. Then,*

$$k \sim \frac{\log n}{\log \log n}$$

Proof. Taking logarithms in Equation (2.9) yields

$$\sum_{i=1}^{k} \log(p_i) < \log n < \sum_{i=1}^{k+1} \log(p_i).$$

This is equivalent to

$$\theta(p_k) < \log n < \theta(p_{k+1}).$$

, where θ is the Chebyshev θ function as defined in Appendix B.1.2. Then, applying Claim B.1.6 yields

$$(1 + o(1))k \log k < \log n < (1 + o(1))(k + 1) \log(k + 1),$$

which is equivalent to

$$1 < \frac{\log n}{p_k} < \frac{(1 + o(1))(k + 1) \log(k + 1)}{(1 + o(1))k \log k}.$$

Letting $k \to \infty$, then, from the squeeze theorem it follows that $p_k \sim \log n$.

Finally it follows that $k = \pi(p_k)$, where π denotes the prime-counting function as defined in Appendix B.1.1. Thus, by the Prime Number Theorem (B.1.3) we conclude

$$k = \pi(p_k) \sim \pi(\log n) \sim \frac{\log n}{\log \log n}.$$

□

As we see in the proof it holds that $p_k \sim \log n$. This implies that $p_k = O(\log n)$. Since $p_i \leq p_k$ for all $i \leq k$ and $t < n$, it holds that PoP can be computed in $O(k \cdot \log(n) \cdot \log(\log(n)))$ time, since the size of each p_i is $O(\log \log n)$. Then, it follows from claim 2.2.4 that PoP can be computed in $O((\log n)^2)$ time. Hence also PoP is equivalent to a multiplication.

Note that not every message bit will be –in general– equal to one. So Classic VSH, based on implementation 2.2.3 requires *at most* 3 multiplications modulo n per iteration. And since each iteration processes exactly k message bits it follows from claim 2.2.4 that Classic VSH requires 3 multiplications modulo n per $\Omega(\log n / \log \log n)$ message bits, which is equivalent to a multiplication modulo n per $\Omega(\log n / \log \log n)$ message bits.

2.3 Improving the Speed of Classic VSH

This section will describe how [6] improves the efficiency of Classic VSH. This section will conclude with Fast-VSH, the variant of VSH, which requires a multiplication per $\Omega(\log n)$ message bits as opposed Classic-VSH, which requires a multiplication per $\Omega(\log n / \log \log n)$ message bits.

2.3.1 Increasing the number of small primes

Because the iteration part of Classic VSH is the time-consuming part, a speed up may be expected when the iteration numbers are reduced. One way of reducing the iteration numbers is by increasing the size of k, i.e. a k larger than implied by Equation (2.9), but still smaller than $(\log n)^c$ to ensure the applicability of the VSSR assumption 2.1.7. A larger k results in more message-bits being processed each iteration and thus less iterations being required. Some drawbacks of increasing k are, for example, that (1) by the computational VSSR assumption (2.4) a larger modulus is required and (2) that PoP in 2.2.3 will be more time-consuming, because it has to do more multiplications and possibly some modulo n reductions as Equation (2.9) doesn't hold anymore. We will discuss this idea in more detail in chapter 5.

2.3.2 Pre-computing products of primes

To decrease the number of multiplications that has to be done by PoP in 2.2.3, one can pre-compute some products of primes and put that in a list so that PoP can look it up from a table.

More precisely, consider again the block-length k. Let b be a divisor of k and let $\bar{k} = k/b$. Let $x_{(j)}$ $(j = 1, \cdots)$ denote the j-th bit of x. Define

$$v_{i,t} = \prod_{j=1}^{b} p_{(i-1)b+j}^{t_{(j)}},$$

where $i = 1, \cdots, \bar{k}$ and $t = 0, \cdots 2^b - 1$. Finally let $m[j]$ denote the j-th b-bit chunk of message m being the binary representative of some non-negative number $\leq 2^b$. Then, the line

$$x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_i^{m_j \cdot k + i}$$

of Algorithm 2.2.1 can be replaced by

$$x_{j+1} = x_j^2 \times \prod_{i=1}^{\bar{k}} v_{i,m[j\cdot\bar{k}+i]}.$$

And the new code for PoP in implementation 2.2.3 becomes:

Product of Primes (PoP):
for$(i = 1; j \leq \bar{k}; i{+}{+})\{$
$t = t * v_{i,m[j\cdot\bar{k}+i]}$;
$\}$

This idea only affects the implementation of Classic VSH and, therefore, it has no effect to the size of $\mathcal{L}$ or the size of the modulus implied by the computational VSSR assumption.

2.3.3 Fast VSH

The idea of pre-computing primes can be improved by replacing the products of primes $v_{i,t}$ by just primes $p_{(i-1)2^b+t+1}$, which are much smaller. Moreover, dropping the condition that b should be a divisor of k and by letting $\bar{k} = bk$, it follows that each iteration processes more message bits. This idea combines the two ideas of the previous subsections and leads to Fast VSH, see algorithm 2.3.1. For some integer b, let again $m[j]$ denote the j-th b-bit chunk of m.

The straightforward implementation for Fast VSH is given by 2.2.3 except that now PoP is given by:

Product of Primes (PoP):
for$(j = 1; j \leq k; j{+}{+})\{$
$t = t * p_{(j-1)2^b+m[ik+j]+1}$;
$\}$

Because Fast VSH involves $k \cdot 2^b$ small primes instead of k, the size of the modulus n should again increase by the computational VSSR assumption to maintain the same level of security. We show in Appendix C.1 that the changes from Classic VSH to Fast VSH do not affect the security proof. We will show here that Fast VSH requires a single multiplication mod n per $\Omega(\log n)$ message-bits.

Algorithm 2.3.1. Fast VSH

Input: An l-bit message m, an integer b and an S-bit RSA modulus n.

Output: An S-bit hash of m.

Procedure: *Initialization:*
(Initial Value): Let $x_0 = 1$.
(Number of iterations): Let $\mathcal{L} = \lceil \frac{l}{bk} \rceil$.
(padding): Let $m_i = 0$ for $l < i \leq \mathcal{L}k$.
(Appending length block): Let l_i be bits so that $l = \sum_{i=1}^{k} l_i 2^{i-1}$ and let $m_{\mathcal{L}k+i} = l_i$ for $i = 1, \cdots, k$.
Iteration:
For $j = 0, \cdots, \mathcal{L}$ compute
$x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_{(i-1)2^b + m[jk+i]+1} \mod n$.
Finalization:
Return $x_{\mathcal{L}+1}$.

In order to do this, the choice for the block length k is changed to be the maximal value such that

$$\prod_{i=1}^{(k+1)2^b} p_i \leq (2n)^{2^b}. \tag{2.10}$$

The following proposition ensures that PoP is still equivalent to a multiplication modulo n.

Proposition 2.3.2. *Given an RSA modulus n, if k is chosen such that Equation* (2.10) *holds, then,*

$$\prod_{i=1}^{k} p_{i2^b} < n.$$

Proof. From Equation (2.10) it follows that

$$\prod_{i=1}^{(k+1)2^b} p_i = \prod_{t=1}^{2^b} \prod_{i=0}^{k} p_{i2^b+t} \leq (2n)^{2^b}.$$

Because $p_i < p_j$ for all $i < j$ it follows that

$$\left(\prod_{i=0}^{k} p_{i2^b+1} \right)^{2^b} < \prod_{t=1}^{2^b} \prod_{i=0}^{k} p_{i2^b+t} \leq (2n)^{2^b},$$

so that $\prod_{i=0}^{k} p_{i2^b+1} < 2n$. It also follows that $\prod_{i=1}^{k} p_{i2^b} < \prod_{i=1}^{k} p_{i2^b+1}$ so that

$$\prod_{i=1}^{k} p_{i2^b} < \prod_{i=1}^{k} p_{i2^b+1} = \frac{\prod_{i=0}^{k} p_{i2^b+1}}{p_1} \leq \frac{2n}{2}.$$

Hence $\prod_{i=1}^{k} p_{i2^b} < n$. □

Lemma 2.3.3. *Fast VSH requires a multiplication per* $\Omega(\log n)$ *message bits using a straightforward implementation, if* 2^b *is chosen as a fixed positive power of* $\log n$*, say* y*.*

Proof. If k is replaced by $(k+1)2^b$ and n by $(2n)^{2^b}$, then claim 2.2.4 implies

$$(k+1)2^b \sim \frac{2^b \log(2n)}{\log(2^b \log(2n))},$$

So that

$$k \sim \frac{\log(2n)}{\log(2^b \log(2n))} - 1. \tag{2.11}$$

Therefore, PoP can be computed in

$$O\left(k \cdot (\log n)(\log\log n)\right) = O\left(\frac{\log(2n) \cdot (\log n)(\log\log n)}{\log(2^b \log(2n))}\right) = O\left((\log n)^2\right)$$

time. As every iteration from algorithm 2.3.1 processes bk message bits, it follows that Fast VSH requires a single multiplication per $\Omega(bk)$ message bits.

From Equation (2.11) it also follows that

$$bk \sim \frac{b \log(2n)}{\log(2^b \log(2n))} - b.$$

Let $y \in \mathbb{R}^+$ be the fixed value, for which b is chosen so that $2^b = (\log 2n)^y$, it follows that $b = y \log\log 2n / \log 2$, so that

$$\frac{b \log(2n)}{\log(2^b \log(2n))} - b = \frac{y}{y+1} \cdot \frac{(\log\log 2n)(\log(2n))}{(\log 2)(\log\log 2n)} - \frac{y \log\log 2n}{\log 2} \sim \frac{y}{y+1} \cdot \frac{1}{\log 2}(\log n).$$

Hence, PoP of Fast VSH requires a multiplication mod n. As Sq and Mul are unchanged with respect to Classic-VSH it holds that Fast VSH requires a multiplication per $\Omega(\log n)$ message bits. □

2.3.4 The Results of [6] concerning the Speed of VSH

This subsection will give the results that [6] achieves by implementing the four variants of VSH. The implementations are done in C, using the GNU Multiple Precision library (GMP). This is nothing more than a library, which performs arithmetic on large numbers very efficiently. We will give more details in the next chapter. The implementations are run on a 1GHz Pentium III computer. In the following table, S denote the bit-size of the RSA modulus n. And S' denotes the corresponding size of an RSA modulus, such that the particular VSH variant give the same level of security as RSA by the computational VSSR assumption 2.4.

In Chapter 4, 5 and 6 we will discuss how we improve the speed of VSH even more and in Chapter 7 we will explain why doubling the security level (S') will not result in a slowdown of factor 8 for VSH as it would for RSA, instead, we will show in Chapter 7 that the slowdown factor for VSH is linear (i.e. ≈ 2).

S'	Variant	# small primes	S	b	# precalculations	Megabyte/second
1024	Classic VSH	152	1234	1	0	0.355
	Adding Small primes	1024	1318	1	0	0.419
	Pre-Calculate products			8	128*256	0.486
	Fast VSH	2^{16}	1516	8	0	1.135
2048	Classic VSH	272	2398	1	0	0.216
	Adding Small primes	1024	2486	1	0	0.270
	Pre-Calculate products			8	128*256	0.303
	Fast VSH	2^{18}	2874	8	0	0.705

Table 2.2: The results of [6]

2.4 Some Possible Practical Applications of VSH

This section discusses some possible practical applications of VSH that [6] introduces. It follows from this discussion that indeed VSH can be very useful in signature schemes as it provides provable collision resistance. We will discuss two applications that [6] suggests. To give an idea about signature schemes we will treat one simple application in full detail, while we will only briefly remark the second application.

In chapter 6 we present a very fast variant of VSH, which makes use of 'trapdoor' information. Precisely, it uses n's factorization. This implies that this variant cannot be used in general, because obviously n's factorization needs to remain secret. However, in the following signature schemes, the signer chooses his own modulus and has, therefore, the possibility to use this variant.

Basically, a signing scheme is a scheme in which a *signer* proves to a *verifier* that he has read, agreed to, etc. a message m. Therefore, a signature scheme is called secure if no adversary is able to impersonate the signer on any message m. This leads to the following requirement of a signature scheme. See also [2].

Definition 2.4.1. *A signature scheme is existentially unforgeable under an adaptive chosen message attack if there exist no polynomial time adversary A making at most q signature queries to (possibly adaptively chosen) messages $m_1, \cdots, m_q$ that is able with non-negligible probability to produce a valid signature $(\tilde{m}, \tilde{\sigma})$, where $\tilde{m} \neq m_i$ for all $i = 1, \cdots, q$.*

2.4.1 'Hash-then-sign' RSA Signatures

Because the size of the output of VSH comparable to the size of the output of RSA, it makes sense to construct a 'hash-then-sign' RSA signature scheme using VSH. However, as [6] shows in section 3.3, this has to be done carefully. It suggests the following construction scheme:

step 1: Defining the short (S-bit) message signature scheme Let $\bar{n}$ be an $(S+1)$-bit RSA modulus and n be an S-bit VSH modulus. Choose $\bar{n}$ and n independently at random. Specify a one-to-one one-way encoding function $f : \{0,1\}^S \to \{0,1\}^S$. Here one-way means that f is pre-image resistance as defined in the introduction. Finally, define the signature scheme using signing function $\sigma_{\bar{n}}(m) := (f(m))^{1/e}$, where e denotes the public RSA-key. The function f has to be chosen such that the signature scheme with signing function $\sigma_{\bar{n}}$ is existentially unforgeable under adaptively chosen message attacks.

step 2: Defining the signature scheme on messages of arbitrary length Let $(\bar{n}, n, e)$ be the signer's public key. Define the signature scheme using signing function $\sigma_{\bar{n},n}(m) := \sigma_{\bar{n}}(VSH_n(m))$.

The resulting signature scheme is then as follows:

- *Key generation:* Choose random $S/2$ bit primes $p, q, \bar{p}$ and a random $(S/2+1)$-bit prime $\bar{q}$. Let $\bar{n} = \bar{p} \cdot \bar{q}$ and $n = p \cdot q$. Choose d co-prime to $\phi(\bar{n})$, the private RSA-key, and calculate $e = 1/d \bmod \phi(n)$, the public RSA-key. The public key is $(\bar{n}, n, e)$.
- *Signing:* If the signer want to sign message m, he computes $\sigma := \sigma_{\bar{n},n}(m)$ and sends the pair (m, σ) to the verifier.
- *Verifying* Given the signer's public key and the signature (m, σ), the verifier computes $\alpha := f(VSH_n(m))$ and $\beta := \sigma^e \bmod \bar{n}$ and checks whether $\alpha = \beta$. If so, only then he accepts.

It can be proven as shown in Appendix C.2 that this scheme is existentially unforgeable under adaptively chosen message attacks under the assumptions that $\sigma_{\bar{n}}$ is and that VSH is collision resistant.

2.5 Cramer-Shoup Signature Scheme

Another signature scheme is the Cramer Shoup signature scheme. According to [6] it is the most efficient factoring-based signature scheme that is provable secure. This signature scheme uses a so called 'Randomized Trapdoor Hash Function'. In section 4 of [6] it is shown how to turn VSH into such function and how to alter the Cramer-Shoup scheme using VSH, without losing its provable properties. We will not give the details here.

It follows that the verification in this altered Cramer-Shoup Scheme can be done about twice as fast using the VSH based 'Randomized Trapdoor Hash Function'. With the idea discussed in Chapter 6, we will show that we also may expect an increase of the speed for the signing procedure, as the time required to derive the hash of the message to be signed can be reduced to one third of the time that Fast-VSH requires.

Some drawbacks are, however, that because of VSH's output length the altered Cramer-Shoup Scheme has a much larger public key and unlike the original scheme, some pre-calculations are required. We think that these drawbacks are acceptable as the efficiency of the scheme is improved.

Chapter 3

Multiple Precision Algorithms

From the description of VSH in the previous Chapter, it follows that VSH calculates the hash of a message by mostly modular multiplications of some large numbers. So the performance of VSH depends very much on the multiplication algorithms that are used. These algorithms are often called *Multiple Precision algorithms*. Basically, the Multiple Precision multiplication algorithms derive the product of two large numbers (Multiple Precision product) by splitting the Multiple Precision product into small (Single Precision) products and additions.

Similar to Contini et al. ([6]), because GNU's Multiple Precision library is one of the most efficient libraries to perform multiple precision calculations, we will implement VSH in the C-computer language using GMP. According to the manual of GMP ([14]), the GMP multiplication function decides what multiplication algorithm to use. The next Chapter will show how one can try to force GMP to perform some very efficient algorithm to perform multiple precision multiplications.

This Chapter presents very briefly the algorithms that are used by GMP. In addition, we will present Barrett's Modular Reduction Algorithm that is not used by GMP. We will implement VSH based on this algorithm and show the results of the speed of VSH.

In this chapter, let a Single Precision multiplication (SP-multiplication) be a multiplication between two integers of at most β-bits. For a Multiple Precision α-bit integer A, let $A_{(\beta)}$ denote the number of β-bit integers required to express A. So $A_{(\beta)} = \lceil \alpha/\beta \rceil$, and

$$A = \sum_{i=1}^{A_{(\beta)}} a_i 2^{\beta(i-1)},$$

where a_i are β-bit integers for all $i = 1, \cdots, A_{(\beta)}$ and $a_{A_{(\beta)}} \neq 0$. We will also use the radix β representation of A, i.e., $A = (a_{A_{(\beta)}} \dots a_2 a_1)_\beta$.

The first section presents some multiple precision algorithms, used by GMP. The second section presents Barrett's Modular Reduction and its effect on our implementation of VSH.

3.1 Multiple Precision Algorithms used by GMP

This section presents some Multiple Precision algorithms that are used by GMP according to Section 16 of [14]. To give a proper introduction for the remainder of this thesis, we will present the Classical Algorithms in detail and consider advanced multiplication algorithms.

The advanced algorithms that GMP uses to multiply are based on splitting the inputs. We will, therefore, describe the most simple example, Katatsuba's Algorithm, and only remark the name and efficiency of the other algorithms.

The first subsection presents the Classical Algorithms to perform Multiple Precision Multiplication, Squaring, Division and Modular Reduction. The second subsection Describes Karatsuba's algorithm to perform Multiple Precision Multiplications.

3.1.1 Classical Algorithms

This subsection presents the Classical Multiple Precision Algorithms for Multiplication, Squaring, Division, and Modular Reduction, which can be found in [19] and in Chapter 14 of [22]. We will give the algorithms and their efficiency, by means of counting the number of Single Precision (SP) multiplications and SP-divisions as they are, usually, the most time consuming instructions of the algorithms.

Multiplication

Consider the product $A * B$, where $A = (a_k a_{k-1} \dots a_1)_\beta$ and $B = (b_l b_{l-1} \cdots b_1)_\beta$. Algorithm 3.1.1 gives the Classic Multiple Precision Multiplication Algorithm, which shows algorithmically how multiplication by hand is performed.

Algorithm 3.1.1. Multiple Precision Multiplication

Input: The multiple precision integers $A = (a_k a_{k-1} \dots a_1)_\beta$ and $B = (b_l b_{l-1} \dots b_1)_\beta$.

Output: The product $A * B = D = (d_{k+l} d_{k+l-1} \dots d_1)_\beta$.

Procedure:
Set $d_i = 0$ for $i = 1, \cdots, k+l$.
For $i = 1, \cdots, k$ do:
  Set $c = 0$.
  For $j = 1, \cdots, l$ compute:
    $(uv)_\beta = d_{i+j-1} + a_j b_i + c.$ (i)
    Set $d_{i+j-1} = v$.
    Set $c = u$.
  Set $d_{i+k} = u$.
Return $D = (d_{k+l} \dots d_1)_\beta$.

The bottle-neck of this algorithm is Equation (i) in the inner-loop. Equation (i) contains 1 SP-multiplication and is computed $k \cdot l$ times. Hence Algorithm 3.1.1 requires $k \cdot l$ SP-multiplications.

Squaring

Consider the square A^2, where $A = (a_k a_{k-1} \dots a_1)_\beta$. Algorithm 3.1.2 gives the Classic Multiple Precision Squaring Algorithm. To avoid overflow, we change the description of Algorithm 3.1.2 slightly with respect to the algorithm given in [22].

Algorithm 3.1.2. Multiple Precision Squaring

Input: The multiple precision integer $A = (a_k a_{k-1} \dots a_1)_\beta$.

Output: The square $A^2 = D = (d_{2k} d_{2k-1} \dots d_1)_\beta$.

Procedure:
Set $d_i = 0$ for $i = 1, \cdots, 2k$.
For $i = 1, \cdots, k$ do:
    Compute $(uv)_\beta = d_{2i-1} + a_i \cdot a_i$.
    Set $d_{2i-1} = v$.
    Set $c_1 = u$.
    For $j = i+1, \cdots, k$ compute:
        $(wuv)_\beta = d_{i+j-1} + 2a_j a_i + (c_2 c_1)_\beta$.
        Set $d_{i+j-1} = v$.
        Set $(c_2 c_1)_\beta = (wu)_\beta$.
    Set $d_{i+k} = u$.
Return $D = (d_{2k} \dots d_1)_\beta$.

The outer loop of Algorithm 3.1.2 requires k SP-multiplications and the inner loop requires

$$\sum_{i=1}^{k} \sum_{j=i+1}^{k} 1 = \sum_{i=1}^{k} (k - i + 1) = k \sum_{i=1}^{k} 1 - \sum_{i=1}^{k} i + \sum_{i=1}^{k} 1 = k^2 - \frac{k(k+1)}{2} - k = \frac{k^2 - k}{2}$$

SP-multiplications, where the multiplication by 2 is not counted, because multiplication by two can be evaluated in much less time than the time an SP-multiplication requires. Hence Algorithm 3.1.2 requires $1/2(k^2 + k)$ SP-multiplications, which is, if k is large, about half of the number of SP-multiplications that 3.1.1 would require for this multiplication.

Division

Consider the division A/B, where $A = (a_k a_{k-1} \dots a_1)_\beta$ and $B = (b_l b_{l-1} \cdots b_1)_\beta$ and $k \geq l \geq 2$. Algorithm 3.1.3 gives the Classic Multiple Precision Division Algorithm. According to [22] the division algorithm is optimal when $b_l \geq \left\lfloor \frac{\beta}{2} \right\rfloor$ and β is even. Because we will apply these algorithms on a binary computer we will assume that β is even. To guarantee that $b_l \geq \beta/2$ on a binary computer, one just shifts the bits of B so that the bit-size of B is a multiple of β and so that the most significant bit is equal to one. This is called *normalization*. Of course, the bits of A should be shifted similar, to get the result of A/B.

Algorithm 3.1.3. Multiple Precision Division

Input: The (normalized) multiple precision integers $A = (a_k a_{k-1} \dots a_1)_\beta$ and $B = (b_l b_{l-1} \dots b_1)_\beta$, with $k \geq l \geq 2$.

Output: The quotient $Q = (q_{k-l} \dots q_1)_\beta$ and remainder $R = (r_l \dots r_1)_\beta$ so that $A = QB + R$.

Procedure:
Set $q_i = 0$ for $i = 1, \cdots, k-l$.
If $(A \geq B\beta^{k-l})$ then do:
  Set $q_{k-l} = 1$.
  Set $A = A - B\beta^{k-l}$.
For $i = k, k-1, \cdots, l+1$ do:
  If $a_i = b_l$, then set $q_{i-l-1} = \beta - 1$.
  Else set $q_{i-l-1} = \lfloor (a_i\beta + a_{i-1})/b_l \rfloor$.
  While$(q_{i-l-1}(b_l\beta + b_{l-1}) > a_i\beta^2 + a_{i-1}\beta + a_{i-2})$ do:
    Set $q_{i-l-1} = q_{i-l-1} - 1$.
  Set $A = A - q_{i-l-1}B\beta^{i-l-1}$.
  If $A < 0$ then set $A = A + B\beta^{i-l-1}$ and set $q_{i-l-1} = q_{i-l-1} + 1$.
Set $R = A$.
Return (Q, R).

Suppose that the normalized values of A and B are obtained by multiplication with β^j. Then obviously $(A\beta^j)/(B/\beta^j) = A/B$, so that Q remains unchanged after normalization. But $R = \beta^j(QB - A)$, so that the remainder of the division of A by B can be obtained by dividing R by β^j.

When counting the number of SP-multiplications, we do not count the multiplications of the form $A * \beta^j$, as these are just bitshifts:

$$A \cdot \beta^j = (a_k \dots a_1)_\beta \cdot \beta^j = \beta^j \cdot \sum_{i=1}^{k} a_i \beta^{i-1} = \sum_{i=1}^{k} a_i \beta^{j+i-1} = (a_k \dots a_1 0 \dots 0)_\beta.$$

According to [22] the `while` loop is evaluated not more than twice. So we observe that each loop requires (using Algorithm 3.1.1) at most 4 SP-multiplications to evaluate $q_{i-l-1}(b_l\beta + b_{l-1})$ and l-SP-multiplications to evaluate $q_{i-l-1}B$. Assuming that normalization adds one β digit to A it holds that the `for`-loop loops $k - l$ times. So Algorithm 3.1.3 requires about $(k-l)(l+4)$ SP-multiplications. According to [22] Algorithm 3.1.3 can be improved so that it requires about $(k-l)(l+3)$ SP-Multiplications. The `for`-loop consists of one SP-division. So Algorithm 3.1.3 requires about $(k-l)$ SP-Divisions.

Multiple Precision Modular Reduction

Consider the modular expression $A = B \bmod C$, where A is the least nonnegative representative. The Classic algorithm to derive A from B and C is simply the Multiple Precision Division Algorithm given by Algorithm 3.1.3. The output R of Algorithm 3.1.3 on the inputs B, C, gives the least nonnegative representative of $B \bmod C$.

3.1.2 Fast Multiplication Algorithms used by GMP

This subsection will very briefly describe how Karatsuba's Multiplication Algorithm splits the inputs so that it can perform Multiple Precision Multiplication using less SP-multiplications than the Classic Multiple Precision Multiplication Algorithm 3.1.1. The more advanced Multiplication Algorithms that GMP applies are based on similar observations, which we will not mention here. This section concludes with a table providing the (asymptotic) running times of the Multiplication Algorithms that GMP uses.

Consider two Multiple Precision numbers A and B of equal size. So $A_{(\beta)} = B_{(\beta)}$. Let $b = \beta^{\lceil A_{(\beta)}/2 \rceil}$. If we split the numbers A and B into two equal parts, we can write $A = (A_1 A_2)_b$ and $B = (B_1 B_2)_b$. Observe that

$$\begin{aligned} A \cdot B &= A_0 \cdot B_0 + A_0 \cdot B_1 b + A_1 b \cdot B_0 + A_1 b \cdot B_1 b \\ &= (A_1 \cdot B_1) b^2 + ((A_1 - A_0) \cdot (B_1 - B_0) - A_0 \cdot B_0 - A_1 \cdot B_1)\, b + A_0 B_0. \end{aligned} \tag{3.1}$$

The Classic Multiplication Algorithm would compute the four products $A_0 \cdot B_0$, $A_0 \cdot B_1 \cdot b$, $A_1 \cdot B_0 \cdot b$, and $A_1 \cdot B_1 \cdot b^2$, and add them together to obtain the result. Equation (3.1) imply that the same product can be derived by calculating just the three products $A_1 \cdot B_1$, $A_0 \cdot B_0$, and $(A_1 - A_0) \cdot (B_1 - B_0)$. Karatsuba's Algorithm is given by recursively splitting the inputs into two (almost) equal parts and by evaluating the three products $A_1 \cdot B_1$, $A_0 \cdot B_0$, and $(A_1 - A_0) \cdot (B_1 - B_0)$, to evaluate Equation (3.1), see Algorithm 3.1.4. Because $A_1 - A_0$ can be negative, we define σ_X by

$$\sigma_X := \begin{cases} +1 & \text{if } X \geq 0 \\ -1 & \text{if } X < 0 \end{cases},$$

so that, for example, A is given by $\sigma_A (A_1 A_0)_b$.

Adding the two numbers A and B requires $O(A_{(\beta)})$ SP-additions and assuming that an addition can be done in $O(1)$ time, the addition of A and B takes $O(A_{(\beta)})$ time. Also subtraction and bit-shifting (i.e. multiplying with a power of β) takes $O(A_{(\beta)})$ time. If we define $T(n)$ as the time that Karatsuba's algorithm takes to compute the product of two $A_{(\beta)}$ digit numbers, it follows that the time to evaluate the product $A \cdot B$ satisfies

$$T(A_{(\beta)}) = 3T\left(\frac{A_{(\beta)}}{2}\right) + cA_{(\beta)} + d,$$

fore some constants c and d. It follows from the Master Theorem of recursive forms (see Chapter 4 of [8]), that since

$$cA_{(\beta)} = O\left(A_{(\beta)}^{\log 3/\log 2 - \epsilon}\right),$$

for any $\epsilon > 0$, $T(A_{(\beta)}) = O(A_{(\beta)}^{\log 3/\log 2}) = O(A_{(\beta)}^{1.585})$.

Algorithm 3.1.4. Karatsuba's Algorithm

Input: The Multiple Precision integers $A = (a_k a_{k-1} \dots a_1)_\beta$ and $B = (b_l b_{l-1} \dots b_1)_\beta$, with $k = l$.

Output: The product $A \cdot B$.

Procedure: For $X = \sigma_X(x_n \dots x_1)_{(\beta)}$ and $Y = \sigma_Y(y_{n'} \dots y_1)_\beta$, define $f(X, Y)$ by:
If $\min\{n, n'\} < 2$: return $X * Y$.
Else do:
Set $b = \lceil \min\{n, n'\}/2 \rceil$.
Set $X_1 = \sigma_X(x_n \dots x_{b+1})_{(\beta)}$ and $X_0 = \sigma_X(x_b \dots x_1)_{(\beta)}$.
Set $Y_1 = \sigma_Y(y'_n \dots x_{b+1})_{(\beta)}$ and $Y_0 = \sigma_Y(y_b \dots y_1)_{(\beta)}$.
Calculate $W = f(X_1, Y_1)$.
Calculate $U = f(X_0, Y_0)$.
Calculate $V = f((X_1 - X_0), (Y_1 - Y_0))$.
Return $\beta^{2b} W - \beta^b (V - W - U) + U$.
Return $f(A, B)$.

If a SP-multiplication takes $O(1)$ time, it follows that the Classical Multiple Precision Multiplication Algorithm would take $O(A^2_{(\beta)})$ time to evaluate the product of A and B. So indeed, Karatsuba's Algorithm is a lot faster than the classical Algorithm if $A_{(\beta)}$ is large.

Table 3.1 gives the names of the Multiplication Algorithms that GMP uses according to Chapter 16 of [14], the threshold for the input length, the number of parts in which the inputs are split, and gives the running time when applied to perform a multiplication of two N-bit numbers.

Algorithm Name	Threshold:	# parts the input is split	Running Time
Classic Multiplication	none		$O(N^2)$
Karatsuba's Multiplication	`MUL_KARATSUBA_THRESHOLD`	2	$O(N^{\log 3/\log 2}) \approx O(N^{1.585})$
Toom-3 way Multiplication	`MUL_TOOM3_THRESHOLD`	3	$O(N^{\log 5/\log 3}) \approx O(N^{1.465})$
FFT Multiplication	`MUL_FFT_THRESHOLD`	2^k	$O(N^{k/(k-2)})$

Table 3.1: Multiplication Algorithms that are used by GMP

A multiplication of an N-bit number A and and M-bit number B, where $M \leq N$, using GMP can be implemented by just calling the `mpz_mul` function. This function decides, whether it has to square and what Multiplication Algorithm it is going to apply to evaluate the multiplication (or square). If $N > M$, then the `mpz_mul`-function will split A into two parts if necessary. For example, if $N \geq M >$`MUL_KARATSUBA_THRESHOLD`, then the `mpz_mul` function will split A into an $N - M$-bit piece A_1 and an M-bit piece A_0, apply Karatsuba's algorithm to calculate $A_0 \cdot B$, and decides again what Algorithm it is going to apply to perform the remaining multiplication of $A_1 \cdot B$, maybe again after splitting either A_1 or B. So this function recursively splits the input into equal parts until the input lengths are below `MUL_KARATSUBA_THRESHOLD`.

3.2 Barrett Modular Reduction

An efficient algorithm to perform reductions modulo $n = (n_k \dots n_1)_\beta$ is given by Barrett's Algorithm 3.2.1. This Algorithm uses precalculations that are based on the modulus n only, to avoid SP-divisions in the main Algorithm. This Algorithm is in particular applicable to VSH as all modular calculations by VSH are based on the same modulus, so the pre-calculations have to be done only once.

Algorithm 3.2.1. Barrett Modular Reduction

Input: Positive integers $A = (a_{2k}a_{2k-1} \dots a_1)_\beta$ and $n = (n_k \dots n_1)_\beta$, and $\mu = \lfloor \beta^{2k}/n \rfloor$.

Output: $R = A \bmod n$.

Procedure:
Calculate $q_1 = \lfloor A/\beta^{k-1} \rfloor$.
Calculate $q_2 = q_1 \cdot \mu$.
Calculate $q_3 = \lfloor q_2/\beta^{k+1} \rfloor$.
Calculate $r_1 = A \bmod \beta^{k+1}$.
Calculate $r_2 = q_3 \cdot n \bmod \beta^{k+1}$.
Calculate $R = r_1 - r_2$.
If $R < 0$ then set $R = R + \beta^{k+1}$.
While $R \geq n$ calculate:
    $R = R - n$.
Return R.

This algorithm works only if $A < \beta^{2k}$. The calculation of q_2 takes $k(k+1)$ SP-multiplications if the Classical Multiplication algorithm is applied to q_1 and μ. Similarly, the calculation of $q_3 \cdot n$ requires $k(k+1)$ SP-multiplications. All other derivations are additions, subtractions, and bit-wise operations. So, using the Classic Multiplication Algorithm (Algorithm 3.1.1) to perform the Multiple Precision Multiplication, Algorithm 3.2.1 requires $2k(k+1)$ SP-multiplications and no SP-divisions, whereas the Classic Modular Algorithm requires $k(k+3)$ SP-multiplications and k SP-divisions. As [22] notes, the derivation of $q_2 = q_1 \cdot \mu$ can be done more efficiently as the $k-1$ least significant digits of q_2 do not need to be computed in order to compute q_3. Similarly, for the evaluation of r_2, only the $k+1$ least significant digits of $(q_3 \cdot n)$ need to be computed. This can be done via partial multiple precision multiplication. It follows that Algorithm 3.2.1 can be computed using $k(k+4)$ SP-multiplications and no SP-divisions using partial multiple precision multiplication. (See [22] and [3] for more details.)

Unfortunately, GMP does not support partial multiple precision multiplications. So we don't expect our GMP based implementation of Barrett's Modular Reduction to increase the speed of VSH. Indeed, on our 3.4 GHz Pentium IV system, Classic-VSH using a 1234-bit modulus n processes 1.055 MegaBytes per second, if the Classical Modular Reduction algorithm is applied to perform the mod n reductions, and it processes 0.953 MegaBytes per second, if Barrett's Modular Reduction Algorithm without partial Multiple Precision Multiplication is applied.

Chapter 4

Some Implementation Ideas

This chapter discusses how the straightforward implementations of PoP in the iteration functions of Classic-VSH and Fast-VSH can be improved by means of implementation techniques. The first section will introduce some computer technical issues on which the implementation ideas are based. This chapter is based on Classic-VSH and Fast-VSH as defined in Chapter 2, with the same parameters as given in Section 2.3.4. The next chapter will show how the speed can be improved by altering the parameters and the computational VSSR assumption.

Firstly, we will present the (straightforward) implementations of Classic-VSH and Fast-VSH of [6]. For readability reasons, these are not the exact C source codes. The exact C source codes can be found in Appendix E.2. We prefer the following notation: let $B *_{\text{gmp}} C$ denote the multiplication $A * B$ using the GMP library. If the reduction of B modulo C is done via the GMP library, we will write $B \mod_{\text{gmp}} C$. Lastly, the evaluation of $A > B$ using the GMP library is denoted as $A >_{\text{gmp}} B$.

With respect to Classic-VSH, let k be as defined in Equation (2.9). The straightforward implementation of the iteration of Classic-VSH, based on the GMP library, is given by Pseudo-Code 4.0.2.

In Fast-VSH k is taken to be larger than suggested in Equation (2.10). Therefore, some extra mod n reductions may be required in PoP. The straightforward implementation of the iteration of Fast-VSH, based on the GMP library, is given by:

Some ideas discussed in this chapter require the changing of the order of multiplication to compute the product $\prod_{i=1}^{k} p_i$ in PoP. The following examples show that this can influence the speed, when Classic Multiplication algorithms are applied (see 3).

Again, let a Single Precision multiplication (SP-multiplication) be a multiplication between two integers of at most β-bits. For a Multiple Precision α-bit integer A, let $A_{(\beta)}$ denote the number of computer words to represent A in base 2^{β}. So $A_{(\beta)} = \lceil \alpha/\beta \rceil$, and $A = \sum_{i=1}^{A_{(\beta)}} a_i 2^{\beta(i-1)}$, where a_i are β-bit integers for all $i = 1, \cdots, A_{(\beta)}$ and $a_{A_{(\beta)}} \neq 0$.

Example 4.0.4. Suppose $\beta = 2$. Consider the following 2-bit numbers $a_1 = 01, a_2 = 11$, and $a_3 = 11$. Suppose that $\prod_{i=1}^{3} a_i$ needs to be computed. Firstly, consider the multiplication in the following order:

1. Calculate $t = a_1 * a_2$.
2. Calculate $t = t * a_3$.

Pseudo-code 4.0.2. Iteration of Classic VSH (straightforward)

Input: The iteration number, j, x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$.
Product of Primes (PoP):
for($i = 1; i \leq k; i{+}{+}$) {
if($m_{j \cdot k+i} == 1$) {
$t = t *_{\text{gmp}} p_i$;
}
}
Squaring (Sq):
$x_j = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n$;
Multiplication (Mul):
$x_{j+1} = (x_j *_{\text{gmp}} t) \mod_{\text{gmp}} n$;
Return x_{j+1}.

Pseudo-code 4.0.3. Iteration of Fast-VSH (straightforward)

Input: The iteration number j, x_j, message m and chunk-length b.

Output: x_{j+1}.

Procedure: Set $t = 1$.
Squaring (Sq):
$x_{j+1} = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n$;
Product of Primes (PoP):
for($i = 1; i \leq k; i{+}{+}$){
$t = t *_{\text{gmp}} p_{(i-1)2^b+m[jk+i]+1}$;
if($t >_{\text{gmp}} n$) {
$x_{j+1} = x_{j+1} * t \mod_{\text{gmp}} n$;
$t = 1$;
}
}
Multiplication (Mul):
$x_{j+1} = (x_{j+1} *_{\text{gmp}} t) \mod_{\text{gmp}} n$;
Return x_{j+1}.

The first step requires 1 SP-multiplication and returns $t = 11$, so that step 2 also requires 1 SP-multiplication. Hence this order of multiplication requires 2 SP-multiplications. On the other hand if we consider the following order of multiplication:

1. Calculate $t = a_2 * a_3$.
2. Calculate $t = t * a_1$.

Then, the first step requires 1 SP-multiplication and returns $t = 1001$, so that step 2 requires 2 SP-multiplications. Hence this order of multiplication requires 3 SP-multiplications.

Example 4.0.5. Consider the product of primes $\prod_{i=1}^{k} p_i$. Let $\beta = 32$ and suppose $k = 153$. Using computer algebra software, we find that the computation of this product requires 2678 SP-multiplications if the product is taken as follows: first calculate $t = p_1 * p_2$, then $t = t * p_3, \cdots, t = t * p_k$. But the computation of the product in opposite direction requires 3324 SP-multiplications.

The first section will discuss some computer architectural issues on which some ideas are based. The second section will discuss the idea of speeding up VSH by means of less GMP-function calls. The third section will present and comment on the actual speedups of the VSH implementations on our system.

4.1 Some Computer Technical Issues

This section will describe very briefly what happens in a computer or compiler, which can influence the performance of a program. This discussion is based on [13] and [15]. The first subsection describes how a Computer Processor (Pentium) guesses the outcome of an `if`-statement to increase the efficiency. The second subsection discusses how proper Scheduling can improve the efficiency. Lastly, the third subsection describes what happens if a function is called and how it can lead to less performance.

4.1.1 Pipelining and Branch Prediction

When a computer processor gets an instruction, it has to translate the instruction so that it can execute it. Then, it has to execute it and return the result of execution. Let a *cycle* denote the time unit that a processor needs to complete an instruction. Consider a processor that requires 3 cycles per instruction and processes a second instruction after completing the first instruction. So completing the following ordered list of instructions $A_1, \cdots, A_n$ takes $3n$ cycles by this processor.

This can be improved by considering several compartments of the processor. Suppose, for example, that above processor has the following compartments: Load, Execute and Store. Suppose that each compartment needs exactly the same time to complete it's task. So, if an instruction A arrives, it enters the processor in compartment Load and remains there for one cycle, then it stays for one cycle in compartment Execute and it leaves the processor after staying another cycle in the compartment Store. Figure 4.1 shows that processing a second instruction after the first instruction is completed is very inefficient as most compartments are idle most of the time. Better throughput can be obtained by letting a second instruction enter, once Load is finished with the first instruction as shown in Figure 4.2. This way, the processor needs $n + 2$ cycles to complete the ordered list $A_1, \cdots, A_n$. This technique is called *pipelining*. The set of consecutive compartments through which an instruction flows is called a *pipeline*.

Unfortunately, the processor does not only get ordered lists of instructions. If, for example, the processor gets an `IF` instruction, then it depends on the result of this instruction what instruction the processor should pick next from the list. So this forces the processor to wait until it has completed the `IF` instruction. Such instructions are called

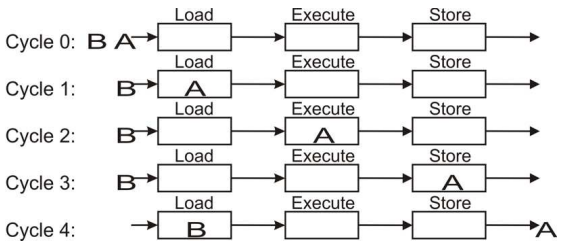

Figure 4.1: Inefficient Processor Throughput

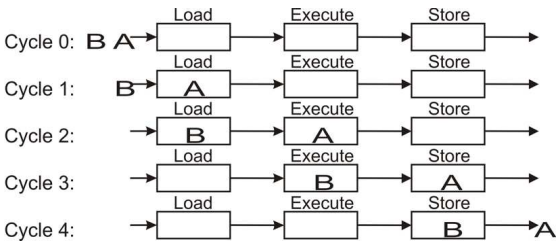

Figure 4.2: Efficient Processor Throughput

branch instructions. Because current processors are far more advanced than our three compartment example, they have much longer pipelines and, therefore, the delay it has when it has to wait for the result of a branch instruction is relatively big.

A solution to this problem is the so called *branch prediction*. Branch prediction is a method in which the processor guesses the output of a branch instruction. Instead of waiting for the result of the branch instruction, it picks the instruction from the branch it guesses. If the result of the branch instruction pops out, the processor checks whether it has guessed right and if so it has saved the time it would otherwise has waited, or if not it empties the pipeline and starts-over by picking the right instruction. So if the processor guesses poorly, then the problem can get worse.

The result of a branch instruction is usually either 0 (false) or 1 (true). If the result is false then the processor picks the next instruction stored in the list, but when the result is true then it jumps to another position in the list of instructions. The early prediction method that the Pentium Family of processors applied (see [13]) is as follows:

- Let R be a 2-bit integer. Set initially $R = 0$.
- If a Branch Instruction enters then predict true as result if $R \geq 2$ and predict false as result otherwise.
- If a Branch Instruction leaves then put $R = \min\{R + 1, 3\}$ if the result of the Branch Instruction is true, put $R = \max\{R - 1, 0\}$ otherwise.

This prediction method was quite poor, so because the length of the pipelines of the advanced processors increased, a better prediction method became necessary. The new Pentium Processors use 16 Registers of the form above and one additional shift register to decide which register to take for prediction. This method is build in such a way that the processor can recognize repetitive patterns of successive results of branch instructions and even deviations of regular patterns. It follows that the processor can predict quite accurate in most cases.

In Chapter 6 we will demonstrate that an implementation of VSH based on predictable sequence of branch instructions leads to significantly better performance compared to an implementation of VSH with a sequence of branch instructions that has random results. Moreover, we will demonstrate that omitting branch instructions where possible, can result in even better performance.

4.1.2 Scheduling

The previous subsection illustrates that pipelined processors can execute multiple instructions in parallel. We have shown that branch instructions may delay the process. But not only branch instructions delay the process. If the second instruction of two successive instructions depends on the result of the first instruction, then the processor has to wait with executing the second instruction until the result of the first instruction is properly stored. For example, consider the following piece of source code:

```
x = x * x;
x = x + 2;
```

Obviously, these instructions cannot be executed in parallel. Firstly, the processor needs access to the result of `x=x*x` before it can start executing `x=x+2`.

These delays can be prevented by proper scheduling. If there are more instructions, which are independent to the instruction `x=x*x`, then placing these instructions before the instruction `x=x+2` will not cause the processor to wait as long as it would when the instruction `x=x+2` is placed right after the instruction `x=x*x`.

We note that `gcc`, the C-compiler, performs these kinds of scheduling if this option is enabled. See section 6.4 of [15].

4.1.3 Function Calls

The following function computes the square of x.

```
sq(int x) {
  int y;
  y = x * x;
  return y;
}
```

When `sq(z)` is used in an implementation, where z is some integer, this is called *calling function* `sq` *with argument* z.

When such *function calls* are used, the processor needs some extra time to perform the call. It has to store the arguments of the function in the memory, then, it has to jump to the beginning of the code of the function, execute the function, and jump back to the original place in the code, where the function is completed; see Figure 4.3. This extra work is called the *function-call overhead*.

If the function contains only a few instructions and the function is called many times then the extra time can become significant. For example, the line `y = sq(x);` causes at least one STORE and two JUMP instructions to be executed by the processor in addition to the instructions caused by $*$, whereas the equivalent line `y = x*x;` only require the instructions caused by $*$ to be executed. Replacing the code `sq(x)` by `x*x` is called *Function Inlining*.

The C-compiler, `gcc`, performs Function Inlining when a function passes some tests. See also section 6.1.2 of [15].

4.2 Reducing the Function-Call Overhead

This section will describe how we reduce the function-call overhead caused by $*_{\text{gmp}}$ in PoP of implementations 4.0.2 and 4.0.3. Because the function $*_{\text{gmp}}$ (=`mpz_mul`) decides whether

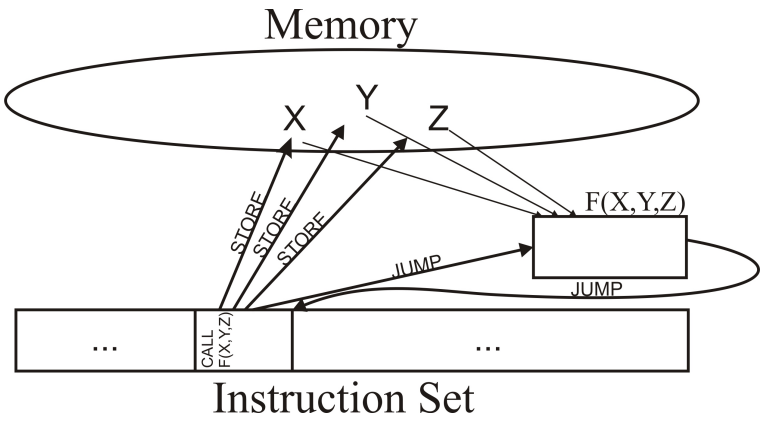

Figure 4.3: Function-Call Overhead

it has to square and what algorithm it is going to use for multiplication, see also section 5.5 and 16.1 of [14], every time it is called it has to perform some tests. Because, initially, the numbers in PoP in Pseudo codes 4.0.2 and 4.0.3 are very small, this extra *overhead* may be relatively large. Also, because we expect that our compiler `gcc` won't *inline* the quite large $*_{\text{gmp}}$ function, we will try to do that in the source code, whenever possible.

Firstly, we will present a method that reduces the amount of times that $*_{\text{gmp}}$ is called. Then, we will show that this method significantly reduces the amount of GMP-function calls for Classic-VSH, but that it hardly reduces the amount of GMP-function calls for Fast-VSH. Lastly, we show how to improve the iteration function of Fast-VSH otherwise.

Pseudo-code 4.2.1. PoP: Filling Integer Array

Input: The iteration number j, and $\mathcal{S} = \{s_1, \cdots, s_v\}$.

Output: $\prod_{i=1}^{v} p_{s_i}$.

Procedure:
Set $t = 1$ and $c = 1$.
Let A be an integer array.
for($j = 1;\ j \leq v;\ j{+}{+}$) {
  if($t * p_{s_j} \geq 2^{32}$) {
    $A_c = t$;
    $c{+}{+}$;
    $t = 1$;
  }
  $t = t * p_{s_j}$;
}
for($j = 1;\ j < c;\ j{+}{+}$) {
  $t = t *_{\text{gmp}} A_j$;
}
Return t.

Roughly the idea is as follows. Assume that the biggest single precision integer of the

computer is 32 bits. Consider PoP of Pseudo codes 4.0.2 and 4.0.3. Evaluate PoP as it is given in these Pseudo codes with the following difference: use single precision multiplications whenever the product $t \cdot p_j$, for some $1 < j \leq k$, fits in one 32-bit word and use $*_{\text{gmp}}$ otherwise. Then, save the value of t in an array, set $t = p_j$, and continue. This way we create an array of integers which we then multiply out using GMP to obtain the result. Pseudo code 4.2.1 makes this precise. Let $\mathcal{S} = \{s_1, \cdots, s_v\}$ denote the ordered set of the indices of the primes over which PoP multiplies, for some v. So $\mathcal{S} = \{i | m_{j \cdot k+i} = 1\}$ for Classic-VSH and $\mathcal{S} = \{(i-1)2^b + m[jk+i] + 1 \mid 1 \leq i \leq k\}$ for Fast-VSH.

Pseudo code 4.2.1 needs to be improved in the line "if($t * p_{s_j} \geq 2^{32}$)". Because we assumed that the largest single precision integer is 32-bits, we cannot define 2^{32} on this computer. Moreover, if $t * p_{s_j} \geq 2^{32}$ then the computer is unable to evaluate $t * p_{s_j}$ properly. We will give two possible implementation solutions to this problem.

Firstly, we consider the largest prime involved, p_k. Assume that p_k has bit-size α. Then, for sure, $t * p_{s_j} < 2^{32}$ if t has bit-size $\leq 32 - \alpha$. In other words: let `mask` denote a 32-bit integer, which is constructed as follows:

$$\texttt{mask} = \sum_{i=1}^{32} a_i 2^{i-1}, \tag{4.1}$$

where $a_i = 0$ for $i \leq 32 - \alpha$ and $a_i = 1$ for $i > 32 - \alpha$, (i.e. setting the α most significant bits to 1 and the remaining bits to zero). The C-statement (`t&mask`) returns 1 if there exist at least one i such that the i-th bit of both t and `mask` equals one and returns zero otherwise. So if (`t&mask`) returns a zero then, for sure, the statement $t * p_{s_j} \geq 2^{32}$ is false. Because t will be less than 2^{32} and `mask` is a 32-bit integer, this can be implemented on the computer.

The advantage of the idea using `mask` is that the statement "if($t * p_{s_j} \geq 2^{32}$)" can be replaced easily by "if(t&`mask`)", which is a relatively cheap comparison as it consists of a bit-wise operation. The disadvantage of this idea is, however, because `t&mask` is equivalent to the statement $t * p_k \geq 2^{32}$, that t may be reset (i.e. set to 1) even if $t * p_{s_j} < 2^{32}$. So this idea may not reduce the amount of GMP function calls as much as Pseudo code 4.2.1 would do.

The following idea deals with this disadvantage. Let `prime_lentgh[]` be a list where `prime_length[i]` denotes the bit-size of p_i. Let `length` denote the intermediate bit-size of t. Then, the statement "if($t * p_{s_j} \geq 2^{32}$)" is equivalent to "if(`length` + `prime_length[`s_j`]` ≥ 32)". The disadvantage from this idea is that more memory is required to store `prime_lentgh[]`, and the altered Pseudo code consists of more instructions than the previous idea. Nevertheless, the extra reduction of the number of GMP function calls can outweigh these disadvantages. The implementation of these ideas are given by respectively Pseudo code 4.2.2 and Pseudo code 4.2.3.

Obviously, Pseudo code 4.2.2 will be effective if p_k is not too large. Because we assumed that the biggest size of a single precision computer integer is 32 bits, Pseudo code 4.2.3 will be effective if the set $\{p_i | i \in \mathcal{S}\}$ consists of a significant amount of primes of bit-size at most 16-bits. We will say that an Pseudo code reduces the amount of GMP function calls *significantly* if it requires at least 25% less GMP function calls than the original Pseudo code. We will now argue that Pseudo codes 4.2.2 and 4.2.3 reduce the amount of GMP function calls significantly for Classic-VSH, but not for Fast-VSH.

The biggest prime that is 16-bits long is p_{6542} (65521). Since $k << 6542$ for Classic-VSH when $S' = 1024$ or $S' = 2048$, every prime p_i, $i \in \mathcal{S}$, is of size at most 16 bits. So both

Pseudo-code 4.2.2. PoP: Filling Integer Array (Testvector)

Input: The iteration number j, mask, and , $\mathcal{S} = \{s_1, \cdots, s_v\}$.

Output: $\prod_{i=1}^{v} p_{s_i}$.

Procedure:

```
Set t = 1 and c = 1.
Let A be an integer array.
for(i = 1; i ≤ v; i++) {
        if(t&mask) {
            A_c = t;
            c++;
            t = 1;
        }
        t = t * p_{s_i};
}
for(i = 1; i < c; i++) {
        t = t *_gmp A_i;
}
Return t.
```

Pseudo-code 4.2.3. PoP: Filling Integer Array (Prime_Length)

Input: The iteration number, j, and the length of the primes, prime_length[], and $\mathcal{S} = \{s_1, \cdots, s_v\}$.

Output: $\prod_{i=1}^{v} p_{s_i}$.

Procedure:

```
Set t = 1, c = 1 and length = 0.
Let A be an integer array.
for(i = 1; i ≤ v; i++) {
        if(length + prime_lentgh[s_i] > 32) {
            A_c = t;
            c++;
            t = 1;
            length = 0;
        }
        t = t * p_{s_i};
        length+ = prime_length[s_i];
}
for(i = 1; i < c; i++) {
        t = t *_gmp A_i;
}
Return t.
```

Pseudo codes 4.2.2 and 4.2.3 reduce the amount of GMP function calls significantly for Classic-VSH. For Fast-VSH, however, where $b = 8$, only the first $\lfloor 6542/256 \rfloor = 25$ small primes p_j, where $j \in \mathcal{S}$, are of size at most 16 bits. As PoP in Fast-VSH computes the product of 256 (see Table 2.2 in Section 2.3.3) small primes if $S' = 1024$ and 1024 small primes if $S' = 2048$, we conclude that Pseudo code 4.2.3 does not reduce the amount of GMP function calls significantly. Also, as it is less effective, Pseudo code 4.2.2 does not reduce the amount of GMP function calls significantly. So we decide to leave Fast-VSH unchanged with respect to these ideas.

Still, the amount of GMP function calls for Fast-VSH can be reduced significantly as follows. Consider Pseudo code 4.0.3 again and observe that for every multiplication performed in PoP, a GMP based comparison is involved. Because the size of t for Fast-VSH is predictable in some sense, we can replace the GMP comparison by a simple comparison as follows. Let w denote the average length of the primes p_j, where $j = 1, 2^8 + 1, 2 \cdot 2^8 + 1, \cdots, (k-1)2^8 + 1$. Then, because on average every product adds w bits to t, perform a $\mod_{\text{gmp}}\ n$ reduction after every S/w multiplications, see Pseudo code 4.2.4, where S denotes the bit-length of the modulus n.

Because w is an average, a mod n reduction may be applied on a later time than the first time when $t > n$. From $p_i \sim i \log i$ it follows that the size of the primes increases logarithmically so that if k is large enough the average will be close to the size of the largest small prime, p_k. So if k is taken large enough then the largest size of t will still be close to S. For example, let again $b = 8$, if $S' = 1024$ and $k = 256$, then $w = 18$, where p_k has bit-size 20

Pseudo-code 4.2.4. Iteration of Fast-VSH (Improved Comparison)

Input: The iteration number, j, x_j, w and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$ and lim= 0 and limit= $\lceil S/w \rceil$.

Squaring (Sq):

$x_{j+1} = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n;$

Product of Primes (PoP):

for($i = 1; i \leq k; i{+}{+}$){

$t = t *_{\text{gmp}} p_{(i-1)2^b+m[jk+i]+1};$

lim++;

if(lim==limit) {

$x_{j+1} = x_{j+1} * t \mod_{\text{gmp}} n;$

$t = 1;$

lim= 0;

}

}

Multiplication (Mul):

$x_{j+1} = (x_{j+1} *_{\text{gmp}} t) \mod_{\text{gmp}} n;$

Return x_{j+1}.

and if $S' = 2048$ and $k = 1024$, then $w = 20$, where p_k has bit-size 22. So if $S' = 1024$, then the size of t is bounded by $\frac{20}{18} \cdot S$ and if $S' = 2048$ then the size of t is bounded by $\frac{22}{20} \cdot S$.

The advantage of this idea is not only that the GMP comparison function is replaced by a simple comparison, but also that the result of the branch instruction "if(lim==limit)" can be predicted easily as it follows a constant pattern. However, as the statement "if($t >_{\text{gmp}} n$)" will be equal to zero most of the time for large n, the gain of better prediction will be small.

4.3 Tree-Based Multiplication

The previous Chapter discussed the very efficient Karatsuba algorithm (Algorithm 3.1.4) for multiple-precision multiplications. From its description it follows that maximal advantage from Karatsuba's algorithm is obtained when the input integers are of (almost) equal size. Most of the time t is much larger than p_k so that Karatsuba will hardly be applied to PoP in Pseudo codes 4.0.2 and 4.0.3.

The following change to the order of multiplication will ensure that more equally sized numbers are multiplied together by PoP. We will refer to this order of multiplication as *Tree-Based Multiplication*. Consider again the ordered set of indices of primes over which PoP multiplies, $\mathcal{S}$. Roughly, the idea is to pair the largest primes with the smallest primes in order to attempt keeping the products about the same size. Then continue pairing until all products are done. More precisely: put the primes into an array $B_0 = (b_{0i})_{i=1}^{v}$, where $b_{0i} = p_{s_i}$. Then calculate B_1 by letting $b_{1i} = b_{0i}b_{0(v-i+1)}$, for $i = 1, 2, \cdots, \lfloor v/2 \rfloor$ and $b_{1(\lceil v/2 \rceil)} = b_{0(\lceil v/2 \rceil)}$ if v is odd. Similar to B_1, B_2 can be computed. Continuing this way, we

end with $B_{\lceil \log_2(v) \rceil}$ which consist of one element, the result of PoP. Figure 4.4 illustrates this idea for $k = v = 8$ and shows why we refer to this multiplication technique as tree-based multiplication. The vertices denote the elements of B_j for some j as shown in the figure and arc (v, w) exists in the graph only if v divides w. Pseudo code 4.3.1 makes this idea precise.

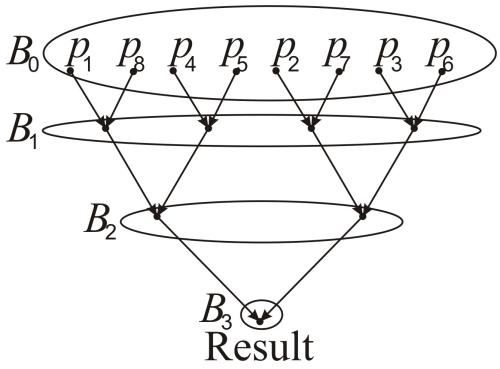

Figure 4.4: Tree Based multiplication

For Classic-VSH the statement "if($B_i >_{\text{gmp}} n$)" can be omitted as k is chosen such that all intermediate products will be less than n. Moreover, as we discussed in previous subsection, we can improve Pseudo code 4.3.1 for Classic-VSH by replacing the line "$B_i = p_{s_i} *_{\text{gmp}} p_{s_{c-i+1}}$;" by "$B_i = p_{s_i} * p_{s_{c-i+1}}$;".

4.4 Results of these Ideas

This section shows what speedups we gain by implementing these ideas for Classic-VSH and Fast-VSH using the same parameters as given in subsection 2.3.4. As we noted in the introduction of this chapter, we have to take into account how the number of single-precision multiplications behaves with respect to the original implementation due to the different order of multiplication. Because the message-bits are taken randomly, the precise number of SP-multiplications cannot be derived in advance. Therefore, we will relate this behavior by considering the all-one message, i.e., the message in which every message bit is equal to one.

The following table will give the results of the performance of the implementations on our system. We use a 3.4 GHz Pentium IV Processor. The source codes of the iteration steps can be found in Appendix E.2. The codes are compiled with `gcc` using the `-O3` tag, which implies that `gcc` will try to perform maximal code-optimization as described in section 6 of [15]. Recall that Pseudo codes 4.0.2 and 4.0.3 correspond to the original versions of Classic-VSH and Fast-VSH respectively as described in [6]. Finally, to give an idea how the order of multiplication influences the amount of work to be done by the PC, the column "# SP-Mults" gives the number of SP-multiplications that the Pseudo code requires to evaluate an all one message block, without performing mod n reductions, if only the Classic Multiplication Algorithm (see Chapter 3) is applied. Because for Fast-VSH the amount of SP-multiplications is also dependent of the place where the mod n reductions are performed, we will leave this column blank for Fast-VSH.

As discussed in Chapter 2, the time needed to evaluate PoP is roughly 1/3 of the total time to evaluate a complete iteration. Therefore, the total speedup cannot be more than

Pseudo-code 4.3.1. PoP:
Tree-based multiplication

Input: The iteration number, j, $\mathcal{S} = \{s_1, \cdots, s_v\}$ and v .

Output: $\prod_{i=1}^{v} p_{s_i}$.

Procedure:

Let B be an integer array.
Set $c = v$. c denotes the size of B.
for($i = 1;\ i \le c/2;\ i{+}{+}$) {
    $B_i = p_{s_i} *_{\text{gmp}} p_{s_{c-i+1}}$;
}
if ($c \bmod 2 == 1$) {
    $B_{c+1} = p_{s_{c+1}}$;
}
$\lfloor c = (c+1)/2 \rfloor$;
while($c > 1$) {
    for($i = 1;\ i \le c/2;\ i{+}{+}$) {
        $B_i = B_i *_{\text{gmp}} B_{c-i+1}$;
        if($B_i >_{\text{gmp}} n$) {
            $B_i = B_i \mod_{\text{gmp}} n$;
        }
    }
    $c = (c+1)/2$;
}
Return B_1.

Note: If this Pseudo-code is applied with respect to Classic-VSH, then the line "if($B_i >_{\text{gmp}} n$)" can be omitted and the line "$B_i = p_{s_i} *_{\text{gmp}} p_{s_{c-i+1}}$;" can be replaced by the more efficient "$B_i = p_{s_i} * p_{s_{c-i+1}}$;".

S'	Pseudo code	# small primes	S	b	# SP-Mults	MB/sec.	speed-up
1024	4.0.2: Classic Original	153	1234		2678	1.055	0.00 %
	4.2.2: Classic Testvector	153	1234		998	1.166	10.52%
	4.2.3: Classic prime_length	153	1234		988	1.150	9.01%
	4.3.1: Classic Tree-based	153	1234		1017	1.093	3.60%
1024	4.0.3: Fast Original	2^{16}	1516	8		3.770	0.00 %
	4.2.4: Fast Improved	2^{16}	1516	8		3.781	0.29 %
	4.3.1: Fast Tree-based	2^{16}	1516	8		3.661	-2.89%
2048	4.0.2: Classic Original	273	2398		9623	0.681	0.00 %
	4.2.2: Classic Testvector	273	2398		3722	0.752	10.43 %
	4.2.3: Classic prime_length	273	2398		3541	0.753	10.57%
	4.3.1: Classic Tree-based	273	2398		3534	0.711	4.40%
2048	4.0.3: Fast Original	2^{18}	2874	8		2.457	0.00%
	4.2.4: Fast Improved	2^{18}	2874	8		2.459	0.08%
	4.3.1: Fast Tree-based	2^{18}	2874	8		2.361	-3.91%

Table 4.1: The results of our Pseudo codes

about 33%. For Classic-VSH we see that the change to the order of multiplication reduces the number of SP-multiplications by about 2/3. So we would expect a speedup of $2/9 \approx 22\%$, disregarding the speedup gained by the reduction of the function call overhead and the possible gain from Karatsuba's Algorithm. We will now try to explain why the actual speedups are less than expected.

With respect to Classic-VSH, the extra work needed to store the intermediate values of the product in an integer array in Pseudo codes 4.2.2 and 4.2.3, may outweigh time saved by reducing the function-call overhead of $*_{\text{gmp}}$ in Pseudo code 4.0.2. Also, it may explain the reduced speedup, because we don't apply the "the usual assembler tricks and obscurities for speed" (see section 16.1.1 of [14]) as GMP does. With respect to the Tree-based multiplication, the profile of the implementation of Classic VSH based on Pseudo code 4.3.1 (using `gprof`) shows that GMP is not applying Karatsuba's Algorithm significantly more than it already did originally. Both in the original versions and in the Tree-based version of Classic- and Fast-VSH Karatsuba's algorithm required a negligible time, see Appendix E.2.5. Since the speed is not improving we conclude that Karatsuba's algorithm is hardly applied by GMP in both the original versions of VSH as the Tree-based versions.

With respect to Fast-VSH, apparently, time saved by reducing the function call overhead of $>_{\text{gmp}}$ in Pseudo code 4.2.4 is negligible. Also, for Fast-VSH Karatsuba's algorithm is not applied more in Pseudo code 4.3.1 than it already was originally. On contrary, the profiler shows that for some reason the original version of Fast-VSH applies Karatsuba's Algorithm more than the Tree-based version.

We will call an Pseudo code *effective* if it speeds-up the original Pseudo code of PoP by at least 25%. Because Pseudo codes 4.2.2 and 4.2.3 makes Classic-VSH about 9% faster, which amount to making PoP about 27% faster, we conclude that these Pseudo codes are effective.

Chapter 5

New Choices for k and the Security Assumption

The previous Chapter discusses some implementation ideas to the iteration part of VSH to improve the speed. This chapter will discuss a more theoretical idea to improve the speed. Mainly, the idea is to reduce the iteration numbers as much as possible by allowing more small primes in the iteration part. In addition, we will introduce a new Computational VSSR assumption of VSH, which is with the present knowledge more realistic. The consequences will be that the security of VSH, using at most $\pi(B)$ small primes, is equivalent to factoring an S'-bit number, where $S = S'$, the size of the VSH modulus n, as opposed to the original Computational VSSR Assumption, where S should be taken larger than S' (see Subsection 2.1.3). Here, $\pi(B)$ denotes the number of primes less than B (see Definition B.1.1), where B is a certain smoothness bound.

Under this assumption it follows that if VSH is based on at most $\pi(B)$ primes then VSH provides the desired security even if its modulus n has size $S = S'$, which is much smaller than suggested by the original Computational VSSR assumption. Therefore, as the modulus is decreased in size, the modular calculations of VSH become less time-consuming. Moreover, we will show that we can take B much larger than suggested in [6], such that $\pi(B)$ is much larger than k as suggested originally by [6]. It follows that we may reduce the amount of iterations by processing more message bits each iteration, while preserving the security of VSH.

Roughly, the observation for Classic-VSH is as follows. Let K denote the number of small primes that is used each iteration. Let k denote the number of small primes as suggested originally by Equation (2.9). Let R_o denote the iteration numbers that Classic VSH requires to derive the hash of an l-bit message m originally, and let R_n denote the iteration numbers that Classic-VSH requires to derive the hash of m using K small primes per iteration. Consider Algorithm 2.2.1. It follows that the ratio of the iteration numbers is approximated by

$$\frac{R_n}{R_o} = \frac{\lceil l/K \rceil + 1}{\lceil l/k \rceil + 1} \approx \frac{k}{K}.$$

Next, consider the iteration of Classic-VSH (see again Pseudo code 4.0.2) and assume that K is a multiple of k. It follows that only PoP has more calculations to perform. If we implement PoP by storing the value of t after every k multiplications in an array and resetting t, and finally multiply this array out mod n, it follows that, roughly, the new PoP

requires K/k old PoP-equivalent and Mul-equivalent operations. So assuming for simplicity that the old PoP, Sq and Mul require the same time (T) each iteration, it follows that the new iteration takes roughly $(2\frac{K}{k}+2)T$ time, as opposed to $3T$ originally. Hence the new iteration takes

$$\frac{\frac{K}{k}\cdot 2+2}{3}=\frac{2}{3}\frac{K}{k}+\frac{2}{3}$$

times more time. So the time needed for Classic-VSH using K small primes requires

$$\frac{k}{K}\cdot\left(\frac{2}{3}\frac{K}{k}+\frac{2}{3}\right)=\frac{2}{3}+\frac{2}{3}\frac{k}{K}\left(\rightarrow\frac{2}{3},\quad K\rightarrow\infty\right)$$

times the time of the original version of Classic VSH. So we expect to increase the speed of VSH by at most 33% by increasing K. Because squaring can be done more efficiently than ordinary multiplying the gain will be less than this rough analysis suggests.

The approximations in this chapter are based on the straightforward Pseudo codes of Classic-VSH and Fast-VSH. See implementations 4.0.2 and 4.0.3 in Chapter 4 on page 26. Note that all approximations in this chapter will be based on the average case, i.e. with probability 1/2 a message bit will be equal to one.

The first section of this chapter will make this idea precise. The second section will present the new Computational VSSR assumption and argue that VSH remains secure under this assumption. The third section will present our results we find when combining these ideas and the ideas of previous Chapter together on our system. Here, we note again that we use a 3.4 GHz Pentium IV system.

5.1 Adding Small Primes

This section discusses how the addition of small primes in each iteration influences the speed of Classic- and Fast-VSH by means of counting the number of SP-multiplications and SP-divisions. Via some number theoretical definitions and theorems given in Appendix B we will approximate these numbers.

The first subsection will discuss this idea with respect to Classic-VSH and the second subsection will discuss this idea with respect to Fast-VSH.

5.1.1 Classic-VSH

This subsection discusses the behavior of the number of SP-multiplications and SP-divisions when altering the number of small primes used by each iteration of Classic-VSH. This subsection uses many number theoretic definitions and theorem that are given in Appendix B. Let K denote the number of small primes used and k denote the original choice of the number of small primes, i.e. k maximal so that $\prod_{i=1}^{k} p_i < n$ (see Equation (2.9) in Chapter 2 on page 10). Furthermore, let m be an l-bit input message.

The first subsection will show how we can approximate the number of single precision calculations needed to perform one Classic-VSH iteration. The second subsection will discuss how the iteration can be improved by means of adding extra mod n reductions. Lastly, the third subsection will compare our approximation to the real implementation on our system.

Approximating the Number of Single Precision Multiplications

This subsection discusses how we approximate the number of single precision multiplications that each iteration requires. These approximations are meant to simplify the calculations of the required number of single precision calculations.

Consider Pseudo code 4.0.2.

Pseudo code 4.0.2. Iteration of Classic VSH (straightforward)

Input: The iteration number, j, x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$.

Product of Primes (PoP):

```
for(i = 1; i ≤ k; i++){
    if(m_{j·k+i} == 1){
        t = t *_gmp p_i;
    }
}
```

Squaring (Sq):

$x_j = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n;$

Multiplication (Mul):

$x_{j+1} = (x_j *_{\text{gmp}} t) \mod_{\text{gmp}} n;$

Return x_{j+1}.

assumptions In our approximations we assume that GMP uses classical methods to perform multiple precision multiplications, squaring and modular reductions as described in chapter 14 of [22]. We also assume that a single precision multiplication is a multiplication of β-bit integers. As there is no general relation between the time needed to perform a single precision multiplication and a single precision division we will treat them separately. Lastly, we assume that the values of the message bits m_i are uniformly distributed.

approximation of Sq From the assumptions above and chapter 14 of [22] it follows that Sq needs approximately

$$\underbrace{\left(n_{(\beta)}^2 + n_{(\beta)}\right)/2}_{x_i^2} \quad + \quad \underbrace{n_{(\beta)}\left(n_{(\beta)} + 3\right)}_{\mod n} \tag{5.1}$$

single precision multiplications and $n_{(\beta)}$ single precision divisions. There is no reason to reduce this number as the probability that x_i can be expressed in less than $n_{(\beta)} - 1$ β-bit words is very small, (see remark 5.1.3 on page 47).

approximation of PoP Again, from chapter 14 of [22] it follows that the multiplication of two integers a and b requires $a_{(\beta)}b_{(\beta)}$ single precision multiplications. So the

multiplication of $P_i := \prod_{j=1}^{i} p_j$ and p_{i+1} requires

$$\begin{aligned} a_{(\beta)}b_{(\beta)} &= \lceil \log_{2^\beta} P_i \rceil \lceil \log_{2^\beta} p_{i+1} \rceil \\ &= \lceil \log_{2^\beta} \prod_{j=1}^{i} p_j \rceil \lceil \log_{2^\beta} p_{i+1} \rceil \\ &= \left\lceil \sum_{j=1}^{i} \frac{\log p_j}{\beta \log 2} \right\rceil \left\lceil \frac{\log p_{i+1}}{\beta \log 2} \right\rceil \\ &= \left\lceil \frac{\theta(p_i)}{\beta \log 2} \right\rceil \left\lceil \frac{\log p_{i+1}}{\beta \log 2} \right\rceil \end{aligned} \tag{5.2}$$

single precision multiplications, where $\theta(x)$ denotes Chebyshev theta function, see Definition B.1.2 and also chapter 2 of [12]. As $\theta(x) \sim x$, by the prime number theorem, we may approximate $\theta(p_i)$ by p_i. So Equation (5.2) becomes:

$$a_{(\beta)}b_{(\beta)} \approx \left\lceil \frac{p_i}{\beta \log 2} \right\rceil \left\lceil \frac{\log p_{i+1}}{\beta \log 2} \right\rceil$$

Thus the straightforward multiplication of all primes $p_1, p_2, \cdots, p_K$ would require approximately

$$\sum_{i=1}^{K-1} \left\lceil \frac{p_i}{\beta \log 2} \right\rceil \left\lceil \frac{\log p_{i+1}}{\beta \log 2} \right\rceil \tag{5.3}$$

single precision multiplications.

Consider again Pseudo-code 4.0.2 and note that PoP calculates the following product

$$\prod_{i=1}^{K} p_i^{m_{jk+i}}.$$

Moreover, if m_{jk+i} is equal to 0 then it does nothing. Hence the total number of SP-multiplications that PoP requires is given by

$$\sum_{i=1}^{K-1} \left\lceil \log_{2^\beta} \left(\prod_{l=1}^{i} p_l^{m_{jk+l}} \right) \right\rceil \lceil \log_{2^\beta} p_{i+1} \rceil \cdot m_{jk+i+1},$$

which can be approximated using the average case assumption that every message bit is equal to one with probability 1/2 to

$$\frac{1}{2} \cdot \sum_{i=1}^{K-1} \left\lceil \log_{2^\beta} \left(\sqrt{\prod_{l=1}^{i} p_l} \right) \right\rceil \lceil \log_{2^\beta} p_{i+1} \rceil .$$

Similar to Equation (5.3) we conclude that this can be approximated by

$$\frac{1}{2} \cdot \sum_{i=1}^{K-1} \left\lceil \frac{1}{2} \frac{p_i}{\beta \log 2} \right\rceil \left\lceil \frac{\log p_{i+1}}{\beta \log 2} \right\rceil . \tag{5.4}$$

Because typically, $\beta = 32$ or a multiple of 32, it follows that each prime p_i can be expressed in one β-bit word –i.e. p_i is smaller than 2^β– for $i \leq \pi(2^\beta)$. Here, $\pi(x)$ denotes the prime-counting function, see Definition B.1.1. For example, if $\beta = 32$ then $\pi(2^{32}) = 202378196$. In classic VSH using a 1024 bit modulus, k is about 132, and, using a 2048 bit modulus, k is about 233. So from a practical point of view, we may assume that all p_i can be expressed in one β-bit word. Hence equation (5.4) becomes

$$\frac{1}{2}\sum_{i=1}^{K-1}\left\lceil\frac{1}{2}\frac{p_i}{\beta\log 2}\right\rceil\left\lceil\frac{\log p_{i+1}}{\beta\log 2}\right\rceil = \frac{1}{2}\sum_{i=1}^{K-1}\left\lceil\frac{1}{2}\frac{p_i}{\beta\log 2}\right\rceil \approx \frac{1}{4}\sum_{i=1}^{K-1}\frac{p_i}{\beta\log 2}. \tag{5.5}$$

This expression can be simplified further by using the following claim, which we will prove in appendix D.1.1:

Claim 5.1.1. *For each integer $x \geq 2$ we have the following relation:*

$$\sum_{p\leq x} p \sim \frac{x^2}{2\log(x)}. \tag{5.6}$$

Because $p_i \sim i\log i$, we approximate p_i by $i\log i$. This is done everywhere in the remainder of this section. Often, for readability reasons, we will still write p_i instead of $i\log i$. Note that this approximation is accurate for big primes because of $\sim$, but not necessarily for small primes. The last subsection shows that this approximation is still good enough for our purposes.

Applying Claim 5.1.1 to Equation (5.5) yields that PoP requires roughly

$$\begin{aligned}\mathrm{PoP}_1(K) &= \frac{1}{4}\cdot\frac{p_{K-1}^2}{2\log(p_{K-1})}\cdot\frac{1}{\beta\log 2} = \frac{p_{K-1}^2}{8\beta\log(p_{K-1})\log 2}\\ &\approx \frac{(K-1)^2\left(\log(K-1)\right)^2}{8\beta\log(2)\log\left((K-1)(\log(K-1))\right)}\\ &\approx \frac{(K\log K)^2}{8\beta\log(2)\left(\log K+\log\log K\right)}\\ &\approx \frac{K^2\log K}{8\beta\log 2}\end{aligned}$$

single precision multiplications.

approximation of Mul Because we assumed that the message-bits are uniformly distributed, it follows that every prime p_i, for $i \leq K$, divides t with probability $1/2$. Therefore, we approximate t by

$$t \approx \sqrt{\prod_{i=1}^{K} p_i}.$$

It follows that

$$\begin{aligned} t_{(\beta)} &\approx \left\lceil \log\left(\sqrt{\prod_{i=1}^{K} p_i}\right) \Big/ \log 2^{\beta} \right\rceil \\ &\approx \frac{1}{2}\sum_{i=1}^{K} \log(p_i) \Big/ \beta \log 2 \\ &= \frac{\theta(P_K)}{2\beta \log 2} \\ &\approx \frac{K \log K}{2\beta \log 2}. \end{aligned} \tag{5.7}$$

So that, approximately, Mul requires

$$t_{(\beta)} n_{(\beta)} + t_{(\beta)}(n_{(\beta)} + 3) \approx \frac{K \log K}{2\beta \log 2}\left(2n_{(\beta)} + 3\right) \tag{5.8}$$

SP-multiplications and $\frac{K \log K}{2\beta \log 2} \cdot n_{(\beta)}$ SP-divisions (see also [22]).

In conclusion, Table 5.1 shows the number of single precision multiplications that are performed by each part of the iteration.

Part	# SP-multiplications	# SP-divisions
Sq	$\left(n_{(\beta)}^2 + n_{(\beta)}\right)/2 + n_{(\beta)}\left(n_{(\beta)} + 3\right)$	$n_{(\beta)}$
PoP	$\left(K^2 \log K\right)/\left(8\beta \log 2\right)$	0
Mul	$(K \log K)\left(2n_{(\beta)} + 3\right)/\left(2\beta \log 2\right)$	$(K \log K) \cdot n_{(\beta)}/\left(2\beta \log 2\right)$

Table 5.1: The cost of each part of the iteration of VSH.

Adding Primes Without Extra mod n Reductions

This subsection shows by how much the number of single precision multiplications that are needed in a VSH run on the ℓ-bit message m can be reduced by multiplying t in PoP with more primes than suggested in [6]. Since K will be larger, the advantage is that the iteration number rounds will be reduced and, therefore, less multiple precision modular multiplications and squarings are required. The disadvantage is that t may become very large, even that large that the extra amount of single precision multiplications required by PoP outweighs the reduced amount of SP-multiplications due to the reduction of the iteration numbers. We will show this mathematically.

The number of rounds needed to derive the VSH hash of m equals:

$$R = \left\lceil \frac{\ell}{K} \right\rceil + 1 \tag{5.9}$$

The total cost to derive the VSH hash for m is then approximated by:

$$\text{Cost}(K) = \underbrace{R\left(\frac{3}{2}n_{(\beta)}^2 + \left(\frac{K \log K}{\beta \log 2} + \frac{7}{2}\right)n_{(\beta)} + \frac{3K \log K}{2\beta \log 2} + \frac{K^2 \log K}{4\beta \log 2}\right)}_{\text{SP multiplications}} + \underbrace{R\left(\frac{K \log K}{2\beta \log 2} + 1\right)n_{(\beta)}}_{\text{SP divisions}}$$

If K is very small, then $R \approx \ell$. In this case the Cost will be $O(l \cdot n_{(\beta)}^2)$ SP multiplications plus $O(l \cdot n_{(\beta)})$ SP divisions, which we may assume very large. On the other hand if we let $K \to \infty$ then Cost will behave like

$$\frac{\text{Cost}(K)}{\ell} > \frac{R \cdot K^2 \log K}{\ell \cdot 4\beta \log 2} > \frac{\ell/K \cdot K^2 \log K}{\ell \cdot 4\beta \log 2} = \frac{K \log K}{4\beta \log 2} \to \infty$$

single precision multiplications.

So we expect a minimum. Figure 1 shows the behavior of the required number of single precision multiplications and divisions, where $\beta = 32$ and K runs from 1 to 1000, on a 1 megabyte random message m. We take the modulus n of equal size as in [6], which is 1234 bits. In our case $k = 153$.

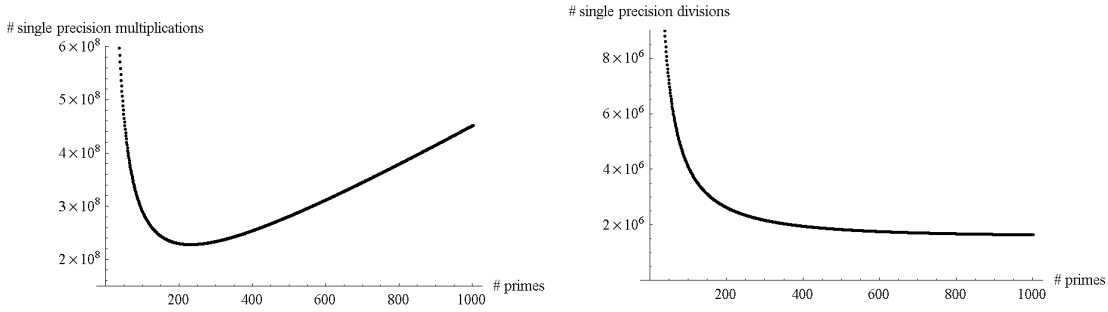

Figure 5.1: Approximate number of Single precision calculations versus K, without adding extra mod n reductions

VSH needs to evaluate approximately $2.42 \cdot 10^8$ single precision multiplications, when $K = k$. According to our assumptions and approximations, figure 1 implies that we can reduce this amount to $2.28 \cdot 10^8$, which is 5.99% less. On the other hand we can reduce the number of single precision divisions by 46.85%.

The next subsection discusses whether this can be improved by applying some additional modular reductions to avoid extremely long sizes of t.

Adding Extra mod n Reductions

This section will describe how the number of single precision multiplications behave if we avoid an extremely large size of t by adding some extra mod n reductions. So Pseudo code 4.0.2 needs to be updated. Pseudo code 5.1.2 presents the updated version of the iteration function of Classic-VSH, where extra mod n reductions are performed.

Obviously the approximation of Section 5.1.1 for Sq is valid here. The remainder of this section discusses the new approximations for PoP and Mul and give the results of this model when applied to different choices for `condition`.

Approximation of the altered PoP and Mul

To approximate the altered PoP and Mul we assume for simplicity that `Condition` is `true` after every fixed r message bits in each iteration.

To approximate the altered PoP, let $P_i = \sqrt{\prod_{j=1}^{i} p_j}$ and let $p_i' \equiv P_{ir} \mod n$, where $P_0 = 1$, and let $s = \lfloor K/r \rfloor$ the number times that PoP performs a mod n reduction. In the

Pseudo-code 5.1.2. Iteration of Classic VSH (extra mod n)

Input: The iteration number j, x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$.
Product of Primes (PoP):
for($i = 1; i \leq k; i{+}{+}$) {
if($m_{j \cdot k+i} == 1$) {
$t = t *_{\text{gmp}} p_i$;
if(condition) $t = t \mod_{\text{gmp}} n$;
}
}
Squaring (Sq):
$x_j = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n$;
Multiplication (Mul):
$x_{j+1} = (x_j *_{\text{gmp}} t) \mod_{\text{gmp}} n$;
Return x_{j+1}.

remainder of this section only, p_0 is equal to 1 instead of -1. Consider t after the $lr + w$-th multiplication in Pseudo-code 5.1.2. It holds that

$$t \equiv \prod_{i=1}^{lr+w} p_i^{m_{jk+i}} \mod n,$$

which is on average equivalent to

$$\sqrt{\prod_{i=1}^{lr+w} p_i} \equiv \sqrt{\prod_{i=1}^{lr} p_i} \sqrt{\prod_{i=lr+1}^{lr+w} p_i} \equiv p'_l \frac{P_{lr+w}}{P_{lr}} \mod n.$$

Next, consider t in Pseudo-code 5.1.2 when the l-th mod n reduction will be performed. So we are considering the lr-th multiplication and the value of t is then on average equal to $p'_l \frac{P_{lr}}{P_{(l-1)r}}$. Hence from the description of the Classical Modular Reduction Algorithm (see Chapter 3) the l-th mod n reduction requires on average

$$\left(\left(p'_l \frac{P_{lr}}{P_{(l-1)r}} \right)_{(\beta)} - n_{(\beta)} \right) \left(n_{(\beta)} + 3 \right)$$

single precision multiplications and

$$\left(\left(p'_l \frac{P_{lr}}{P_{(l-1)r}} \right)_{(\beta)} - n_{(\beta)} \right)$$

single precision divisions. In conclusion PoP of Pseudo-code 5.1.2 requires

$$\underbrace{F(K)}_{\text{multiplications}} + \underbrace{\sum_{i=0}^{s-1} \left(\left(p'_i \frac{P_{(i+1)r}}{P_{ir}} \right)_{(\beta)} - n_{(\beta)} \right) \left(n_{(\beta)} + 3 \right)}_{\text{modular reductions}} \tag{5.10}$$

single precision multiplications, and

$$\sum_{i=0}^{s-1}\left(\left(p_i'\frac{P_{(i+1)r}}{P_{ir}}\right)_{(\beta)}-n_{(\beta)}\right) \tag{5.11}$$

single precision divisions. Here $F:\mathbb{N}\to\mathbb{N}$ is on average given by

$$\begin{aligned}F(K)=&\sum_{i=0}^{s-1}\left(\lceil\log_{2^\beta}p_i'\rceil\lceil\log_{2^\beta}p_{ir+1}\rceil\cdot m_{jk+ir+1}\right.\\&+\sum_{l=ir+1}^{(i+1)r-1}\lceil\log_{2^\beta}\left(p_i'\sqrt{\prod_{w=ir+1}^{l}p_w}\right)\rceil\lceil\log_{2^\beta}p_{l+1}\rceil\Bigg)\cdot m_{jk+ir+l}\\&+\frac{1}{2}\lceil\log_{2^\beta}p_s'\rceil\lceil\log_{2^\beta}p_{sr+1}\rceil\cdot m_{jk+sr+1}\\&+\sum_{l=sr+1}^{K-1}\lceil\log_{2^\beta}\left(p_s'\sqrt{\prod_{w=sr+1}^{l}p_w}\right)\rceil\lceil\log_{2^\beta}p_{l+1}\rceil\cdot m_{jk+sr+l}.\end{aligned}$$

Again, as β will be ≥ 32 and $K\ll 202378196\leq\pi(2^\beta)$, we may take $\lceil\log_{2^\beta}p_i\rceil=1$. So $F(K)$ can be approximated by

$$\begin{aligned}&\frac{1}{2}\left(\sum_{i=0}^{s-1}(p_i')_{(\beta)}+\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}\lceil\log_{2^\beta}\left(p_i'\frac{P_j}{P_{ir}}\right)\rceil+\sum_{j=sr}^{K-1}\lceil\log_{2^\beta}\left(p_i'\frac{P_j}{P_{sr}}\right)\rceil\right)\approx\\&\frac{1}{2}\left(\sum_{i=0}^{s-1}r(p_i')_{(\beta)}+\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}(\log_{2^\beta}P_j-\log_{2^\beta}P_{ir})+\right.\\&\qquad\left.(p_s')_{(\beta)}+\sum_{j=sr}^{K-1}(p_s')_{(\beta)}+\log_{2^\beta}P_j-\log_{2^\beta}P_{sr}\right)=\\&\frac{1}{2}\left(\sum_{i=0}^{s-1}r(p_i')_{(\beta)}+\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}\left(\log_{2^\beta}\sqrt{\prod_{k=1}^{j}p_k}-\log_{2^\beta}\sqrt{\prod_{k=1}^{ir}p_k}\right)+\right.\\&\qquad\left.(p_s')_{(\beta)}+\sum_{j=sr}^{K-1}(p_s')_{(\beta)}+\log_{2^\beta}\sqrt{\prod_{k=1}^{j}p_k}-\log_{2^\beta}\sqrt{\prod_{k=1}^{sr}p_k}\right)=\\&\frac{1}{2}\left(\sum_{i=0}^{s-1}r(p_i')_{(\beta)}+\frac{1}{2}\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}\left(\sum_{k=1}^{j}\log_{2^\beta}p_k-\sum_{k=1}^{ir}\log_{2^\beta}p_k\right)+\right)\\&\qquad\left.(p_s')_{(\beta)}+\sum_{j=sr}^{K-1}(p_s')_{(\beta)}+\frac{1}{2}\left(\sum_{k=1}^{j}\log_{2^\beta}p_k-\sum_{k=1}^{sr}\log_{2^\beta}p_k\right)\right)=\\&\frac{1}{2}\left(\sum_{i=0}^{s-1}r(p_i')_{(\beta)}+\frac{1}{2}\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}\left(\frac{\theta(p_j)}{\beta\log 2}-\frac{\theta(p_{ir})}{\beta\log 2}\right)+\right.\end{aligned}$$

$$(p'_s)_{(\beta)} + \sum_{j=sr}^{K-1} (p'_s)_{(\beta)} + \frac{1}{2}\left(\frac{\theta(p_j)}{\beta\log 2} - \frac{\theta(p_{sr})}{\beta\log 2}\right)\Bigg) \approx \tag{5.12}$$

$$\frac{1}{2}\left(\sum_{i=0}^{s-1} r(p'_i)_{(\beta)} + \frac{1}{2}\sum_{i=0}^{s-1}\sum_{j=ir+1}^{(i+1)r-1}\left(\frac{p_j}{\beta\log 2} - \frac{p_{ir}}{\beta\log 2}\right) + \right.$$

$$\left.(p'_s)_{(\beta)} + \sum_{j=sr}^{K-1}\left((p'_s)_{(\beta)} + \frac{1}{2}\left(\frac{p_j}{\beta\log 2} - \frac{p_{sr}}{\beta\log 2}\right)\right)\right) =$$

$$\frac{1}{2}\sum_{i=0}^{s-1} r\left((p'_i)_{(\beta)} - \frac{1}{2}\frac{p_{ir}}{\beta\log 2}\right) + \frac{1}{2}(K-sr)\left((p'_s)_{(\beta)} - \frac{1}{2}\frac{p_{sr}}{\beta\log 2}\right) + \frac{1}{4}\sum_{i=1}^{K-1}\frac{p_i}{\beta\log 2}. \tag{5.13}$$

Unfortunately we cannot simplify $F(K)$ further using claim 5.1.1. We will show this in Appendix D.1.2.

Similarly, we can simplify the second term of (5.10) to

$$\sum_{i=0}^{s-1}\left((p'_i)_{(\beta)} + \frac{1}{2}\frac{p_{(i+1)r} - p_{ir}}{\beta\log 2} - n_{(\beta)}\right)(n_{(\beta)} + 3).$$

Hence the altered PoP requires

$$F(K) + \textstyle\sum_{i=0}^{s-1}\left(\underbrace{(p'_i)_{(\beta)} + \frac{1}{2}\frac{p_{(i+1)r} - p_{ir}}{\beta\log 2}}_{t_{(\beta)}} - n_{(\beta)}\right)(n_{(\beta)} + 3) \tag{5.14}$$

single precision multiplications and

$$\frac{1}{2}\sum_{i=0}^{s-1}\left((p'_i)_{(\beta)} + \frac{1}{2}\frac{p_{(i+1)r} - p_{ir}}{\beta\log 2} - n_{(\beta)}\right)$$

single precision divisions.

To approximate Mul, note that as the length of the output of PoP is changed, the number of single precision multiplications that Mul requires is changed. As we cannot say anything yet about the length of t we are unable to make a good approximation for the number of single precision multiplications that are required by Mul. Recall that it equals

$$t_{(\beta)}n_{(\beta)} + t_{(\beta)}(n_{(\beta)} + 3) \tag{5.15}$$

single precision multiplications and $t_{(\beta)}$ single precision divisions.

Reduction when $t > n$

We will now discuss how the number of single precision multiplications and divisions behave when `condition` is simply replaced by $t >_{\text{gmp}} n$ in Pseudo code 5.1.2. The following remark shows that once $t > n$ in this implementation, with high probability, a mod n reduction will be performed after every multiplication of t by a small prime.

Remark 5.1.3. Note that it is reasonable to assume that the result of a reduction mod n returns a value (say X) which is uniformly in $\mathbb{Z}_n$. So the probability that $X_{(1)} < n_{(1)} - 1$ is less than $1/2$. Formally, let I denote the set of values less than n of bit-length $n_{(1)}$. The probability that $X = x$ is then given by $\mathbb{P}(X = x) = 1/(2^{n_{(1)}-1} + 2^{|I|})$. Thus, as there are $2^{|I|}$ possible values that $x_{(1)} = n_{(1)}$, the probability that $X_{(1)} = n_{(1)}$ equals $2^{|I|}/(2^{n_{(1)}-1} + 2^{|I|})$. So, similarly, the probability that $x_{(1)} < n_{(1)} - 1$ equals

$$\frac{2^{n_{(1)}-2}}{2^{(n_{(1)}-1+|I|)}} < \frac{2^{n_{(1)}-2}}{2^{(n_{(1)}-1)}} < \frac{1}{2}.$$

More general for $0 < y < n_{(1)}$ the probability that $X_{(1)} < y$ is given by

$$\mathbb{P}(X_{(1)} < y) = \frac{2^{y-1}}{2^{(n_{(1)}-1+|I|)}} < \frac{2^{y-1}}{2^{(n_{(1)}-1)}} = 2^{y-n_{(1)}}.$$

As the proof of Claim 2.2.4 shows, it holds that $p_k \sim \log n$. So $k \log k \sim \log n$. Thus, if $K = 2k$ then by Equation 5.7 it follows that

$$t_{(\beta)} \approx 1/2 \cdot \frac{2k \log 2k}{\beta \log 2} \approx \frac{k \log k}{\beta \log 2} \approx n_{(16)}.$$

Therefore, approximately, extra mod n reductions are applied by Pseudo code 5.1.2 if $K \geq 2k$. It follows by Remark 5.1.3 that if $K \geq 2k$, then, since $m_i = 1$ with probability $1/2$, with a high probability we need to perform approximately $1/2(K - 2k)$ mod n reductions. $F(K)$ is then approximated similarly to Equation (5.5) by

$$\frac{1}{4} \sum_{i=1}^{\min\{K,\ 2k\}} \frac{p_i}{\beta \log 2} + \frac{1}{2} \left(\max\{K - 2k, 0\}\right) \cdot n_{(\beta)},$$

so that PoP requires approximately

$$F(K) + \frac{1}{2} \left(\max\left\{K - 2k, 0\right\}\right) \left(n_{(\beta)} + 3\right) \tag{5.16}$$

single precision multiplications and $1/2 \max\left\{K - 2k, 0\right\}$ single precision divisions. Because when $K \geq 2k$, then $t_{(\beta)} \approx n_{(\beta)}$, it holds that Mul requires approximately

$$\left(\min\{\frac{K \log K}{2\beta \log 2}, 1\} \cdot n_{(\beta)}\right) \cdot n_{(\beta)} + \left(\min\{\frac{K \log K}{2\beta \log 2}, 1\} \cdot n_{(\beta)}\right) \left(n_{(\beta)} + 3\right) \tag{5.17}$$

single precision multiplications and $\min\{K/2k, 1\}$ $n_{(\beta)}$ single precision divisions.

Figure 5.2 shows how the single precision calculations behave with respect to Pseudo code 5.1.2 under these approximations using again $\beta = 32$ and a 1234-bit modulus n, for which $k = 153$ on a one Megabyte message m.

According to Figure 5.3, the extra mod n reductions are advantageous –if one considers SP-multiplications– in the sense that the cost increases not so fast after the optimum as it does without the extra mod n reductions. On the other hand, the optimum remains exactly the same. The next subsection tries to improve this Pseudo code.

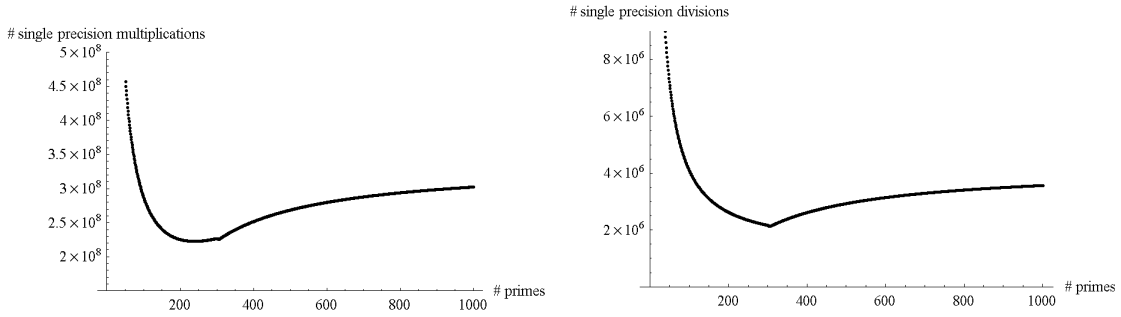

Figure 5.2: Approximate number of Single precision multiplications on the left and divisions on the right versus K; apply a mod n reduction after each time $t > n$.

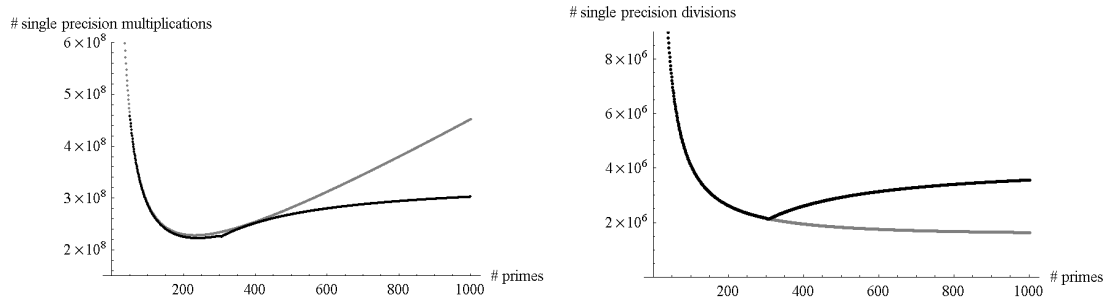

Figure 5.3: Gray: Figure 5.1, Black: Figure 5.2

Improved Reduction when $t > n$

Again, we will replace `condition` by $t >_{\text{gmp}} n$, but instead of evaluating $t = t \mod_{\text{gmp}} n$, we will store the mod n reduced value of t in an array, set $t = 1$ and continue. Pseudo code 5.1.4 makes this precise.

As the size of the primes increase logarithmically it holds that the numbers of multiplications between two mod n reductions are more or less equal. So we simply assume that equation (5.13) may be applied to approximate the number of SP-multiplications, where $r = 2k$. Because we set $t = 1$ after a mod n reduction, it follows that p_i' can be neglected for all i in Equation (5.13). Moreover, since for every prime p_i for $i < 202378196$ it holds that $(p_i)_{(\beta)} = 1$ and as soon as $t > n$ then $t_{(\beta)} \leq n_{(\beta)} + 1$. Therefore, $F(K)$ is approximately

$$\frac{1}{4}\left(\sum_{i=1}^{K-1}\frac{p_i}{\beta\log 2} - \sum_{i=0}^{s-1} r\left(\frac{p_{ir}}{\beta\log 2}\right) - (K-sr)\left(\frac{p_{sr}}{\beta\log 2}\right)\right) \tag{5.18}$$

and Equation (5.14) implies that PoP, without the multiplications required by multiplying the elements of A, then requires

$$\text{PoP}_2(K) = F(K) + \left\lfloor\frac{K}{2k}\right\rfloor (n_{(\beta)} + 3)$$

single precision multiplications and $\lfloor\frac{K}{2k}\rfloor$ single precision divisions. Because $w \approx \lceil\frac{K}{2k}\rceil$ and

Pseudo-code 5.1.4. Iteration of Classic VSH (extra mod n improved)

Input: The iteration number, j, x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$, $w = 1$.
Let A be an integer array, containing multiple precision integers.
Product of Primes (PoP):

```
for(i = 1; i ≤ k; i++){
    if(m_{j·k+i} == 1){
      t = t *_gmp p_i;
      if(condition) {
        A[w++] = t  mod_gmp  n;
        t = 1;
      }
    }
}
for(i = 1; i < w; i++) {
    t = t *_gmp A[i]  mod_gmp  n;
}
```

Squaring (Sq):

$x_j = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n;$

Multiplication (Mul):

$x_{j+1} = (x_j *_{\text{gmp}} t) \mod_{\text{gmp}} n;$

Return x_{j+1}.

the elements of A have size $n_{(\beta)}$, it follows that the full PoP requires approximately

$$\text{PoP}_2(K) + \left\lfloor \frac{K}{2k} \right\rfloor \cdot \left(n_{(\beta)}^2 + n_{(\beta)}(n_{(\beta)} + 3) \right) \tag{5.19}$$

single precision multiplications and

$$\lfloor \frac{K}{2k} \rfloor \left(1 + n_{(\beta)}\right)$$

single precision divisions.

If $K < 2k$ then Mul is approximated by Equation (5.8), but if $K > 2k$ then $t_{(\beta)} \approx n_{(\beta)}$, so that Mul requires

$$n_{(\beta)}^2 + n_{(\beta)}(n_{(\beta)} + 3) \tag{5.20}$$

single precision multiplications and $n_{(\beta)}$ single precision divisions.

Figure 5.4 shows how the number of SP-multiplications and SP-divisions behave with respect to Pseudo code 5.1.4. Again, $\beta = 32$, the modulus has size 1234 and $k = 153$. We let K run from 1 to 1000.

It follows that applying extra mod n reductions in a clever way improves the optimum: instead of reducing the amount of the SP-multiplications by 5.99% the number can be

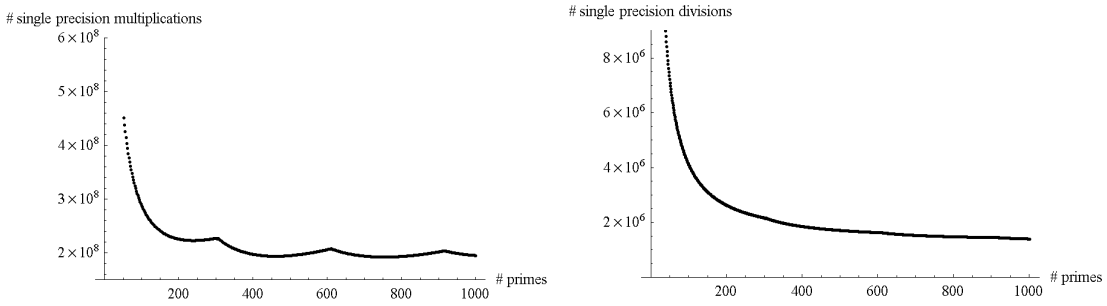

Figure 5.4: Approximate number of Single precision multiplications on the left and divisions on the right versus K; apply a mod n reduction after each time $t > n$ (improved)

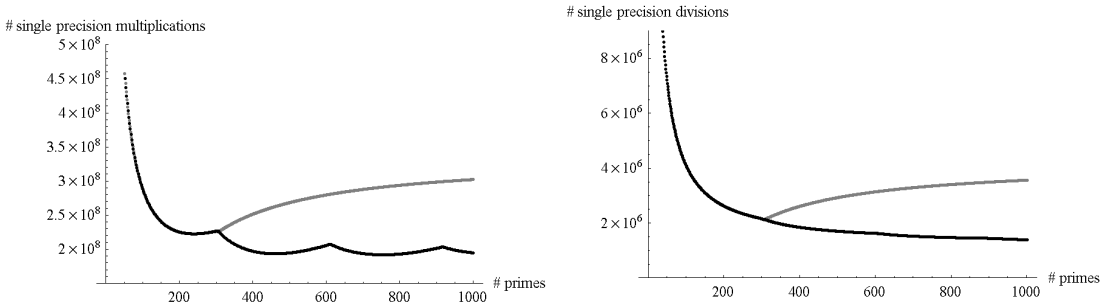

Figure 5.5: Gray: Figure 5.2, Black: Figure 5.4

reduced by 19.9%. However, for these K, the asymptotic behavior is not visible and it seems to increase instead of decrease. We will show why this is indeed the case.

Observe that the assumption that PoP, Mul and Sq require the same amount of time each iteration of the introduction is not very precise. In fact, using $K = 2k = 306$ and a 1234-bit n, we find that if $\beta = 32$, then, according to our model, PoP, Mul and Sq require 50%, 29% and 21% of the total amount of SP-multiplications per iteration respectively. If we argue similar as in the introduction, we would conclude that we could reduce the number of SP-multiplications by 21%. On the other hand, consider Pseudo code 5.1.4. Each time, when a mod n reduction is performed, the number of SP-multiplications needed to calculate the next r small primes increases with respect to the previous calculation of r primes because the values of the primes are increasing. Taking that into account, it may hold that the extra number of SP-multiplications needed to derive the product of the next r primes outweighs the number of SP-multiplications gained by reducing the iteration numbers. For example, if $K = 2000k$ and $\beta = 32$, then,

$$\frac{\mathrm{PoP}_{2000k} + \mathrm{Sq} + \mathrm{Mul}}{1000(\mathrm{PoP}_{2k} + \mathrm{Sq} + \mathrm{Mul})} \approx 1.49,$$

implying that Classic-VSH applied with $K = 2000k$ requires 50% more SP-multiplications than Classic-VSH applied with $K = 2k$. Figure 5.6 shows the behavior of the number of SP-multiplications of the same version of Classic-VSH, when K runs to 10000. In conclusion, the number of SP-multiplications increases for this Pseudo code, because PoP

becomes dominant and requires more extra SP-multiplications than the amount reduced by reducing the iteration numbers.

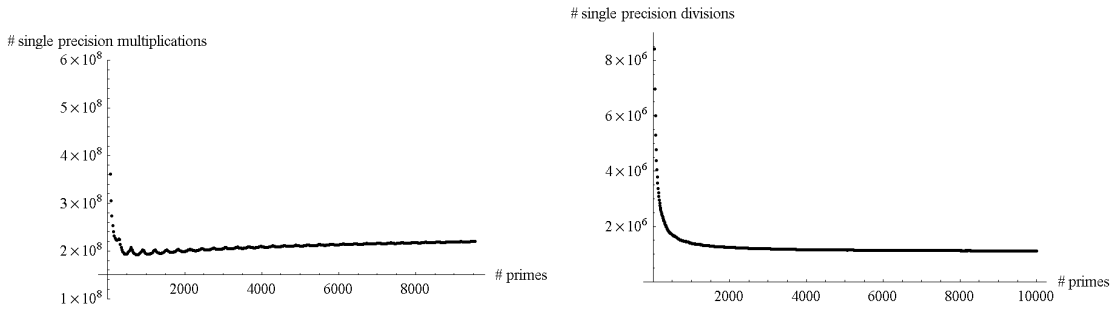

Figure 5.6: Approximate number of Single precision calculations versus K, apply a mod n reduction after each time $t > n$ (improved)

The last subsection shows that although the addition of small primes doesn't decrease the number of SP-multiplications a lot, still, we get better results due to the use of GMP.

5.1.2 Fast-VSH

This subsection discusses the behavior of the amount of SP-multiplications and SP-divisions, when K is taken larger than originally for Fast-VSH.

Consider Pseudo code 4.0.3 and observe that it is almost similar to Pseudo code 5.1.4. The only difference is that Fast-VSH derives exactly K multiplications in PoP, where for each consecutive primes p_i, p_j it holds that $i - j \geq 2^b$.

Pseudo code 4.0.3. Iteration of Fast-VSH (straightforward)

Input: The iteration number, j, x_j and message m.

Output: x_{j+1}.

Procedure: Set $t = 1$.
Squaring (Sq):
$x_{j+1} = (x_j *_{\text{gmp}} x_j) \mod_{\text{gmp}} n;$
Product of Primes (PoP):
for$(i = 1; i \leq k; i{+}{+})\{$
$t = t *_{\text{gmp}} p_{(i-1)2^b + m[jk+i]+1};$
if$(t >_{\text{gmp}} n)$ {
$x_{j+1} = x_{j+1} * t \mod_{\text{gmp}} n;$
$t = 1;$
}
}
Multiplication (Mul):
$x_{j+1} = (x_{j+1} *_{\text{gmp}} t) \mod_{\text{gmp}} n;$
Return x_{j+1}.

Because of the gaps between the successive primes, the corresponding model is much more complex, because we cannot use the approximations of previous subsections. Also, because of the similarity between the implementation of Fast-VSH and Pseudo code 5.1.4, we will omit such approximation here and assume that the corresponding model is also similar.

It follows that the figure of the behavior of the number of SP-calculations for Fast-VSH is very similar to Figure 5.4. Because of the gaps between the small primes it follows that the product of small primes (PoP) becomes dominant faster than it comes dominant with respect to Classic-VSH. So, asymptotically, the number of SP-multiplications required by Fast-VSH based on $K * 2^b$ small primes will increase faster than suggested by Figure 5.4.

The next subsection will show the actual speed on our system when K is increased.

5.1.3 Results

This section presents the results we get by allowing more small primes in our implementations on our system. First of all, the results are presented in figures, where the speed of VSH is plotted against the amount of small primes.

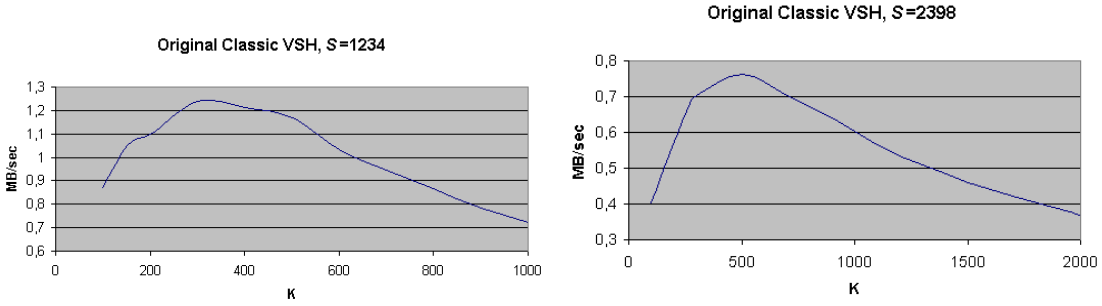

Figure 5.7: The speed of the original Classic VSH, where on the left figure $S = 1234$ and $S = 2398$ on the right figure.

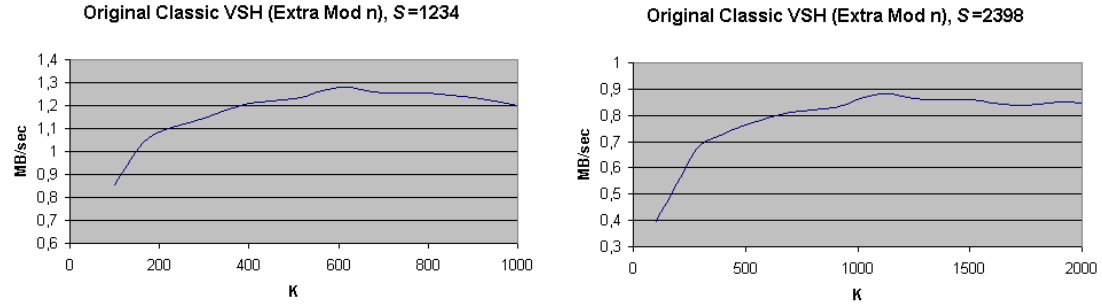

Figure 5.8: The speed of the original Classic VSH with extra mod n reductions, where on the left figure $S = 1234$ and $S = 2398$ on the right figure.

Because GMP is designed to use the computer processor as efficient as possible, we assume that it is able to perform SP-calculations, where $\beta = 32$, because our Pentium IV processor is 32-bit. As our model suggests, the speed does not increase asymptotically. On

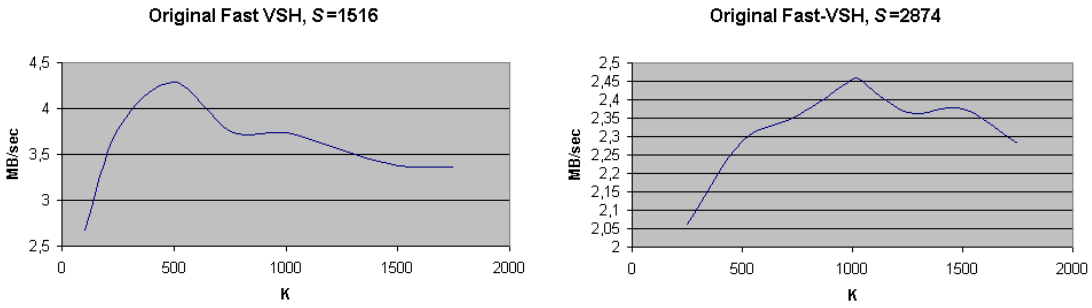

Figure 5.9: The speed of the original Fast VSH, where $b = 8$ and on the left figure $S = 1516$. $S = 2874$ on the right figure.

the other hand our approximations show that the increment of the speed for Classic-VSH, using $S = 1234$ and applying extra mod n reductions, for example, should be at most about $100/(1 - 0.045) - 100 = 4.7\%$, while we observe that the speed increases by about 26%, which is much more. Also, the maximum speed is measured with a larger K than Figure 5.4 suggests. Because our approximations are mostly based on $\sim$ we expect that for large K our approximations are accurate. For small K, the difference between the results and approximations may be due to relatively big errors in the approximations. But there is more. Our approximations do not take "all the usual assembler tricks and obscurities for speed" (see [14]) into account. The following Example shows that this can lead to a big difference between a correct model and the real performance.

Example 5.1.5. Consider the 32-bit based multiple precision reduction mod X of the integers Y and Z, using the classical reduction algorithm, where $X_{(32)} = \alpha$. If $Y_{(32)} = \alpha + 1$, then the reduction Y mod X requires $\alpha + 3$ SP-multiplications. And if $Z_{(32)} = \alpha + \gamma$ then the reduction Z mod X requires $\gamma(\alpha + 3)$ SP-multiplications. So evaluating γ times the reduction Y mod X costs an equal amount of SP-multiplications as evaluating Z mod X once. Therefore, we would conclude that evaluating γ times the reduction Y mod X takes the same amount of time as evaluating Z mod X once. In other words, the time that it takes to evaluate Z mod X divided by γ should be equal to the time that it takes to evaluate Y mod X.
The following table shows the time that GMP requires to evaluate Y mod X 2^{32} times. We take $X_{(32)} = n_{(32)} = 39$.
Note that the time GMP takes to evaluate Y mod X, where $Y_{(32)} - X_{(32)} = 30$, divided by

$Y_{(32)} - \alpha$	Time (seconds)	Time divided by $(Y_{(32)} - \alpha)$
1	2.71	2.71
5	7.1	1.42
10	11.53	1.15
20	21.2	1.06
30	29.59	0.99

30 is about one third of the time GMP takes to evaluate Y mod X, where $Y_{(32)} - X_{(32)} = 1$.

It follows that GMP cannot be modeled by means of calculating the number of

SP-calculations only. So Example 5.1.5 illustrates the remark on top of page 12 of [6], which says that the best value for K is "best determined experimentally". Thus, the best value for K is given by Figure 5.8 for Classic VSH and by Figure 5.9 for Fast VSH if $b = 8$. Table 5.2 summarizes the results.

S'	S	VSH-variant	Optimal K	speed (MB/sec)	speed-up
1024	1234	Classic-VSH	600	1.279	26%
	1516	Fast-VSH	500	4.290	14%
2048	2398	Classic-VSH	1100	0.880	33%
	2874	Fast-VSH	1024	2.457	0%

Table 5.2: The best values for K for Classic-VSH and Fast-VSH and their speed-ups with respect to the original choice of K.

We conclude that with our model, we are able to show that if $\beta = 32$ that the performance of VSH will be less than optimal if K is very large due to the increment of the SP-multiplications by PoP. We note that because the number of SP-multiplications is much bigger than the number of SP-divisions, the gain of the decrement of SP-divisions is outweighed by the increment of SP-multiplications. Also, because GMP uses the maximal performance of the CPU at some stage, we conclude that our model, at some stage, can be applied to VSH if K is large. Figure 5.10 supports this.

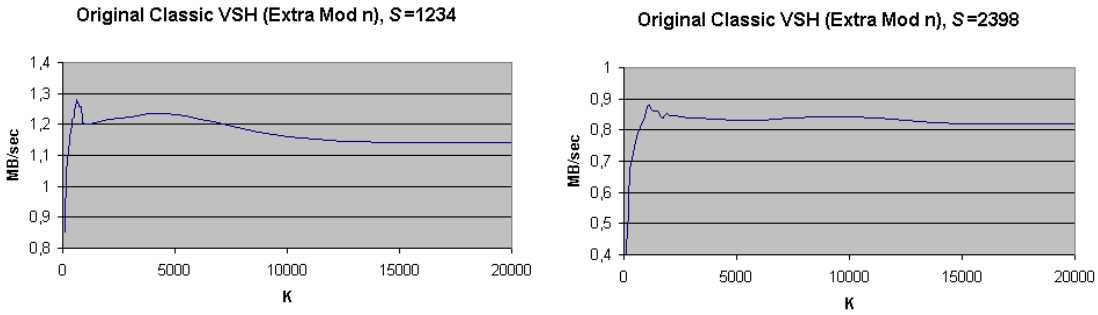

Figure 5.10: The speed of the original Classic VSH with extra mod n reductions, where on the left figure $S = 1234$ and $S = 2398$ on the right figure.

The next section will discuss the new Computational VSSR assumption and show how this improves the speed of VSH.

5.2 The new Computational VSSR assumption

This section shows that with the current state of the art of integer factorization, the original Computational VSSR Assumption is very pessimistic. Recall from Subsection 2.1.3 that originally it is assumed that solving VSSR based on u small primes and a modulus of bit-size S is at least as hard as factoring an S'-bit integer for which holds that

$$L(S') \geq \frac{L(S)}{u}.$$

Here, $L(S)$ denotes the time that the most efficient factoring algorithm, the Number Field Sieve, takes to factor an S-bit composite. So, formally, under this assumption the idea of adding primes without increasing the size of the modulus from previous subsection would affect the security of VSH. However, the original VSH paper remarks that this assumption is "certainly overly conservative". In this section we will demonstrate why it is overly conservative while introducing a *new* Computational VSSR assumption that is based upon the present state of knowledge on the best algorithm for solving VSSR. We will then show that the new assumption allows us to use $S' = S$ as well as more small primes in the hash function. Consequently, the idea of adding primes from the previous can be done without any degradation of security.

There are two approaches to trying to solve VSSR: (i) first factor the modulus and then VSSR can be solved trivially, or (ii) try to solve VSSR directly without factoring. Observe that the second approach depends upon the number of small primes that are used while the first approach does not. If the best known algorithm for the second approach takes at least as much time as the best known algorithm for the first approach, then the second approach is of no concern to us. This observation is the idea behind the new Computational VSSR Assumption.

There has been many years of research on integer factorization for finding smooth modular square roost (i.e. solutions to VSSR) since such algorithms are heuristically converted into factorization algorithms. The best algorithm known to date to solve VSSR –without factoring– is roughly given by Algorithm 5.2.1. This algorithm is the basis for well studied factorization algorithms such as Quadratic Sieve and Continued Fraction Integer Factorization algorithms. The idea is to try to generate residues with known square-root that are "small", and hope that the result is smooth. The latter part is left to chance: one simply tries to factor the result into small primes, keeping it if they succeed and discarding it when they do not. It follows from analytic number theory that if we assume that the residues are randomly distributed then it follows that the smaller we can generate the residues with known modular square root, the more likely they are smooth. The Quadratic Sieve and the Continued Fraction Integer Factorization algorithms generate such residues of size $\sqrt{n}$ in polynomial time, which is nowadays the best researchers are able to. Observe that if one can find such residues of size less than $\sqrt{n}$ in polynomial time, then one improves the efficiency of Algorithm 5.2.1. It can be shown that if one can do better than finding a residue of order $\sqrt{n}$ (with known square root), then one can improve the running time of Quadratic Sieve. The argument is from [5], which we will sketch in Appendix D.2. Observe that in general –because Algorithm 5.2.1 is the basis for Quadratic Sieve– if there exists a more efficient algorithm than Algorithm 5.2.1 to solve VSSR, then one improves on the speed of the Quadratic Sieve.

So if it is easy to find x such that $y \equiv x^2 \bmod n$ has least positive representative of order $\sqrt{n}$ and is p_u-smooth (see Definition 2.1.1), then solving VSSR may be easy. Therefore, we want that

$$\mathcal{P}(B) = Pr\left[y \text{ is } B-\text{smooth}|\ y \text{ is about } \sqrt{n}\right]$$

is small for $B = p_u$. In fact, we want that finding an x such that $y \equiv x^2 \bmod n$ has least positive representative of order $\sqrt{n}$ and is p_u-smooth takes at least as much time as it takes to factor n via the Number Field Sieve. So, conservatively assuming that finding an x such that $y \equiv x^2 \bmod n$ has least positive representative of order $\sqrt{n}$ takes $O(1)$ time, we want to find u such that $L(S) \geq 1/\mathcal{P}(p_u)$.

Algorithm 5.2.1. Solve VSSR

Input: A hard to factor modulus n and a fixed constant u.

Output: $x^2 \equiv \prod_{i=0}^{u} p_i^{e_i} \bmod n$, where at least one e_i is odd.

Procedure: Define condition C as $y \equiv \prod_{i=0}^{u} p_i^{e_i} \bmod n$, where at least one e_i is odd.
Initially, set C :=false.
Loop:
While C is false do:
Find x such that $y \equiv x^2 \bmod n$ has least positive representative modulo n of order $\sqrt{n}$.
Try to factor the residue of y modulo n into small primes.
If it factors, then set C =true.

Return $(x, (e_i)_{i=0}^{u})$.

Consider VSSR (Definition 2.1.7) with u small primes and an S-bit modulus. Our new Computational VSSR Assumption is given by

New Computational VSSR Assumption: *Solving VSSR based on u small primes and an S-bit modulus is as hard factoring an S'-bit number, where S' the smallest value such that*

$$L(S') \geq 1/\mathcal{P}(p_u).$$

The following lemma shows how the running time of Algorithm 5.2.1 can be (conservatively) approximated.

Lemma 5.2.2. *Let n be an integer and $U = \frac{\log \sqrt{n}}{\log B_n}$, where B_n denotes the smoothness bound. Asymptotically, Algorithm 5.2.1 has running time T, which is well estimated by*

$$U^{U(1+o(1))}. \tag{5.21}$$

Proof. Consider VSSR, where p_u is the largest prime smaller than B_n. The expected amount of loops in Algorithm 5.2.1 is given by $1/\mathcal{P}(B_n)$. If we conservatively assume that each loop runs in $O(1)$ time, then $1/\mathcal{P}(B_n)$ is the expected running time of Algorithm 5.2.1.

Next, we will show that $1/\mathcal{P}(B_n)$ is well estimated by $U^{U(1+o(1))}$.

Firstly, we will derive a lower bound for the maximal value of B_n such that $U^{U(1+o(1))} = L(S)$. Then, we will show that this lower bound satisfies the conditions for the validness of the estimate in Equation (5.21). According to Subsection 1.4.5 of [9] the probability $\mathcal{P}(B_n)$ is well estimated by $U^{-U(1+o(1))}$, if

- (i) $\sqrt{n}$ is large, and
- (ii) $(\log \sqrt{n})^{1+\epsilon} < B_n < \sqrt{n}$.

It follows from Section 6.2 of [9] that the running time for the Number Field Sieve to factor n is given by $\exp\big((c + o(1))(\log n)^{1/3}(\log \log n)^{2/3}\big)$, where $c = (64/9)^{1/3}$. So, $L(S) = 1/\mathcal{P}(B_n)$, where the $o(1)$'s are dropped, is equivalent to

$$U^U \approx e^{c(\log n)^{\frac{1}{3}}(\log \log n)^{\frac{2}{3}}}.$$

We find that if $\log U > 1$ the maximal value for B_n is at least:

$$\begin{aligned}
U \log U &= c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}} && \Leftrightarrow \\
U &= \frac{c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}{\log U} && (\log U > 1) \Rightarrow \\
U &< c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}} && \Leftrightarrow \\
\frac{\log\sqrt{n}}{\log B_n} &< c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}} && \Leftrightarrow \\
\log B_n &> \frac{1}{2}\cdot\frac{\log n}{c(\log\log n)^{\frac{1}{3}}(\log n)^{\frac{2}{3}}} && \Leftrightarrow \\
\log B_n &> \frac{1}{2c}\left(\frac{\log n}{\log\log n}\right)^{\frac{2}{3}} && \Leftrightarrow \\
B_n &> \exp\frac{1}{2c}\left(\frac{\log n}{\log\log n}\right)^{\frac{2}{3}}.
\end{aligned}$$

Define

$$H(n) := \exp\frac{1}{2c}\left(\frac{\log n}{\log\log n}\right)^{\frac{2}{3}}.$$

From the condition $\log U > 1$, it follows that

$$\log U > 1 \Leftrightarrow \frac{\log\sqrt{n}}{\log B_n} > e \Leftrightarrow B_n < \exp\frac{\log n}{2e}.$$

Using $B_n = H(n)$ yields

$$\begin{aligned}
\exp\frac{1}{2c}\left(\frac{\log n}{\log\log n}\right)^{\frac{2}{3}} &< \exp\frac{\log n}{2e} && \Leftrightarrow \\
(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}} &> \frac{e}{c},
\end{aligned}$$

which is true for $n \geq 16$.

We will now show that the conditions (i) and (ii) are satisfied. Firstly, note that (i) is true by assumption. Next, consider $(\log\sqrt{n})^{1+\epsilon} < B_n < \sqrt{n}$. The upper-bound is obvious since

$$\sqrt{n} = e^{\frac{1}{2}\log n} > e^{\frac{1}{2c}\left(\frac{\log n}{\log\log n}\right)^{2/3}}.$$

The lower bound is also obvious for n large enough since $H(n)$ is sub-exponential –i.e. hyper polynomial– in $\log n$, whereas the term $(\log\sqrt{n})^{1+\epsilon}$ is polynomial in $\log n$.

It follows that, if n is large enough, $(\log\sqrt{n})^{1+\epsilon} < B_n < \sqrt{n}$, so that Equation 5.21 is a good estimate for the expected running time of Algorithm 5.2.1. □

Obviously, as VSH can be broken either by solving VSSR or by factoring its modulus n, it holds that $S \geq S'$ if VSH has to be as secure as factoring an S'-bit modulus. Our New Computational VSSR Assumption implies that $S = S'$ if u is taken such that

$$L(S) \geq \frac{1}{\mathcal{P}_u}.$$

Lemma 5.2.2 then implies –assuming n is large enough– that u has to be taken such that p_u is not larger than the value implied by

$$U^{U(1+o(1))} = L(S),$$

where $U = \frac{\log\sqrt{n}}{\log p_u}$ and $L(S)$ denotes the running time of the Number Field Sieve to factor n. So this equation is equivalent to

$$U^{U(1+o(1))} = e^{(c+o(1))(\log n)^{1/3}(\log\log n)^{2/3}}, \tag{5.22}$$

where $c = (64/9)^{1/3}$.

Because for both $S' = 1024$ and $S' = 2048$ it holds that $H(2^{S'}) > (\log\sqrt{n})^{1+\epsilon}$, where we take $\epsilon = 0.001$, where $H(n)$ is defined in the proof of Lemma 5.2.2, we assume that Equation (5.22) is valid. Solving Equation (5.22) by dropping the $o(1)$'s and using Newton's method for finding roots (see Appendix E.1.3), yields $S = S' = 1024$ for u at most $\approx 2^{21}$. And for $S' = 2048$ it holds that $S = S'$ for u at most $\approx 2^{35}$. It follows for the Original versions of Classic-VSH and Fast-VSH, that for the the original choices of K, we may take $S = S'$. This leads to the speed-ups for VSH as presented in Table 5.3.

S	S'	VSH-Variant	K	# primes	Speed (MB/sec)	Speed-up
1024	1234	Classic-VSH	153	153	1.055	0 %
	1024		131	131	1.232	17 %
	1516	Fast-VSH	256	2^{16}	3.770	0 %
	1024		256	2^{16}	5.100	35 %
2048	2398	Classic-VSH	272	272	0.681	0 %
	2048		233	233	0.770	13 %
	2874	Fast-VSH	1024	2^{18}	2.457	0 %
	1024		1024	2^{18}	3.356	37 %

Table 5.3: The speed-up for Classic and Fast VSH under the new VSSR assumption

5.3 Combining all Ideas

This section will present the results if we combine the ideas discussed in this chapter and previous Chapter. We note that the Computation VSSR Assumption from previous section imply that Classic-VSH is secure if $S = S' = 1024$ for $K \leq 2^{21}$ and if $S = S' = 2048$ for $K \leq 2^{35}$. Fast-VSH is secure if $S = S' = 1024$ for $K \leq 2^{21-b}$ and if $S = S' = 2048$ for $K \leq 2^{35-b}$. Practically, it means for Classic-VSH and Fast-VSH, where $b = 8$, that we may take $S = S'$, for all K, without affecting the security of VSH.

This section provides figures in which the speed of a certain variant of VSH is put against K. Then, with respect to Classic-VSH and Fast-VSH, it is concluded which implementation using what K has the best performance and give the total speed-up. Chapter 8 provides a table with the best value for K for all Pseudo codes and their speed-ups. We will split the presentation in two subsections. The first subsection shows how much we can improve the speed of Classic-VSH. The second subsection shows how much we can improve the speed for Fast-VSH.

5.3.1 Classic-VSH

The following figures show the speed of the effective Pseudo codes of Chapter 4 against K, the number of bits processed each iteration. Due to the analysis in the first subsection of this chapter we alter the Pseudo codes so that extra mod n reductions are performed similarly to Pseudo code 5.1.4. See Appendix E.2 for the C-source codes of these altered Pseudo codes.

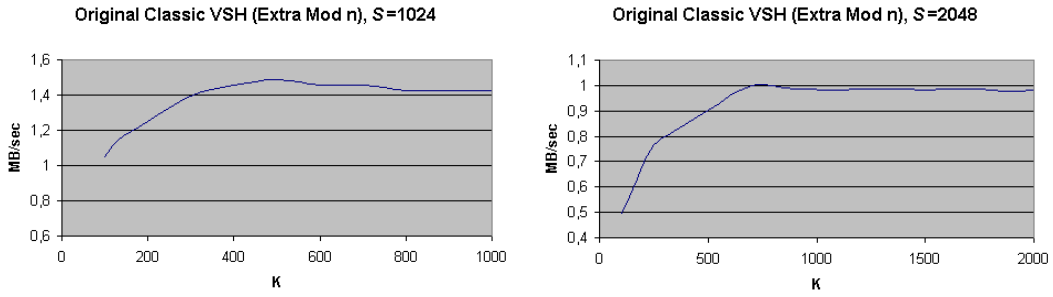

Figure 5.11: The speed of Original Classic VSH, where on the left figure $S = 1024$ and $S = 2048$ on the right figure.

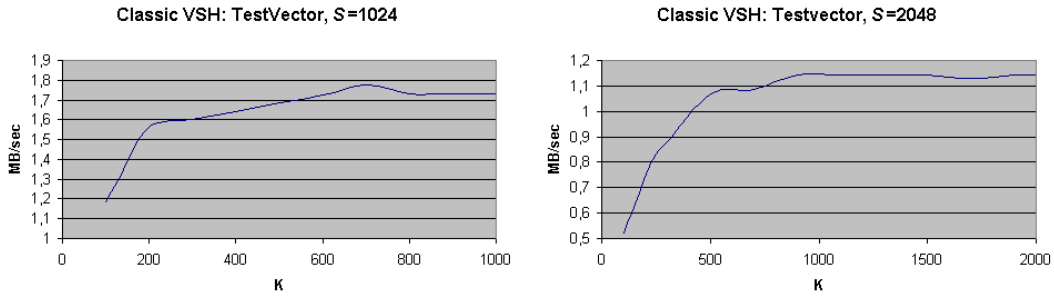

Figure 5.12: The speed of Classic VSH, based on Pseudo code 4.2.2, where on the left figure $S = 1024$ and $S = 2048$ on the right figure.

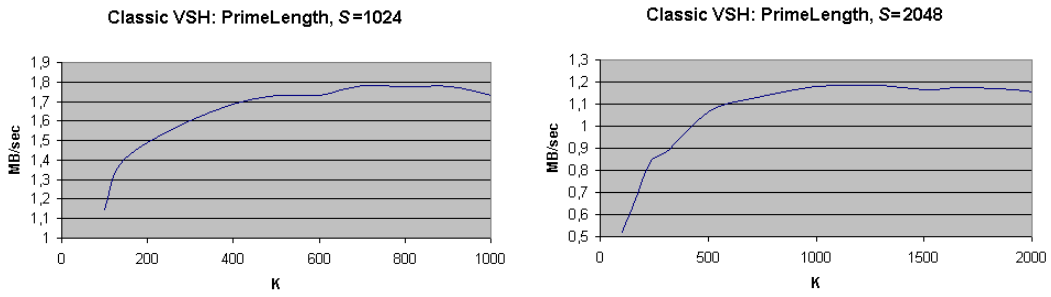

Figure 5.13: The speed of Classic VSH, based on Pseudo code 4.2.3, where on the left figure $S = 1024$ and $S = 2048$ on the right figure.

It follows that the best Pseudo code for Classic-VSH is Pseudo code 4.2.3, where extra mod n reduction are introduced similar to Pseudo code 5.1.4. If $S' = 1024$, then, the optimal value for $K = 700$. The speed is of Classic-VSH is then 1.780 MB/sec, which is 68.72% more than 1.055 MB/sec originally. If $S' = 2048$, then, the optimal value for $K = 1300$ and the speed of Classic-VSH is 1.186 MB/sec, which is 74.23% more than 0.6807 MB/sec originally.

5.3.2 Fast-VSH

The following Figure shows the speed of Fast-VSH against K, the number of small prime multiplications each iteration. Here again $b = 8$.

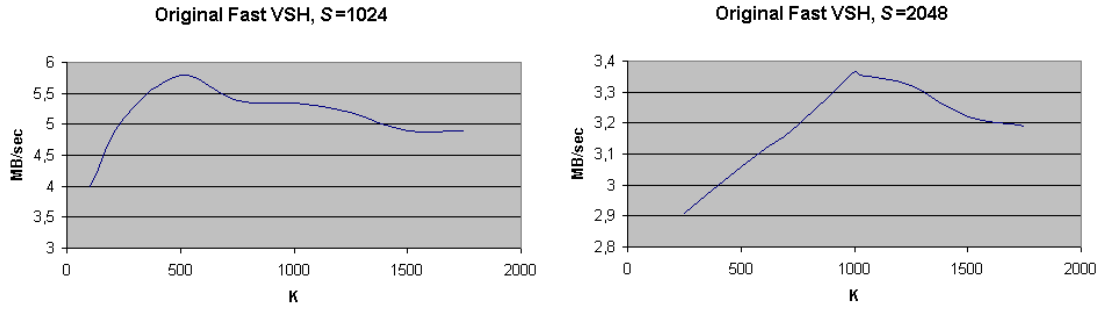

Figure 5.14: The speed of Original Fast-VSH, where on the left figure $S = 1024$ and $S = 2048$ on the right figure.

It follows that the best value for K in Fast-VSH is 500 if $S' = 1024$. The speed of Fast-VSH is then 5.800 MB/sec, which is 53.84% more than 3.770 MB/sec originally. If $S' = 2048$, then, the best value for K is 1000 and the speed of Fast-VSH is 3.365 MB/sec, which is 36.96% more than 2.457 MB/sec originally.

Chapter 6

Reducing Prime Exponents by $\phi(n)$

This chapter discusses an idea to replace the time-consuming iteration by a relatively cheap iteration. This version of VSH assume that each user generates their own hash function. This chapter will show that they can compute their hash efficiently using their Trapdoor information, but the other user has to use the relatively slow Classic-VSH to compute that hash. Therefore, this version of VSH is called *Trapdoor-VSH*. We will show that Trapdoor-VSH, using Euler's Phi function ($\phi(n)$), is about 3 times faster than Fast-VSH.

Obviously, from the discussion in Subsection 2.2.1 of Chapter 2, if $\phi(n)$ is known, collisions can be created easily. Thus, it immediately follows that Trapdoor-VSH may not be used by the ones for which the factorization of n should remain unknown. In fact, only the parties that generate n can apply Trapdoor-VSH.

Trapdoor-VSH can still be very advantageous, for example, if exactly one party has to use a collision resistant hash function many times and if he has access to the factorization of n. This party can benefit a lot by applying Trapdoor-VSH, while the other parties, who don't have to calculate hashes that many times, still use Classic-VSH.

For example, [6] shows how to apply VSH in the Cramer-Shoup Signature scheme. In this scheme VSH is turned into a *Randomized Trapdoor Hash Function* by replacing the initial value for x_0 by a random value and return $x_{\mathcal{L}+1}^2$ instead of $x_{\mathcal{L}+1}$. So this Randomized Trapdoor Hash Function has the same running time as VSH. In this application, the signer generates the modulus for VSH, so the signer can always apply the implementation of Trapdoor-VSH. So if a company has to sign many large messages, then, it can benefit from the security and efficiency that the Cramer-Shoup Signature Scheme offers, and can save more time than suggested in [6] by applying Trapdoor-VSH for the evaluation of the hashes of the messages.

According to [5] there are companies that require to sign many 1 MB or larger messages. This implies that Trapdoor-VSH is applicable in real world applications.

Consider again the algorithm of Classic-VSH (Algorithm 2.2.1 in Chapter 2), which we will repeat here.

Let $\phi(n)$ denote Euler's phi-function. By construction, the output hash of VSH is equivalent to

$$\prod_{i=1}^{K} p_i^{e_i} \mod n, \tag{6.1}$$

where $e_i = \sum_{j=0}^{\mathcal{L}} 2^{\mathcal{L}-j} m_{jK+i}$. As [6] remarks in section 5, the speed of VSH will increase if

Algorithm 2.2.1. Classic VSH

Input: An l-bit message m and an S-bit RSA modulus n.

Output: An S-bit hash of m.

Procedure: *Initialization:*
(Initial Value): Let $x_0 = 1$.
(Number of iterations): Let $\mathcal{L} = \lceil \frac{l}{K} \rceil$.
(padding): Let $m_i = 0$ for $l < i \leq \mathcal{L}K$.
(Appending length block): Let l_i be bits so that $l = \sum_{i=1}^{K} l_i 2^{i-1}$ and let $m_{\mathcal{L}K+i} = l_i$ for $i = 1, \cdots, K$.
Iteration:
For $j = 0, \cdots, \mathcal{L}$ compute
$x_{j+1} = x_j^2 \times \prod_{i=1}^{K} p_i^{m_{j \cdot K+i}} \mod n$.
Finalization:
Return $x_{\mathcal{L}+1}$.

one calculates $e_i' \equiv e_i \mod \phi(n)$ in each iteration, where e_i' denotes the least positive representant of $e_i \mod \phi(n)$, and, in the end, evaluates the product

$$\prod_{i=1}^{K} p_i^{e_i'} \mod n.$$

As we discussed in Chapter 2, each iteration of Classic-VSH takes $O((\log n)^2)$ time. Because, originally, there are $K = k = O((\log n)/\log\log n)$ message bits processed each iteration, the iteration of Classic-VSH requires $O((\log n)(\log\log n))$ time per message bit. The first section will give an implementation of the new iteration function that runs in $O(\log n)$ time per message-bit. So we expect a speedup of factor $\log\log n$.

This chapter will discuss this idea in detail. Section 6.1 discusses how to implement this idea. The second section gives the result of our implementations on our 3.4 GHz Pentium IV system. The last section will discuss the security of this idea and give some practical solutions.

6.1 Pseudo code

This section shows how to implement the following idea of implementing VSH: instead of calculating the output hash by computing the products of primes in each iteration, compute iteratively the exponent vector $(e_i)_{i=1}^{K}$ of the resulting hash, reduce these numbers modulo $\phi(n)$, and then calculate the result by multiplying the products of primes.

Let $e_{i,j}$ denote the power of prime p_i, for which holds that

$$x_j \equiv \prod_{i=1}^{K} p_i^{e_{i,j}} \mod n,$$

for $i = 1, \cdots, K$ and $j = 0, \cdots, \mathcal{L} + 1$. By the iteration step of Classic-VSH, for

$i = 1, \cdots, K$ and $j = 0, 1, \cdots, \mathcal{L}+1$ it holds that $e_{i,j}$ satisfies

$$e_{i,0} = 0$$
$$e_{i,j+1} = \begin{cases} 2e_{i,j}+1 & \text{if } m_{j\cdot K+i} = 1 \\ 2e_{i,j} & \text{if } m_{j\cdot K+i} = 0 \end{cases}.$$

Multiplying by (a power of) two can be done on a computer very efficient by just shifting the bits to left. So we can implement the iteration that derives the exponent vector of the resulting hash $(e_i)_{i=1}^K = (e_{i,\mathcal{L}+1})$ very efficient. Let $X \ll i$ denote the left-shift of X over i-bits, i.e., $X * 2^i$. Pseudo code 6.1.1 shows how to find, iteratively, the exponent vector $(e'_i)_{i=1}^K \equiv (e_i \bmod \phi(n))_{i=1}^K$.

Pseudo-code 6.1.1. Iteration of Trapdoor-VSH (Find Exponents)

Input: Iteration index j and the exponent vector $(e_{i,j})_{i=1}^K$.

Output: The exponent vector $(e_{i,j+1})_{i=1}^K$.

Procedure:
```
for (i = 1; i ≤ K; i++) {
        e_{i,j+1} = e_{i,j+1} ≪_gmp 1;
        if (m_{j·K+i} == 1) {
          e_{i,j+1}++_gmp ;
        }
        if (e_{i,j+1} >_gmp φ(n)) {
          e_{i,j+1} = e_{i,j+1} −_gmp φ(n);
        }
}
Return (e_{i,j+1})_{i=1}^K
```

Pseudo code 6.1.1 has a lot of overhead: for each bit the GMP-shift, GMP-addition and GMP-comparison functions are called. From the discussion in Section 4.1 in Chapter 4, it follows that this Pseudo code can be improved by *inlining* the shifting and addition functions as much as possible. In addition, from Example 5.1.5 it follows that GMP is more efficient when applied to perform a reduction modulo $\phi(n)$ to a number which is much larger than $\phi(n)$, than when it has to perform a reduction modulo $\phi(n)$ to a number of almost equal size. This leads to Pseudo code 6.1.2, where $(A_{i,j})_{i=1}^K$ and C_j are temporary values, for which, initially, $A_{i,0} = C_0 = 0$, for all $i, \cdots, K$.

Pseudo-code 6.1.2. Iteration of Trapdoor-VSH (Find Exponents (continued))

Input: Iteration index j, the exponent vector $(e_{i,j})_{i=1}^{K}$, and the temporary values $(A_{i,j})_{i=1}^{K}$, C_j.
Let $(A_{i,j+1})_{i=1}^{K}$ be a 32-bit integer array.
Let C_{j+1} be a counter and $C = \phi(n)_{(32)}$.

Output: The exponent vector $(e_{i,j+1})_{i=1}^{K}$, and the temporary values $(A_{i,j+1})_{i=1}^{K}$, C_{j+1}.

Procedure:

```
/*Filling the 32 bit-array on the spot*/
for (i = 1; i ≤ K; i++) {
        A_{i,j+1} = A_{i,j} ≪ 1;
        if (m_{j·K+i} == 1) {
           A_{i,j+1}++;
        }
}
/*If A is filled or if j = L; update C and (e_{i,j+1}).*/
if (j + 1 mod 32 == 0 || j == L) {
        for (i = 1; i ≤ K; i++) {
                if(j + 1 mod 32 == 0) {
                   e_{i,j+1} = e_{i,j} ≪_gmp 32;
                   C_{j+1} = C_j + 1;
                } else {
                   e_{i,j+1} = e_{i,j} ≪_gmp (j + 1 mod 32);
                }
                e_{i,j+1} = e_{i,j} +_gmp A_{i,j};
                /*Reduce when e_{i,j+1} are about twice the size as φ(n)*/
                if( C_{j+1} == 2C || j == L) {
                   C_{j+1} = C;
                   e_{i,j+1} = e_{i,j+1}  mod_gmp  φ(n);
                }
        }
}

Return (A_{i,j+1})_{i=1}^{K}, C_{j+1}, (e_{i,j+1})_{i=1}^{K}
```

Lastly, since the message-bits are uniformly random, on average, the CPU-will wrongly predict the outcome of the statement `if(`$m_{j\cdot K+i}$`==1)` half of the time. So by the discussion in Section 4.1 of Chapter 4, Pseudo code 6.1.2 can be improved further by replacing

```
for (i = 1; i ≤ K; i++) {
        A_{i,j+1} = A_{i,j} ≪ 1;
        if (m_{j·K+i} == 1) {
           A_{i,j+1}++;
        }
}
```

by

```
for (i = 1; i ≤ K; i++) {
```

$A_{i,j+1} = (A_{i,j} \ll 1) \mid m_{j \cdot K+i};$
}

where | denote the bit-wise OR-function. Thus, we have eliminated pipeline delays by replacing a conditional statement with a binary operation. The next section shows that this small change to the source code doubles the performance of VSH! This final implementation of the new iteration function is referred to as *Find_Exponents_Final*.

It remains to show how to implement the product

$$\prod_{i=1}^{K} p_i^{e'_i} \mod n.$$

This can be done very efficient by using VSH's original iteration function, see Pseudo code 6.1.3. Straightforward multiplication would require $\sum_{i=1}^{K} e'_i = O(K \cdot n)$ multiplications and $O(n)$ reductions modulo n, whereas this Pseudo code requires
$O((K+1) \cdot \log n) = O(K \log n)$ multiplications, $O(\log n)$ squarings, and $O(\log n)$ reductions modulo n.

Pseudo-code 6.1.3. Finalizing

Input: The exponent vector $(e'_i)_{i=1}^{K}$.

Output: The Classic-VSH hash of message m.

Procedure: Let $b(x,i)$ denote the value of bit i of x, where $b(x,0)$ denotes the least significant bit of x.
Set $t = 1$ and $H = 1$.
for $(j = 1; j \leq S; j{+}{+})$ {
  for $(i = 1; i \leq K; i{+}{+})$ {
    if $(b(e'_i, S-j) == 1)$ {
      $t = t *_{\text{gmp}} p_i;$
    }
  }
  $H = H *_{\text{gmp}} H \mod_{\text{gmp}} n;$
  $H = H *_{\text{gmp}} t \mod_{\text{gmp}} n;$
}
Return H.

Algorithm 6.1.4 shows how the new iteration and finalization are used to calculate the Classic-VSH hash of message m.

Algorithm 6.1.4. Trapdoor-VSH

Input: An l-bit message m and an S-bit RSA modulus n.

Output: An S-bit hash of m.

Procedure: *Initialization:*
(Initial Values): Let $(e_{i,0})_{i=1}^{K} = (A_{i,j})_{i=1}^{K} = (0)_{i=1}^{K}$, and $C_0 = 0$.
(Number of iterations): Let $\mathcal{L} = \lceil \frac{l}{K} \rceil$.
(padding): Let $m_i = 0$ for $l < i \leq \mathcal{L}K$.
(Appending length block): Let l_i be bits so that
$l = \sum_{i=1}^{K} l_i 2^{i-1}$ and let $m_{\mathcal{L}K+i} = l_i$ for $i = 1, \cdots, K$.
Iteration:
For $j = 0, \cdots, \mathcal{L}$ compute
$\left((e_{i,j+1})_{i=1}^{K}, (A_{i,j+1})_{i=1}^{K}, C_{j+1}\right)$ =Find_Exponents_Final$(j, (e_{i,j})_{i=1}^{K}, (A_{i,j})_{i=1}^{K}, C_j)$.
Finalization:
Return Finalize$((e_{i,\mathcal{L}+1})_{i=1}^{K})$.

6.2 Results

This section presents the results of the speed of Trapdoor-VSH based on an S-bit modulus n on our 3.4 GHz Pentium IV system. By the new Computational VSSR Assumption of the previous Chapter, we take $S = S'$, i.e., we assume that Trapdoor-VSH is as secure as it is to factor an $S' = S$ bit number. Because the finalizing part of Algorithm 6.1.4 is equivalent to the iteration part of the Original Classic-VSH based on an $S \cdot K$-bit message, this idea benefits only for messages which are significantly larger than $S \cdot K$-bits i.e., suppose that the length of the input message is $\alpha \cdot S \cdot K$, for some α. The time that Algorithm 6.1.3 needs to calculate the finalizing part is about the time the full Original Classic VSH takes divided by α. It follows that if $\alpha >> 1$ then the time that Algorithm 6.1.3 requires to evaluate the finalizing part is negligible.

Because the iteration part of Algorithm 6.1.4 performs a reduction modulo $\phi(n)$ for messages which are larger than $K \cdot S$-bits, we expect a speed-up, with respect to Classic-VSH, for messages larger than 2^{7+10} bits= 16kilo Byte, if $S = 1024$ and $K = 131 \approx 2^7$. For the same K, we expect a speed-up, with respect to Classic-VSH if the input message is larger than 32kilo Byte, if $S = 2048$. Figure 6.1 shows the speed of Trapdoor-VSH against the size of the input message. With respect to Classic-VSH, we observe that Trapdoor-VSH is faster when the message is larger than 25kilo Bytes, if $S = 1024$, and for messages larger than 80 kilo Byte if $S = 2048$. Trapdoor-VSH is faster than Fast-VSH if the input message is larger than 200 kilo Byte, when $S = 1024$ and if the message is larger than 420 kilo Byte, when $S = 2048$. In the remainder of this section we consider 1 Mega-Byte messages.

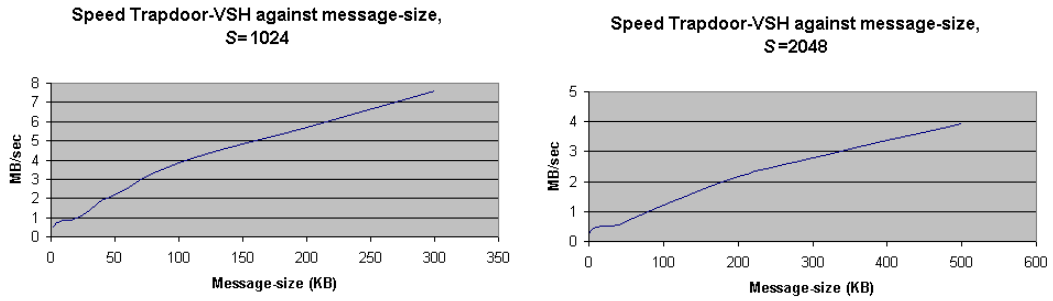

Figure 6.1: The speed of Trapdoor-VSH against the message-length, where on the left figure $S = 1024$ and $S = 2048$ on the right figure. Here, $K = 131$ on both Figures.

Ignoring the finalizing part, Pseudo code 6.1.1 implies that every iteration consists of K bit-shifts and at most K additions of 1 to $e_{i,j}$. Because $e_{i,j}$ is by construction less than $\phi(n) = O(n)$, it holds that these shifts and additions can be done in $O(K \cdot \log n)$ time. Because the value of $e_{i,j}$ is multiplied by 2 every iteration, for $i = 1, \cdots, K$, there are K reductions modulo n performed after every $O(\log_2(\phi(n))) = O(\log n)$ iterations. If Classic Modular Algorithms are used, then every reduction modulo $\phi(n)$ requires $O((\log n)^2)$ time, so that we conclude that Trapdoor-VSH requires

$$O\left(\frac{K \log n + K\frac{(\log n)^2}{\log n}}{K}\right) = O(\log n)$$

time per message-bit, which is a factor $\log \log n$ less than Classic-VSH requires per message-bit originally.

Remark 6.2.1. From Figure 6.1 one may be tempted to conclude that the speed of Trapdoor-VSH increases linear in the size of the message. This is not true as (1) the iteration part Trapdoor-VSH takes $O(\log n)$ time per message bit, which is independent of the message length ℓ, and (2) the finalizing part takes a fixed amount of time for large enough messages. So we expect the speed to grow to some constant level. However, for small messages it holds that the Finalizing part is the bottle-neck of Trapdoor-VSH. Now suppose that for messages of size less than some ℓ' the time taken by the iteration part is negligible with respect to the finalizing part. If there exist messages m_1 and m_2 of size less than ℓ', where m_2 has twice the size as m_1. If they are both large enough such that the Finalizing part takes the same amount of time, then the speed of Trapdoor-VSH on input m_2 is twice the speed of Trapdoor-VSH on input m_1. This could explain the linearity of Figure 6.1.

Table 6.1 shows the speed of Algorithm 6.1.4, for the different implementations for the iteration-function as described in previous subsection.

Remark 6.2.2. It follows from Table 6.1 that Pseudo code 6.1.1 has indeed a lot of overhead, so that the reduction of the overhead in Pseudo code 6.1.1 is very effective. Also, the removal of the `if` statement in Pseudo code 6.1.2 doubles the speed. The following results strengthen the argument that the latter improvement is due to better branch-prediction and/or scheduling (see again Section 4.1). The following table gives the speed of Trapdoor-VSH based on Pseudo code 6.1.2, where M_8 denotes the hexadecimal

S'	Pseudo code	K	S	MB/sec	speed-up
1024	Original Classic-VSH	153	1234	1.055	0.00%
	6.1.1: Trapdoor VSH (Find Exponents)	131	1024	1.034	-1.99%
	6.1.2: Trapdoor VSH (Find Exponents continued)	131	1024	5.025	476.30 %
	6.1.4: Trapdoor VSH (Find Exponents Final)	131	1024	10.870	1030.33 %

Table 6.1: The speed of the Trapdoor Classic-VSH based on the various implementations

representation of the 8-bit repetitive pattern of message m. So, for example, if $M_8 = 0xaa$, then m is equal to $10101010 \ldots 10101010$, and if M_8 =random then m is uniformly random. Because the removed `if` statement is one of the main instructions and because branch-predicting and scheduling can be done better for a fixed message m, we expect better results. Here, $S' = S = 1024$ and $K = 131$.

M_8	Speed (MB/sec)
random	5.034
0xff	6.452
0xaa	10.753

Note that the time needed to evaluate $m = 01010101 \cdots$ is about the same as the time that Trapdoor-VSH based on Algorithm 6.1.4 requires to evaluate the hash of a random message. This is what we expected, since the message bits of the random message are uniformly distributed.

The following Figure shows the speed of Algorithm 6.1.4 against K, when it evaluates the hash of a 1 MB message. For $S = 1024$ it follows that Trapdoor-VSH has the best

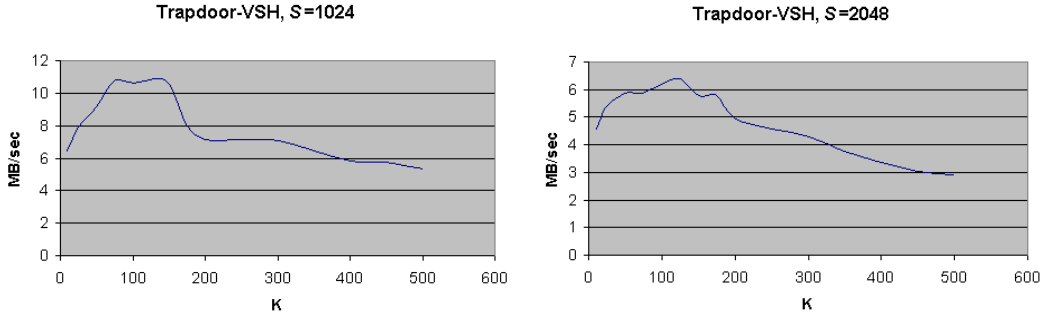

Figure 6.2: The speed of Trapdoor-VSH, where on the left figure $S = 1024$ and $S = 2048$ on the right figure.

performance if $K = k = 131$. The speed is then 10.870 MB per second, which is 10.3 times faster than 1.055 MB per second, the speed of Classic-VSH originally. Moreover, it is 2.88 times faster than 3.77 MB per second, the speed of Fast-VSH, originally. For $S = 2048$ it follows that Trapdoor-VSH has the best performance if $K = 125$. The speed is then 6.4 MB per second, which is 9.70 times faster than 0.66 MB per second, the speed of Classic-VSH originally. Moreover, it is 2.60 times faster than 2.457 MB per second, the speed of Fast-VSH originally.

6.3 Discussion

One might think to use Trapdoor-VSH based on Fast-VSH so that the other parties can use Fast-VSH, instead of Classic-VSH, which is about three times faster than than Classic-VSH. Consider Algorithm 6.1.4 based on K primes and an S-bit modulus n. Obviously, if no reductions modulo $\phi(n)$ are performed, then the finalization part is equivalent to Classic-VSH evaluating the input message. So with respect to Classic-VSH, the message should be at least KS-bits long to take advantage of the iteration of Algorithm 6.1.4. If the iteration function is changed to evaluate the exponents of the primes of the output of Fast-VSH, then the input message should be at least 2^bKS-bits long. Moreover, the finalizing part for these message is equivalent to Classic-VSH evaluating an 2^bKS-bit message. So for $S = 1024$, $b = 8$ and $K = 256 = 2^8$, Trapdoor-VSH based on Fast-VSH is effective for messages which are larger than 2^{10+8+8} bits $= 2^{26}$ bits $= 8$ Mega Byte. Moreover, the finalizing part will take as much time as Classic-VSH takes to evaluate an 8MB message. Hence, practically, it makes no sense to apply Trapdoor-VSH with respect to Fast-VSH.

Lastly, as originally the fastest version of VSH (the original version of Fast-VSH) requires 25 times more time than SHA1 to evaluate the hash of a message m, it follows that the Trapdoor-VSH requires 'only' 10 times more time to evaluate the hash of a message m than SHA1.

Chapter 7

On The Linearity of the Speed of VSH

As Lenstra notes in [18] with respect to factoring 1024 bit RSA-numbers: "It is about time to change". Based on the new developments in factoring large numbers, we should consider the consequences of using larger moduli to the applicability of VSH. Because this thesis deals with the speed of VSH, we will focus on how larger VSH-moduli affect the speed of VSH.

Consider VSH with modulus n. Let n' be the new VHS modulus, which is x times larger than n, i.e.

$$\frac{\log n'}{\log n} = x.$$

Roughly, we will show that since each iteration takes $O((\log n')^2)$ time, each iteration takes x^2 times the amount of time as it does with n. But as –under the new modulus– we process x times more bits each iteration, the total time used by VSH with modulus n' is $O(x)$ times the total running time of VSH with modulus n.

So we expect that the running time of VSH is linear in the size of the modulus. Our results support this when n is 1024 bits long and $x = 2$, see Table 7.1. Let S denote the bit-size of the modulus.

VSH-variant	S	speed (MB/sec)
Original Classic-VSH	1024	1.232
	2048	0.770
Optimized Classic-VSH	1024	1.780
	2048	1.186
Fast-VSH	1024	5.800
	2048	3.365
Trapdoor-VSH	1024	10.870
	2048	6.400

Table 7.1: The reduction of the speed of VSH, when the modulus size is doubled.

The first section shows that the running time of Classic-VSH is linear in the size of its modulus. The second section shows that the running time of Fast-VSH is linear in the size of its modulus and the last section shows that the running time of Trapdoor-VSH is linear in the size of its modulus, for large messages.

7.1 Classic-VSH

This section shows that the running time of Classic-VSH is in some sense linear in the modulus size it uses. We will prove this for the Original Classic-VSH and show that it also holds in some sense for the optimized version of Classic-VSH, where the number of message-bits processed each iteration is bounded.

Theorem 7.1.1. *The running time of the Original Classic-VSH is almost linear in the size of the modulus it uses. Precisely, let n be an S-bit modulus and n' an $(x \cdot S)$-bit modulus, for which holds that $1 < x < \log n$. If Original Classic-VSH, based on modulus n, takes $T_n(m)$ time to evaluate the hash of message m, it holds that*

$$\frac{T_{n'}(m)}{T_n(m)} = O(x).$$

Proof. Consider the original version of Classic VSH based on an S-bit modulus n. It follows from the analysis of the efficiency of Classic-VSH in subsection 2.2.2 of Chapter 2 that each iteration of Classic-VSH can be done in $O((\log n)^2)$ time. So if we take a new modulus n' of bit-size $x \cdot S$, each iteration will require $O((\log n')^2) = O(x^2(\log n)^2)$ time.

Let $R_n(m)$ denote the iteration numbers that Classic-VSH, based on modulus n requires to evaluate the hash of the l-bit message m. Recall that $R_n(m) = l/K_n + 1$. Let K_n be the value such that

$$\prod_{i=1}^{K_n} p_i < n < \prod_{i=1}^{K_n+1} p_i.$$

From Claim 2.2.4 in subsection 2.2.2 it follows that $K_n \sim (\log n)/(\log\log n)$. So, assuming $x < (\log n)^d$, for some constant d, we find that the iteration numbers is reduced by a factor

$$\begin{aligned}
\frac{R_{n'}(m)}{R_n(m)} &= \frac{\frac{l}{K_{n'}(m)} + 1}{\frac{l}{K_n} + 1} \\
&= O\left(\frac{K_n}{K_{n'}}\right) \\
&= O\left(\frac{\frac{\log n}{\log\log n}}{\frac{\log n'}{\log\log n'}}\right) \\
&= O\left(\frac{\frac{\log n}{\log\log n}}{\frac{x\log n}{\log x + \log\log n}}\right) \\
&= O\left(\frac{1}{x}\left(\frac{\log x}{\log\log n} + 1\right)\right) \\
&= O\left(\frac{1}{x}\right).
\end{aligned}$$

Hence,

$$\frac{T_{n'}(m)}{T_n(m)} = O\left(\frac{R_{n'}(m)(\log n')^2}{R_n(m)(\log n)^2}\right) = O\left(\frac{1}{x} \cdot \frac{x^2(\log n)^2}{(\log n)^2}\right) = O(x).$$

□

Consider the optimal version of Classic-VSH, based on modulus n, where the iteration function is implemented by Pseudo code 5.1.4, see page 49. Let n and n' satisfy the properties of Theorem 7.1.1. The model and the results presented in Chapter 5 suggest to take K_n not too large to get optimal performance. So it is reasonable to suppose that K_n is given by $c_n \cdot k_n$, where c_n is some small constant and k_n is the value such that

$$\prod_{i=1}^{k_n} p_i < n < \prod_{i=1}^{k_n+1} p_i.$$

Then, the fact that the values of the small primes increase logarithmic, i.e.,

$$\begin{aligned}
\log p_{K_n} &\approx \log(K_n \log K_n) \\
&= \log(c_n k_n \log c_n k_n) \\
&= \log(k_n(\log k_n + \log c_n)) + \log(c_n \log c_n k_n) \\
&\approx \log(k_n \log k_n) \\
&\approx \log p_{k_n},
\end{aligned}$$

implies that an extra multiplication modulo n in Pseudo code 5.1.4 is required after approximately every k_n message-bits. Thus, the number of extra multiplications modulo n of the optimal implementation of Classic-VSH will be bounded from above by $c' \frac{K_n}{k_n}$, for some small constant $c' > 1$. Hence, the iteration function of this implementation of Classic-VSH requires

$$O\left(\frac{K_n}{k_n}(\log n)^2\right) = O\left(\frac{c_n k_n}{k_n}(\log n)^2\right) = O((\log n)^2)$$

time.

Let m be an l-bit message. From the proof of Theorem 7.1.1 it follows that

$$\begin{aligned}
\frac{T_n(m)}{T_{n'}(m)} &= \frac{R_n(m)}{R_{n'}(m)} \cdot O(x^2) \\
&= O\left(x^2 \frac{K_n}{K_{n'}}\right) \\
&= O\left(x^2 \cdot \frac{c_n}{c_{n'}} \cdot \frac{k_n}{k_{n'}}\right) \\
&= O\left(x^2 \cdot \frac{1}{x}\right) \\
&= O(x).
\end{aligned}$$

We conclude that the speed of Classic-VSH based on an S-bit modulus n behaves linearly in S, if S increases less than a factor $\log n$.

7.2 Fast-VSH

This section shows, similarly to the previous section, that the running time of Fast-VSH is also linear in some sense in the size of the modulus it uses. We will prove this for some instance of Fast-VSH. Then we will show that it also holds for Fast-VSH, where the number of b-bit blocks K of message m that are processed each iteration, is bounded.

Theorem 7.2.1. *The running time of Fast-VSH, where the number of b-bit blocks K of message m that are processed each iteration, is chosen so that*

$$\prod_{i=1}^{K} p_{i2^b} < n < \prod_{i=1}^{K+1} p_{i2^b},$$

is almost linear in the size of the modulus it uses. Precisely, consider Fast-VSH, where the number of b-bit blocks K of message m that are processed each iteration, is chosen so that above equation holds. Let n be an S-bit modulus and n' an $(x \cdot S)$-bit modulus, for which holds that $1 < x < \log n$. If Fast-VSH, based on modulus n, takes $T_n(m)$ time to evaluate the hash of message m, it holds that

$$\frac{T_{n'}(m)}{T_n(m)} = O(x).$$

Proof. Consider Fast VSH based on an S-bit modulus n. It follows from the analysis of Fast-VSH in subsection 2.3.3 of Chapter 2 that each iteration of Fast-VSH can be done in $O((\log n)^2)$ time. So if we take a new modulus n' of bit-size $x \cdot S$, each iteration will require $O((\log n')^2) = O(x^2(\log n)^2)$ time.

Let $R_n(m)$ denote the iteration numbers that Fast-VSH, based on modulus n requires to evaluate the hash of the l-bit message m. Recall that $R_n(m) = l/(bK_n) + 1$. Fix the chunk-size b and let K_n be the value such that

$$\prod_{i=1}^{K_n} p_{i2^b} < n < \prod_{i=1}^{K_n+1} p_{i2^b}.$$

From the proof of Lemma 2.3.3 in Subsection 2.3.3 it follows that

$$bK_n \sim \frac{b\log(2n)}{\log(2^b\log(2n))} - b \sim \frac{b\log(n)}{\log(2^b\log(2n))}.$$

So, assuming $1 < x < (\log n)^d < (\log(2n))^d$, for some constant d, we find that the iteration numbers is reduced by a factor

$$\begin{aligned}
\frac{R_{n'}}{R_n} &= \frac{\frac{l}{K_n b} + 1}{\frac{l}{K_{n'} b} + 1} \\
&= O\left(\frac{K_{n'} b}{K_n b}\right) \\
&= O\left(\frac{\log(n)}{\log\left(2^b \log 2n\right)} \Big/ \frac{x\log(n)}{\log\left(x2^b \log 2n\right)}\right) \\
&= O\left(\frac{1}{x} \cdot \frac{\log\left(x2^b \log 2n\right)}{\log\left(2^b \log 2n\right)}\right) \\
&= O\left(\frac{1}{x} \cdot \frac{2\log\left(2^b \log 2n\right)}{\log\left(2^b \log 2n\right)}\right) \\
&= O\left(\frac{1}{x}\right)
\end{aligned}$$

Hence,

$$\frac{T_{n'}(m)}{T_n(m)} = O\left(\frac{R_{n'}(m)(\log n')^2}{R_n(m)(\log n)^2}\right) = O\left(\frac{1}{x} \cdot \frac{x^2(\log n)^2}{(\log n)^2}\right) = O(x).$$

□

Consider the optimal version of Fast-VSH based on modulus n and chunk-size b, where the implementation of the iteration function is given by Pseudo code 4.0.3 on page 26. Let n and n' satisfy the properties of Theorem 7.2.1. Again, the model and the results presented in Chapter 5 suggest to take K_n not too large to get optimal performance. So, again, it is reasonable to suppose that K_n is given by $c_n \cdot k_n$, where c_n is some small constant and k_n is the value such that

$$\prod_{i=1}^{k_n} p_{i2^b} < n < \prod_{i=1}^{k_n+1} p_{i2^b},$$

Then, the fact that the values of the small primes increase logarithmic, i.e.,

$$\begin{aligned}
\log p_{2^b K_n} &\approx \log(2^b K_n \log 2^b K_n) \\
&= \log(c_n 2^b k_n \log c_n 2^b k_n) \\
&= \log(2^b k_n \log(2^b k_n + \log c_n) + \log(c_n \log c_n 2^b k_n)) \\
&\approx \log(2^b k_n \log 2^b k_n) \\
&\approx \log p_{2^b k_n},
\end{aligned}$$

implies that an extra multiplication modulo n in Pseudo code 4.0.3 is required after approximately every bk_n message-bits. Thus, the number of extra multiplications modulo n of the optimal implementation of Fast-VSH will be bounded from above by $c' \frac{bK_n}{bk_n}$, for some small constant $c' > 1$. Hence, the iteration function of this implementation of Fast-VSH requires

$$O\left(\frac{K_n}{k_n}(\log n)^2\right) = O\left(\frac{c_n k_n}{k_n}(\log n)^2\right) = O((\log n)^2)$$

time.

Let m be an l-bit message. From the proof of Theorem 7.2.1 it follows that

$$\begin{aligned}
\frac{T_n(m)}{T_{n'}(m)} &= \frac{R_n(m)}{R_{n'}(m)} \cdot O(x^2) \\
&= O\left(x^2 \frac{bK_n}{bK_{n'}}\right) \\
&= O\left(x^2 \cdot \frac{c_n}{c_{n'}} \cdot \frac{k_n}{k_{n'}}\right) \\
&= O\left(x^2 \cdot \frac{1}{x}\right) \\
&= O(x).
\end{aligned}$$

We conclude that the speed of Fast-VSH based on an S-bit modulus n behaves linearly in S, if S increases less than a factor $\log n$.

7.3 Trapdoor-VSH

This section shows that the running time of Trapdoor-VSH is linear in the size of the modulus it us based on, if the evaluation time of the finalization part is negligible. Let n be the modulus used by Trapdoor-VSH, and n' be the new modulus. Let x be value so that $\log n' = x \log n$.

As we discussed in the results of Chapter 6, Trapdoor-VSH requires $O(\log n)$ time per message-bit. So if Trapdoor-VSH is applied with modulus n', then Trapdoor-VSH requires $O(x)$ times more time per message-bit. Hence, the evaluation time of the iteration of Trapdoor-VSH is linear in the size of n. Moreover, if the evaluation time of the finalization part is negligible, the running time of Trapdoor-VSH is linear in the size of n.

The running time of the finalization part of Trapdoor-VSH using K_n primes is by construction equivalent to the running time of Classic-VSH, using K_n primes, evaluating a (l_n)-bit message m, where $l_n = K_n \lceil \log_2 n \rceil$. So, the fraction of the iteration numbers within the finalizing part of Trapdoor-VSH based on moduli n and n' respectively is given by

$$\begin{aligned}
\frac{\frac{l_{n'}}{K_{n'}} + 1}{\frac{l_n}{K_n} + 1} &= rO\left(\frac{\frac{K_{n'} \log n'}{K_{n'}}}{\frac{K_n \log n}{K_n}}\right) \\
&= O\left(\frac{\log n'}{\log n}\right) \\
&= O(x).
\end{aligned}$$

It also follows from the proof of theorem 7.1.1 that each iteration within the finalizing part requires $O(x^2)$ times more time. Hence, the finalizing part of Trapdoor-VSH, based on modulus n', requires $O(x^3)$ times more time than Trapdoor-VSH, besed on modulus n.

So we conclude that if the input message is "large enough", then the running time of Trapdoor-VSH is linear with respect the modulus length. For small messages, it holds that the running time of Trapdoor-VSH is almost cubic in the size of the modulus.

Chapter 8

Conclusion and Further Research

In this thesis, we have described The Very Smooth Hash Function (VSH) and how we improved its speed. This thesis doesn't consider memory usage and the length of the source code. We note here that especially with respect to Fast-VSH, memory usage can be an important issue to consider, because huge lists of primes are used. Chapter 2 discussed the original paper of VSH ([6]), in which two variants of VSH are discussed: Fast-VSH and Classic-VSH. Originally, Fast-VSH took about 25 times more time than SHA1 to compute a hash and Classic-VSH took about 80 times more time than SHA1.

In Chapter 4 we showed how improving the (C) source code improves the speed of Classic-VSH by about 10%. In Chapter 5 we gave a model of how the workload in terms of single precision multiplications behaves with respect to the number of message bits processed each iteration. Furthermore, we introduced a new computational security assumption, which is less pessimistic than the original computational security assumption as defined in [6]. Finally, we showed how the new computational security assumption leads to a speedup of about 70% with respect to Classic-VSH and a speedup of about 50% with respect to Fast-VSH. In Chapter 6 we introduced Trapdoor-VSH, that uses secret information to calculate the hash very efficiently. This version of VSH is about 10 times faster than Classic-VSH originally and 2.5 times faster than Fast-VSH originally.

Table 8.1 shows the speedups with respect to the original versions of VSH. The run times are obtained using GMP-based implementations and a 3.4 GHz Pentium IV computer. Here, S' denotes the size of the VSH modulus n such that finding a collision for VSH is as hard as factoring an S-bit (RSA) composite. Here we summarize the best versions of VSH; in Appendix A we give the speedups of all variants of VSH.

It follows that our best version for Classic-VSH is about 47 times slower than SHA1 (as opposed to 80 times originally), and our best version of Fast-VSH is about 17 times slower than SHA1 (as opposed to 25 times originally). Trapdoor-VSH is about 10 times slower (on 1 Mega Byte messages) than SHA1.

Although Trapdoor-VSH requires knowledge of the factorization of n, we showed in Chapter 6 that this version is still very applicable in some situations. For example, [6] showed that VSH can be used efficiently in Cramer-Shoup Signature schemes. So if some company has to sign many documents, whereas the verifiers has to verify only a few of such signatures, then the company can benefit of the speed of Trapdoor-VSH while the verifier does not suffer so much from using Classic-VSH instead of Fast-VSH.

A. Lenstra gives in [20] a relation between breaking RSA by means of factoring its

S	VSH-variant	S'	(optimal) K	speed (MB/sec)	Scaled to 1 GHz	speed-up
1024	Original Classic-VSH	1234	153	1.055	0.310	0.00%
	Classic-VSH: Pseudo Code 4.2.3	1024	700	1.780	0.524	68.72%
	Trapdoor-VSH	1024	131	10.870	3.197	1030%
1024	Original Fast-VSH	1516	256	3.770	1.109	0.00%
	Original Fast-VSH	1024	500	5.800	1.706	53.84%
	Trapdoor-VSH	1024	131	10.870	3.197	288%
2048	Original Classic-VSH	2398	272	0.681	0.200	0.00%
	Classic-VSH: Pseudo Code 4.2.3	2048	1300	1.186	0.349	74.16%
	Trapdoor-VSH	2048	125	6.400	1.882	970%
2048	Original Fast-VSH	2874	1024	2.457	0.723	0.00%
	Original Fast-VSH	2048	1000	3.365	0.990	36.96%
	Trapdoor-VSH	2048	125	6.400	1.882	260%

Table 8.1: Speedups of the best versions of VSH

modulus and performing an exhaustive search on λ-bit keys in a symmetric encryption system (such as DES). λ is called the *security level*. Because our new Computational VSSR Assumption implies that finding a collision for VSH with an S-bit (RSA) modulus n is as hard as factoring n, this analysis also applies to VSH. So Table 4 of [20] shows that VSH does not provide adequate protection anymore if $S = 1024$. If $S = 2048$, then VSH offers adequate protection until the year 2030 if we are pessimistic and 2040 if we are optimistic. We showed in Chapter 7 that the speed of VSH is linear in the size of its modulus. As Table 8.1 also shows, the speed of VSH with $S = 2048$ is much higher than one-eight of the speed of VSH with $S = 1024$. It follows that the efficiency of VSH is not affected as much as RSA, when the size of the modulus has to increase in order to provide adequate security.

In conclusion, we improved the speed of VSH significantly. However, VSH is still quite slow compared to common Hash Functions like SHA1. But, VSH is important as it is a step towards closing the gap between hash functions with provable security properties and practical hash functions. Because we increased the speed of VSH significantly, our work is another important step towards closing the gap.

We think that the gap can be closed a bit more by considering the following ideas:

- With respect to the implementation of VSH:
  - Our implementations are based on GNU's Multiple Precision Library, which seems to be the most efficient library to perform multiple precision calculations. As we showed in Chapter 3 GMP uses the very efficient Karatsuba's Algorithm. As we conclude in Section 4.4, based on the profiles as shown in Appendix E.2.5, GMP hardly applies Karatsuba's Algorithm for some reason. Why is this? And how can we force GMP to use Karatsuba's Algorithm more?
  - Moreover, it follows from [14] that GMP also supports (asymptotically) more efficient algorithms than Karatsuba's Algorithm, like Fast Fourier Transforms. For some reason GMP does not apply these in our implementations. How can these algorithms improve the speed even more and how can we force GMP to use these?
  - Besides the algorithms that GMP uses, there is room for gaining speed by using a more optimized version of GMP. For example J.W. Martin writes on his page

("http://www.math.jmu.edu/∼martin") how GMP can be optimized for Intel Core 2 Machines. In addition, on the GMP benchmark page ("http://gmplib.org/gmpbench.html") it is shown how to compile GMP on a computer, depending on what processor the computer has, to make GMP as efficient as possible on that computer. We note that it has been reported (according to [5]) that GMP is about one-third faster on a 64-bit processor than it is on a 32-bit processor. We used a 32-bit processor.

- Nowadays computers often have more than one processor. So suppose one has access to x processors. The speed of VSH could be improved by about x times by using them all. For example, let one processor do the initialization step. Let K be number of message bits that are processed each iteration of VSH. By construction, the initialization part of VSH pads the message such that its length becomes a multiple of K. Now, let the first processor perform an iteration on the first K message bits, and let the second processor perform an iteration on the second K message bits, etcetera.

  Maybe, there are more efficient ways to use more processors.

- VSH is not provable one-way. Is it possible to make VSH provable one-way without losing the provable collision resistant property?

- With respect to Trapdoor-VSH: Currently, only parties that generate the hash function have access to the trapdoor information and can use the Trapdoor information. Is it possible to use existing network security protocols to somehow let the verifier use the trapdoor information without getting to know anything about it?

Bibliography

[1] Digital signature standard. *FIBS-PUB 186-2*, 2000.

[2] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational diffie-hellman. In *Public Key Cryptography*, pages 229–240, 2006.

[3] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. *Advances in Cryptoloy-CRYPTO 1993*, 1993.

[4] D. Charles, E. Goren, and K. Lauter. Cryptographic hash functions from expander graphs. *Cryptology eprint archive*, Report 2006/021, 2006.

[5] Scott Contini. Private chat with Dr. Scott Contini.

[6] Scott Contini, Ron Steinfeld, and Arjen K. Lenstra. VSH, an efficient and provable collision-resistant hash function. *Cryptology ePrint Archive: Report 2005/193*, 2005.

[7] Scott Contini, Ron Steinfeld, Jozef Pieprzyk, and Krystian Matusiewicz. A critical look at cryptographic hash function literature. *ECRYPT Hash Workshop*, 2007.

[8] Thomas H. Cormen, Charles E, Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[9] R. Crandall and C. Pomerance. *Prime Numbers. A Computational Perspective.* Springer-Science + Business Media, second edition, 2005.

[10] I. Dåmgard. Collision-free hash functions and public key signature schemes. In *EUROCRYPT 87*, volume 304 of *LCNS*, pages 203–216, Berlin, 1987. Springer-Verlag.

[11] S.J.A. de Hoogh. Cryptographic hash functions, and what happened on the crypto 2004 conference in santa barbara? Bachelor thesis, Eindhoven University of Technology, 2005.

[12] E.Bach and J.Shallit. *Algorithmic Number Theory, volume 1: Efficient Algorithms*, volume 1 of *Foundation of Computing*. The MIT Press, 1996.

[13] Agner Fog. Branch prediction in the pentium family; how the branch prediction mechanism in the pentium has been uncovered with all its quirks, and the incredibly more effective branch prediction in the later versions. http://www.x86.org/articles/branch/branchprediction.htm.

[14] GNU-team. GNU MP. http://gmplib.org/manual/index.html.

[15] Brian J. Gough. *An Introduction to GCC - for the GNU compilers gcc and g++.* Network Theory Ltd, 2004.

[16] G.J.O. Jameson. *The Prime Number Theorem.* Camebridge University Press, 2003.

[17] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. *Advances in Cryptology EUROCRYPT 2005*, 3494/2005:474–490, 2005.

[18] Jeremy Kirk. Researcher: RSA 1024-bit encryption not enough As computers and math techniques become more powerful and sophisticated, current encryption standards could be made obsolete in as little as five years. http://www.infoworld.com/article/07/05/23/RSA-1024-bit-encryption-not-enough_1.html.

[19] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition).* Addison-Wesley Professional, November 1997.

[20] A. Lenstra. *Handbook of Information Security*, chapter Key Lengths.

[21] Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes, and Paul Leyland. Factoring estimates for a 1024-bit rsa modulus. In *proc. Asiacrypt 2003*, volume 2894 of *LCNS*, pages 331–346. Springer, 2003.

[22] Alfred J. Menezes, Paul C. van Oosrschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[23] D. Pointcheval. The composite discrete logarithm and secure authentication. *PKC 2000*, 1751 of *LNCS*:page 113–128, 2000.

[24] Phillip Rogaway. Formalizing human ignorance. *Vietcrypt 2007*, 2007.

[25] A. Shamir and Y. Tauman. Improved online/offline signature schemes. *CRYPTO 2001*, 2139 of *LNCS*:page 355–367, 2001.

[26] I.E. Shparlinski and I.F. Blake. Statistical distribution and collisions of the vsh. *J. Math. Cryptology*, (to appear), 2006.

[27] X. Wang, Y.L. Yin, and H. Yu. Finding collisions in the full sha-1. *Advances in Cryptology Crypto05*, Springer-Verlag, 2005.

[28] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. *Eprint Archive*, 2004.

Appendix A

Full Speed Table

S	VSH-variant	S'	(optimal) K	speed (MB/sec)	Scaled to 1 GHz	speed-up
1024	Original Classic-VSH	1234	153	1.055	0.310	0.00%
1024	Original Classic-VSH (No reduction)	1024	300	1.393	0.410	32.04%
	Original Classic-VSH (Extra reduction)	1024	500	1.490	0.438	41.23%
	Classic-VSH: Pseudo Code 4.3.1	1024	500	1.597	0.470	51.37%
	Classic-VSH: Pseudo Code 4.2.2	1024	700	1.779	0.523	68.63%
	Classic-VSH: Pseudo Code 4.2.3	1024	700	1.780	0.524	68.72%
	Trapdoor-VSH	1024	131	10.870	3.197	1030%
1024	Original Fast-VSH	1516	256	3.770	1.109	0.00%
	Original Fast-VSH	1024	500	5.800	1.706	53.84%
	Fast-VSH: Treebased	1024	500	4.650	1.368	23.34%
	Trapdoor-VSH	1024	131	10.870	3.197	288%
2048	Original Classic-VSH	2398	272	0.681	0.200	0.00%
2048	Original Classic-VSH (No reduction)	2048	500	0.865	0.254	27.02%
	Original Classic-VSH (Extra reduction)	2048	700	0.998	0.294	46.55%
	Classic-VSH: Pseudo Code 4.3.1	2048	1100	1.076	0.316	58.00%
	Classic-VSH: Pseudo Code 4.2.2	2048	1500	1.145	0.337	68.14%
	Classic-VSH: Pseudo Code 4.2.3	2048	1300	1.186	0.349	74.16%
	Trapdoor-VSH	2048	125	6.400	1.882	970%
2048	Original Fast-VSH	2874	1024	2.457	0.723	0.00%
	Original Fast-VSH	2048	1000	3.365	0.990	36.96%
	Fast-VSH: Treebased	2048	750	3.048	0.897	24.05%
	Trapdoor-VSH	2048	125	6.400	1.882	260%

Table A.1: Speedups of all versions of VSH (as described in Chapter 4)

Appendix B

Number Theoretic Background

In this chapter we will give some important and fundamental definitions, theorems and lemmas that are used throughout this thesis. We refer to [12] and [16] for more details.

B.1 The Prime Number Theorem

Definition B.1.1. *Let x be integer. Then $\Pi(x) := \{0 < i \leq x |\ i \text{ is prime}\}$ denotes the set of all primes of value at most x. $\pi(x)$ denotes the number of primes of value at most equal to x. In other words:*

$$\pi(x) := \#\Pi(x) \tag{B.1}$$

Definition B.1.2. *Chebyshev θ: Let x be an integer. Then $\theta(x)$ denotes the sum over the logarithm of all primes of value at most equal to x. In other words:*

$$\theta(x) := \sum_{i \in \Pi(x)} \log(i) \tag{B.2}$$

One of the most fundamental theorems in number theory is the prime number theorem.

Theorem B.1.3. *Prime Number Theorem: Let x be an integer. Then*

$$\pi(x) \sim \frac{x}{\log x}.$$

For an accessible introduction to the proof of this theorem see [16].

The following corollaries follow from the prime number theorem. See also [16] and chapter 8 of [12].

Corollary B.1.4. *Let x be an integer. Then,*

$$\theta(x) \sim x.$$

Corollary B.1.5. *Let n be a positive integer. Then,*

$$p_n \sim n \log n$$

From these two Corollaries we claim the following:

Claim B.1.6. *Let n be a positive integer. Then,*

$$\theta(p_k) \sim k \log k.$$

Proof. This proof is based on the transitivity property of $\sim$. To demonstrate this one easily derives:

$$\lim_{k\to\infty} \frac{\theta(p_k)}{k \log k} = \lim_{k\to\infty} \frac{\theta(p_k)}{p_k} \cdot \frac{p_k}{k \log k} = 1.$$

using Corollaries B.1.4 and B.1.5. □

One of the building blocks of the proof of the Prime Number Theorem is the following lemma (Proposition 1.3.6 of [16], or lemma 2.5.1 of [12]):

Lemma B.1.7. *Abel's Identity Let $a(k)_{k\in\mathbb{N}}$ be some real valued series and let $A(n) = \sum_{i=1}^{n} a(i)$ denote its partial sum. If some real valued function f has a continuous derivative on $[1, x]$, then*

$$\sum_{i=1}^{x} a(i)f(i) = A(x)f(x) - \int_1^x A(t)f'(t)\mathrm{dt}.$$

Finally we state the following theorem (Theorem 2.6.1 from [12]).

Theorem B.1.8. *Let f be a positive real-valued function defined in $[a, \infty)$. Assume that f is continuously differentiable, and that $f'(x)/f(x) \sim \mu/x$ for some real number $\mu > -1$. Then if $\mu \neq 0$, we have*

$$\int_a^x f(t)\mathrm{dt} \sim \frac{\mathrm{xf(x)}}{\mu + 1}.$$

If $\mu = 0$, then

$$\int_a^x f(t)\mathrm{dt} \sim \mathrm{xf(x)}.$$

Appendix C

Some Proofs

C.1 The Security Proof of Fast VSH

Theorem C.1.1. *Fast VSH is collision resistant under the VSSR assumption. In other words: Finding a collision for algorithm 2.3.1 is as hard as solving VSSR.*

Proof. This proof is essentially exactly the same as the proof for Classic VSH (2.2.2). Again, let x_j and x'_j denote the j-th iterated values of algorithm 2.3.1 applied to m and m' respectively. Let $l, \mathcal{L}$ and $l', \mathcal{L}'$ denote the bit-length and the number of blocks respectively of m and m'. As m and m' collide it holds that $m \neq m'$ and $x_{\mathcal{L}+1} = x_{\mathcal{L}'+1}$.

Firstly, consider the case in which $l = l'$. Let $m[j]$ denote the j-th b-bit block of m, and $m(j)$ the j-th kb-bit block of m, and we define $m(\mathcal{L}+1) = \emptyset$. Let t be the largest index such that $(x_t, m(t)) \neq (x'_t, m'(t))$, but $(x_j, m(j)) = (x'_j, m'(j))$ for $t < j \leq \mathcal{L}+1$. Because $l = l'$ it holds again that $t \leq \mathcal{L}$. From the choice of t it holds that

$$(x_t)^2 \times \prod_{i=1}^{k} p_{(i-1)2^b + m[tkb+i]+1} \equiv (x'_t)^2 \times \prod_{i=1}^{k} p_{(i-1)2^b + m'[tkb+i]+1} \mod n. \tag{C.1}$$

Define $S_m(t) := \{(i-1)2^b + m[tkb+i] + 1 : 1 \leq i \leq k\}$. Now let $\Delta = \{i \in \{1, \cdots, k2^b\} :$
$i \in S_m(t) \triangle S_{m'}(t)\}$ and $\Delta_{10} = \{i \in \{1, \cdots, k2^b\} : i \in S_m(t) \backslash S_{m'}(t)\}$. Here, $\triangle$ denotes the symmetric difference, i.e., $i \in A \triangle B \Leftrightarrow i \in A \backslash B \cup B \backslash A$. As the primes p_i are coprime to n for all $i = 1, \cdots, k2^b$ it holds that all primes p_i are invertible modulo n, and by the iteration step of algorithm 2.3.1 it holds also that $(x_t)^2$ and $(x'_t)^2$ are invertible modulo n. So equation (C.1) is equivalent to:

$$\left[\frac{x_t}{x'_t} \times \prod_{i \in \Delta_{10}} p_i \right]^2 \equiv \prod_{i \in \Delta} p_i \mod n \tag{C.2}$$

Obviously if $\Delta \neq \emptyset$ then equation (C.2) solves VSSR as stated in Definition 2.1.7. If $\Delta = \emptyset$ then also $\Delta_{10} = \emptyset$ and it follows from equation (C.2) that $(x_t)^2 \equiv (x'_t)^2 \mod n$. Recall that $x_0 = x'_0 = 1$. As $\Delta = \emptyset$ implies that $m(t) = m'(t)$ it follows from the choice of t and $m \neq m'$ that $t \geq 1$. If $x_t \not\equiv \pm x'_t \mod n$ then it follows from remark 2.1.5 that VSSR can be solved by factoring n. If $x_t \equiv \pm x'_t \mod n$, then $x_t \equiv -x'_t \mod n$, because $m(t) = m'(t)$ implies by the

choice of t that $x_t \neq x'_t$. It follows that

$$\begin{aligned} x_t \equiv -x'_t \mod n \quad &\Leftrightarrow \\ (x_{t-1})^2 \times \prod_{i=1}^{k} p_{(i-1)2^b+m[(t-1)bk+i]+1} \equiv -(x'_{t-1})^2 \times \prod_{i=1}^{k} p_{(i-1)2^b+m'[(t-1)bk+i]+1} \mod n \quad &\Leftrightarrow \\ \left[\frac{x_t}{x'_t}\right]^2 \equiv -1 \times \prod_{i=1}^{k} p_{(i-1)2^b+m[(t-1)bk+i]+1} \prod_{j=1}^{k} \left(p^{-1}_{(j-1)2^b+m'[(t-1)bk+j]+1}\right) \mod n, \end{aligned}$$

A solution to VSSR as $p_0 = -1$ has an odd degree. Thus for the case $l = l'$ it holds that the colliding messages m and m' can be used to find a solution for VSSR.

Next, consider colliding messages m and m', where $l \neq l'$. Since $l \neq l'$, it holds for at least one $i \in \{1, \cdots, k\}$ that $l_i \neq l'_i$. Similar to equation (C.1), we find from $x_{\mathcal{L}+1} = x'_{\mathcal{L}'+1}$ and the fact $m_{\mathcal{L}\cdot k+i} = l_i$ (see initialization part of algorithm 2.3.1) that

$$(x_{\mathcal{L}})^2 \times \prod_{i=1}^{k} p_i^{l_i} \equiv (x'_{\mathcal{L}'})^2 \times \prod_{i=1}^{k} p_i^{l'_i} \mod n. \tag{C.3}$$

Letting $\Delta_l := \left\{i \in \{1, \cdots, k2^b\} : i \in S_m(\mathcal{L}) \triangle S_{m'}(\mathcal{L}')\right\}$ and $(\Delta_l)_{10} := \left\{i \in \{0, \cdots, k2^b\} : i \in S_m(\mathcal{L}) \backslash S_{m'}(\mathcal{L}')\right\}$, we derive similar to equation (C.2) an immediate solution to VSSR, since $\Delta_l \neq \emptyset$:

$$\left[\frac{x_{\mathcal{L}}}{x_{\mathcal{L}'}} \times \prod_{i \in (\Delta_l)_{10}} p_i\right]^2 \equiv \prod_{i \in \Delta_l} p_i \mod n, \tag{C.4}$$

which concludes the proof. □

C.2 $\sigma_{\bar{n},n}$ being Existentially Unforgeable

Theorem C.2.1. *Suppose that VSH$_n$ is collision resistant and suppose that the signature scheme using signing function $\sigma_{\bar{n}}$ is existentially unforgeable under chosen message attacks. Then, the signature scheme as given in subsection 2.4.1 is existentially unforgeable under adaptive chosen message attacks.*

Proof. Assume that there exists an polynomial time adversary $\mathcal{A}$, which is able with non-negligible probability to forge a valid message-signature pair $(\tilde{m}, \tilde{\sigma}$ after, say, q signature queries on the messages $m_1, \cdots, m_q$, where $\tilde{m} \neq m_i$ for all $i = 1, \cdots, q$.

Suppose that $\mathcal{A}$ gets signature (m_i, σ_i) after query m_i. We distinguish two types of forgeries:

- **Forgery I:** For $(\tilde{(m)}, \tilde{\sigma})$ it holds that there exists an $i \in \{1, \cdots, q\}$ such that $\tilde{\sigma} = \sigma_i$.
- **Forgery II:** For $(\tilde{(m)}, \tilde{\sigma})$ it holds that $\tilde{\sigma} \neq \sigma_i$, for all $i = 1, \cdots, q$.

We will show that there exists an adversary $\mathcal{B}$ that either can create a collision for VSH$_n$ if the forgery is of type I or a forgery to the scheme based on $\sigma_{\bar{n}}$ if the forgery is of type II, using $\mathcal{A}$.

Firstly, assume that the forgery is of type I: Let $\mathcal{B}_1$ be a polynomial time adversary that is given public key n. $\mathcal{B}_1$ runs $\mathcal{A}$ as follows to obtain a collision for VSH$_n$.

setup $\mathcal{B}_1$ generates a random $S/2$-bit prime $\bar{p}$ and a random $(S/2+1)$-bit prime $\bar{q}$. He then computes $\bar{n} = \bar{p}\bar{q}$ and $\phi(\bar{n}) = (\bar{p}-1)(\bar{q}-1)$. He chooses a private key d co-prime to $\phi(\bar{n})$. Lastly, $\mathcal{B}_1$ send the public key $(\bar{n}, n, e)$ to $\mathcal{A}$, where $e = 1/d \bmod \bar{n}$.

queries When $\mathcal{A}$ asks a signature on message m_i, $\mathcal{B}_1$ responds with (m_i, σ_i), where $\sigma_i = \sigma_{\bar{n}}(t_i)$ and $t_i = \mathrm{VSH}_n(m_i)$.

output When $\mathcal{A}$ outputs $(\tilde{m}, \tilde{\sigma})$, $\mathcal{B}_1$ looks up the value j so that $\sigma_j = \tilde{\sigma}$. Then, he outputs the colliding messages for VSH_n: $\tilde{m}$ and m_j, both with VSH_n output t_j.

This always works, because $\sigma_{\bar{n}}$ is by construction injective.

Secondly, assume that the forgery is of type II: Let $\mathcal{B}_2$ be a polynomial time adversary that is given public key $(\bar{n}, e)$. $\mathcal{B}_2$ runs $\mathcal{A}$ as follows to obtain a forgery for $\sigma_{\bar{n}}$.

setup $\mathcal{B}_2$ chooses random $S/2$-bit primes p and q, to obtain $n = pq$. He provides $\mathcal{A}$ with the public key $(\bar{n}, n, e)$.

queries Upon receive of query m_i, $\mathcal{B}_2$ responds as follows: firstly he queries $t_i = \mathrm{VSH}_n(m_i)$ to his challenger. After receiving the signature (t_i, σ_i), $\mathcal{B}_2$ sends (m_i, σ_i) to $\mathcal{A}$.

output When $\mathcal{A}$ outputs forgery $(\tilde{m}, \tilde{\sigma})$, then $\mathcal{B}_2$ outputs the forgery $(\tilde{t}, \tilde{\sigma})$, where $\tilde{t} = \mathrm{VSH}_n(\tilde{m})$.

□

Appendix D

Some Considerations With respect to Chapter 5

D.1 Considerations Concerning Section 5.1

D.1.1 Proof of claim 5.1.1

This proof is an alternative to the proof of this claim that can be found in section 2 of [12]. We will use Theorem 2.6.1 of [12], see Theorem B.1.8.

We will repeat the claim:

Claim 5.1.1. *For each integer $x \geq 2$ we have the following relation:*

$$\sum_{p \leq x} p \sim \frac{x^2}{2\log(x)}. \tag{D.1}$$

proof of claim 5.1.1: Define the function $u_p : \mathbb{N} \to \{0, 1\}$ as follows:

$$u_p(x) = \begin{cases} 1 & \text{if } x \text{ is prime} \\ 0 & \text{otherwise} \end{cases}.$$

Using Abel's identity (see Lemma B.1.7) we find

$$\sum_{p \leq x} p = \sum_{n=1}^{x} u_p(n)n = x\pi(x) - \int_2^x \pi(t)\ \mathrm{d}t$$

We will first consider the integral. Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = x/\log(x)$. Then

$$\lim_{x\to\infty} \frac{xf'(x)}{f(x)} = \lim_{x\to\infty} \frac{x\log x - x}{\log^2 x} \Big/ \frac{x}{\log x} = \lim_{x\to\infty} \frac{\log^2 x - \log x}{\log^2 x} = \lim_{x\to\infty} \left(1 - \frac{1}{\log x}\right) = 1.$$

Hence $f'(x)/f(x) \sim 1/x$.

By the prime number theorem and theorem B.1.8 it follows that

$$\int_2^x \pi(t)\ dt \sim \int_2^x \frac{t}{\log t}\ dt \sim \frac{x^2}{2\log x}.$$

Hence, we find

$$\lim_{x\to\infty}\frac{x\pi(x)-\int_2^x\pi(t)\mathrm{d}t}{\frac{x^2}{2\log x}}=\lim_{x\to\infty}\left(2\frac{x\pi(x)}{\frac{x^2}{\log x}}-\frac{\int_2^x\pi(t)\ \mathrm{d}t}{\frac{x^2}{2\log x}}\right)$$
$$=2\lim_{x\to\infty}\frac{\pi(x)}{\frac{x}{\log x}}-\lim_{x\to\infty}\frac{\int_2^x\pi(t)\ \mathrm{d}t}{\frac{x^2}{2\log x}}=2-1=1.$$

□

D.1.2 Problems with $\sim$

Note that for arbitrary functions $a(x), b(x), c(x)$ and $d(x)$ it generally holds that

$$\left.\begin{array}{l}a(x)\sim b(x)\\ c(x)\sim d(x)\end{array}\right\}\not\Rightarrow a(x)+c(x)\sim b(x)+d(x)$$

Example D.1.1. Define $f:\mathbb{R}\to\mathbb{R}$ by $f(x)=(-1)^{\lfloor x\rfloor}(x+1)$ and $g:\mathbb{R}\to\mathbb{R}$ by $g(x)=x$. Then

$$\lim_{x\to\infty}\frac{(-1)^{\lfloor x\rfloor}(x+1)}{(-1)^{\lfloor x\rfloor}x}=\lim_{x\to\infty}\frac{x+1}{x}=1$$

Hence $f(x)\sim(-1)^{\lfloor x\rfloor}x$ and $g(x)\sim x$. However,

$$\lim_{x\to\infty}\frac{(-1)^{\lfloor x\rfloor}(x+1)+x}{(-1)^{\lfloor x\rfloor}x+x}$$

is unpredictable. Therefore, $f(x)+g(x)\not\sim(-1)^{\lfloor x\rfloor}x+x$.

So we need to be careful when we want to approximate $F(x)$. Simplifying (5.13) further using claim 5.1.1 yields

$$F(K)=\sum_{i=0}^{l-1}r\left((p_i')_{(\beta)}-\frac{p_{ir}}{\beta\log 2}\right)+(K-lr)\left((p_l')_{(\beta)}-\frac{p_{lr}}{\beta\log 2}\right)+\sum_{i=1}^{K-1}\frac{p_i}{\beta\log 2}$$
$$\approx\sum_{i=0}^{l-1}r\left((p_i')_{(\beta)}-\frac{p_{ir}}{\beta\log 2}\right)+(K-lr)\left((p_l')_{(\beta)}-\frac{p_{lr}}{\beta\log 2}\right)+\frac{p_{K-1}^2}{32\log(2)\log(p_{K-1})}. \tag{D.2}$$

If we let $K\to\infty$ then the number of single precision calculations calculated by (5.10) will behave as $F(K)/K$. Using equation (D.2) we find

$$\frac{F(K)}{K}<\left(K(n)_{(\beta)}-r\sum_{i=0}^{l-1}\frac{p_{ir}}{\beta\log 2}+\frac{p_{K-1}^2}{32\log(2)\log(p_{K-1})}\right)\Big/\ K\to-\infty.$$

We found this using mathematical software. Similarly we find that using (5.13) that $F(K)\to\infty$.

D.2 The Optimal Running Time to Solve VSSR

This section summarizes how we can show according to [5] that if we can solve VSSR without factoring, with a more efficient algorithm than Algorithm 5.2.1, then we can improve the speed of Quadratic Sieve. Let B be the smoothness bound and x a residue mod n. If x is such that $y = x^2 \bmod n$ has least positive representative modulo n of order n^d, then we call x a *target residue*. If, moreover, y has largest prime factor $\leq B$, then we call x a smooth residue. The idea is roughly as follows:

- Suppose we can efficiently generate target residues.
- Write down the time that is needed to find one smooth residue.
- Conclude that the optimal time is attained when $d \approx 1/2$.

According to Subsection 1.4.5. of [9], one expects to consider u^u target residues in order to find one smooth residue, where

$$u = \frac{\log n^d}{\log B}.$$

Similarly to the analysis on page 4 of [6], one expect to need about B relations in order to factor n. Thus the expected time needed to factor n is given by

$$T(B) = u^u B(1 + o(1)) = e^{(1+o(1))[u \log u + \log B]}.$$

The optimal solution is implied by $[\log(T(B))]' = 0$. Now [5] shows that the optimal solution for the running time is given by

$$T_{\text{opt}}(B) = e^{(1+o(1))\sqrt{2d \log n \log\log n}}.$$

Substituting $d = 1/2$ gives the Quadratic Sieve running time. From the fact that there is no published algorithm that finds B smooth residues faster than the Quadric Sieve, it follows that, with the current state of the art of finding smooth residues, we cannot do better than setting $d = 1/2$, implying that the best algorithm to solve VSSR is given by Algorithm 5.2.1.

Appendix E

Source codes

E.1 Mathematica Source Codes

E.1.1 Calculating the number of SP-multiplications w.r.t. Section 4.4

Here, we present the source code of Mathematica that calculates the number of SP-multiplications is required by implementations 4.0.2, 4.2.2, 4.2.3, and 4.3.1.

- **This code calculates the number of SP-multiplications that is required for evaluating one round of PoP on an all one message block. We will be sloppy and write for example: "This function calculates the number of SP-multiplication required by Original Classic VSH".**

Straightforward, left to right, multiplication:

```
In[1]:= Array[Array[Prime[154 - #] &, # + 1] &, 152];

In[2]:= Plus @@ Array[Ceiling[Log[2^32, Times @@ Array[Prime[#] &, #]]] &, 152]

Out[2]= 2678
```

Straightforward, right to left, multiplication

```
In[3]:= Plus @@ Array[Ceiling[Log[2^32, Times @@ Array[Prime[154 - #] &, #]]] &, 152]

Out[3]= 3324
```

Precalculation of Classic PrimeLength

```
In[91]:= PrimeLength[L_, b_] := Module[{i, T, A},
           i = 1;
           A = {L[[1]]};
           T = {};
           While[i < Length[L],
            While[(If[i + 1 ≤ Length[L], Times @@ A * L[[i + 1]], ∞] < b ),
             i = i + 1;
             A = Join[A, {L[[i]]}];
            ];
            T = Join[T, {A}];
            A = {};
           ];
           T]
```

Precalculation of Classic TestVector

```
In[75]:= TestVector[L_, b_] := Module[{i, T, A},
          i = 1;
          A = {L[[1]]};
          T = {};
          While[i < Length[L],
           While[(Times @@ A * L[[-1]] < b && i < Length[L]),
            i = i + 1;
            A = Join[A, {L[[i]]}];
           ];
           T = Join[T, {A}];
           A = {};
          ];
          T]
```

This function calculates the number of SP-multiplications. It outputs both the number of SP-multiplication needed for evaluating $\prod_{i=1}^{k} p\_i$ and the result of the product.

```
In[79]:= SPProds[L_, b_, tg_] := Module[{x, y, O, T, P},
          O = 0;
          P = 1;
          If[tg == 0, O = Plus @@ Array[Ceiling[Log[b, Times @@ Array[L[[#]] &, #]]] &, Length[L] - 1]; P = Times @@ L;,
           For[i = 1, i ≤ Length[L], i++,
            O = O + Plus @@ Array[Ceiling[Log[b, Times @@ Array[L[[i]][[#]] &, #]]] &, Length[L[[i]]] - 1;];
           T = Array[Times @@ L[[#]] &, Length[L]];
           O = O + Plus @@ Array[Ceiling[Log[b, Times @@ Array[T[[#]] &, #]]] &, Length[L] - 1];
           P = Times @@ T;
          ];
          {O, P}]
```

This function claculates the number of SP-multiplications required by Treebased multiplication.

```
In[94]:= TreeProds[L_, b_] := Module[{x, O, T},
          x = Length[L];
          T = L;
          O = 0;
          While[x > 1,
           O = O + Plus @@ Array[Ceiling[Log[b, T[[#]]] * Ceiling[Log[b, T[[x - # + 1]]]] &, Floor[x/2]];
           For[i = 1, i <= Floor[x/2], i++,
            T[[i]] = T[[i]] * T[[x - i + 1]];
           ];
           x = Floor[(x + 1)/2];
          ];
          {O, T[[1]]}];
```

Calculate the number of SP-multiplications for Original Classic vsh, where k=153.

```
In[6]:= P1 = Array[Prime[#] &, 153];

In[19]:= SPProds[P1, 2^32, 0]

Out[19]= {2678,
          44159768317044272188382334777409207932311464552775745543544776074936365729284862784016885685167297653877398203010085977935815
           25761418091777184651252796897083296030758673487413296623696063666473800020633201352491926021275918043317732020498722592681634
           94767128233020577424744436183049068049018803979356829115364232915007768544148206289204116598313059937061581084679039853 0}
```

Calculating the number of SP-multiplications for Classic TestVector for k=153.

```
P2 = TestVector[P1, 2^32];

SPProds[P2, 2^32, 1]

{998,
 44159768317044272188382334777409207932311464552775745543544776074936365729284862784016885685167297653877398203010085977935815\
 25761418091777184651252796897083296030758673487413296623696063666473800020633201352491926021275918043317732020498722592681634\
 94767128233020577424744436183049068049018803979356829115364232915007768544148206289204116598313059937061581084679039853 0}
```

Calculating the number of SP-multiplications of Classic PrimeLength for k=153.

```
P3 = PrimeLength[P1, 2^32];

SPProds[P3, 2^32, 1]

{988,
 44159768317044272188382334777409207932311464552775745543544776074936365729284862784016885685167297653877398203010085977935815\
 25761418091777184651252796897083296030758673487413296623696063666473800020633201352491926021275918043317732020498722592681634\
 94767128233020577424744436183049068049018803979356829115364232915007768544148206289204116598313059937061581084679039853 0}
```

Calculate the number of SP-multiplications for Classic Treebased, where k=153.

```
TreeProds[P1, 2^32]

{1017,
 44159768317044272188382334777409207932311464552775745543544776074936365729284862784016885685167297653877398203010085977935815\
 25761418091777184651252796897083296030758673487413296623696063666473800020633201352491926021275918043317732020498722592681634\
 94767128233020577424744436183049068049018803979356829115364232915007768544148206289204116598313059937061581084679039853 0}
```

Calculate the number of SP-multiplications for Original Classic, where k=273.

```
In[96]:= PC = Array[Prime[#] &, 273];

In[97]:= SPProds[PC, 2^32, 0]

Out[97]= {9623,
 16387839763162561236903037285550222310081007101447288068230588107891953060749038186529424621086603991859727630011635856669044\
 25424850390843511765726121244465046563359766698570139372076159834380089805984046126041862284639480048397252271145038026985165\
 87605879796176088484633121836603360005041854402604661408015020544054088257342735372361873736564142205543361611551711845628619\
 80616909106325373426042805144746673932291425010163193770671216825449678609435901555549251601190527260449950186442383933165205\
 88493281898021300388402913040961514275930276481528512758579869098128551876039376811256828831698644698963611320923661063244449\
 23180300843552269008994588601600420428186261110691477908076125352096166110953063005443405896284532143119953065789311 90}
```

Calculate the number of SP-multiplications for Classic TestVector, where k=273.

```
In[106]:= PC2 = TestVector[PC, 2^32];

In[100]:= SPProds[PC2, 2^32, 1]

Out[100]= {3722,
 16387839763162561236903037285550222310081007101447288068230588107891953060749038186529424621086603991859727630011635856669044\
 25424850390843511765726121244465046563359766698570139372076159834380089805984046126041862284639480048397252271145038026985165\
 87605879796176088484633121836603360005041854402604661408015020544054088257342735372361873736564142205543361611551711845628619\
 80616909106325373426042805144746673932291425010163193770671216825449678609435901555549251601190527260449950186442383933165205\
 88493281898021300388402913040961514275930276481528512758579869098128551876039376811256828831698644698963611320923661063244449\
 23180300843552269008994588601600420428186261110691477908076125352096166110953063005443405896284532143119953065789311 90}
```

Calculate the number of SP-multiplications for Classic PrimeLength, where k=273.

```
In[107]:= PC3 = PrimeLength[PC, 2^32];

In[102]:= SPProds[PC3, 2^32, 1]

Out[102]= {3541,
 1638783976316256123690303728555022231008100710144728806823058810789195306074903818652942462108660399185972763001163585666904457
  2542485039084351176572612124446504656335976669857013937207615983438008980598404612604186228463948004839725227114503802698516
  8760587979617608848463312183660336000504185440260466140801502054405408825734273537236187373656414220554336161155171184562861
  8061690910632537342604280514474667393229142501016319377067121682544967860943590155554925160119052726044995018644238393316520
  8849328189802130038840291304096151427593027648152851275857986909812855187603937681125682883169864469896361132092366106324444
  2318030084355226900899458860160042042818626111069147779080761253520961661109530630054434058962845321431199530657893119 0}
```

Calculate the number of SP-multiplications of Classic Treebased, where k=273.

```
In[103]:= TreeProds[PC, 2^32]

Out[103]= {3534,
 1638783976316256123690303728555022231008100710144728806823058810789195306074903818652942462108660399185972763001163585666904457
  2542485039084351176572612124446504656335976669857013937207615983438008980598404612604186228463948004839725227114503802698516
  8760587979617608848463312183660336000504185440260466140801502054405408825734273537236187373656414220554336161155171184562861
  8061690910632537342604280514474667393229142501016319377067121682544967860943590155554925160119052726044995018644238393316520
  8849328189802130038840291304096151427593027648152851275857986909812855187603937681125682883169864469896361132092366106324444
  2318030084355226900899458860160042042818626111069147779080761253520961661109530630054434058962845321431199530657893119 0}
```

E.1.2 Creating the figures of subsection 5.1

Formulas

```
P[X_] := If[X == 0, 0, Prime[X]];
nn =
  14793758366415191538168947928202797686992186012184912034877181729077790117942781610958374291428504990254453945614073077660724 4
   93683782121275790319055815524259474347114305806633585641312504607762480684608497321331821652386163183853202897726894921441837 5
   29010164789049821460734731855993471320241871155287097637581668607216097454545694620916080381743403812547468928155125409307;
FindK[n_] := Module[{i},
  i = 0;
  While[(Times @@ Array[P[#] &, i + 1]) < n, i = i + 1];
  i]
k = FindK[nn]
$TextStyle = {FontFamily → "Times", FontSize → 25}
P2[X_] := If[X < 1, 0, If[X < 2, 2, If[X < 3, 3, X Log[X]]]];
Pcalc2ext[X_, Y_] := Ceiling[0.25 * Plus @@ Array[(P2[# + Y] - P2[Y])/(32 Log[2]) &, X]]
Pcalc5[X_] := 1/8 X^2 Log[X]/(32 Log[2]);
Modcalc[X_, Y_] := (Y Log[Y]/(64 Log[2]) + X/32) * (X/32 + 3) + ((X/32)^2 + (X/32))/2 + (Y Log[Y]/(64 Log[2])) (X/32)
Modcalcd[X_, Y_] := Y Log[Y]/(64 Log[2]) + X/32;
Modcalcnnaive[X_, Y_] := ((Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]) + X/32) * (X/32 + 3) + ((X/32)^2 + (X/32))/2 +
  (Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]) (X/32)
Modcalcnnaived[X_, Y_] := (Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]) + X/32
Modcalcnrealistic[X_, Y_] := (Min[{Y Log[Y]/(64 Log[2]), X/32}] + X/32) (X/32 + 3) + ((X/32)^2 + (X/32))/2 + Min[{Y Log[Y]/(64 Log[2]), X/32}] (X/32)
Modcalcnrealisticd[X_, Y_] := Min[{Y Log[Y]/(64 Log[2]), X/32}] + X/32;
Modcalcnnormalrealistic[X_, Y_] :=
 (Min[{Y Log[Y]/(64 Log[2]), X/32}] + If[Min[{Y, 2 k}] == 2 k, (Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]), 0] + X/32) (X/32 + 3) +
 ((X/32)^2 + (X/32))/2 + (Min[{Y Log[Y]/(64 Log[2]), X/32}] + If[Min[{Y, 2 k}] == 2 k, (Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]), 0]) (X/32)
Modcalcnnormalrealisticd[X_, Y_] := (Min[{Y Log[Y]/(64 Log[2]), X/32}] + If[Min[{Y, 2 k}] == 2 k, (Mod[Y, 2 k] + 1) Log[(Mod[Y, 2 k] + 1)]/(64 Log[2]), 0]) + X/32;
Roundsnum[X_, Y_] := Ceiling[(X * 2^23)/Y];
Totalcost24n[X_, Y_] := Roundsnum[X, Y] * (Pcalc5[Y] + Modcalc[1234, Y]);
Totalcost24nd[X_, Y_] := Roundsnum[X, Y] * (Modcalcd[1234, Y]);
Totalcost3nnaive[X_, Y_] := Roundsnum[X, Y] * (PcalcRedNaive[Y, 1234] + Modcalcnnaive[1234, Y]);
Totalcost3nnaived[X_, Y_] := Roundsnum[X, Y] * (PcalcRedNaived[Y] + Modcalcnnaived[1234, Y]);
Totalcost3nrealistic[X_, Y_] := Roundsnum[X, Y] * (PcalcRedRealistic[Y, 1234] + Modcalcnrealistic[1234, Y]);
Totalcost3nrealisticd[X_, Y_] := Roundsnum[X, Y] * (PcalcRedRealisticd[Y] + Modcalcnrealisticd[1234, Y]);
Totalcost3nnormalrealistic[X_, Y_] := Roundsnum[X, Y] * (PcalcRedNormalRealistic[Y, 1234] + Modcalcnnormalrealistic[1234, Y]);
Totalcost3nnormalrealisticd[X_, Y_] := Roundsnum[X, Y] * (PcalcRedNormalRealisticd[Y, 1234] + Modcalcnnormalrealisticd[1234, Y]);
PcalcRedNaive[X_, Y_] := Module[{i, j, outp},
  i = Floor[X/(2 k)];
  j = Mod[X, 2 k];
  outp = Plus @@ Array[Pcalc2ext[2 k, (# - 1) * 2 k] &, i];
  outp = outp + Pcalc2ext[j, i * 2 k];
  outp = outp + i ((Y/32 + 3));
  outp]
```

```
PcalcRedNaived[X_] := Floor[X/(2 k)]

PcalcRedNormalRealistic[X_, Y_] := Module[{i, j, outp},
  i = Floor[X/(2 k)];
  j = Mod[X, 2 k];
  outp = Plus @@ Array[Pcalc2ext[2 k, (# - 1) * 2 k] &, i];
  outp = outp + Pcalc2ext[j, i * 2 k];
  outp = outp + i ((Y/32 + 3));
  outp = outp + Max[(i - 1), 0] (Y/32)^2;
  outp = outp + Max[(i - 1), 0] (Y/32) (Y/32 + 3);
  outp]
PcalcRedNormalRealisticd[X_, Y_] := Module[{i, j, outp},
  i = Floor[X/(2 k)];
  outp = i;
  outp = outp + Max[(i - 1), 0] Y/32;
  outp]
PcalcRedRealistic[X_, Y_] := Module[{i, j, outp},
  outp = Pcalc2ext[Min[{X, 2 k}], 0];
  outp = outp + 0.5 Max[{X - 2 k, 0}] (Y/32);
  outp = outp + 0.5 Max[{X - 2 k, 0}] (Y/32 + 3);
  outp
 ]
PcalcRedRealisticd[X_] := Max[{X - 2 k, 0}]/2;
```

No Reduction

```
ANR = Array[Totalcost24n[1, # + 2] &, 1000];
HNR1 = ListPlot[Array[{# + 2, ANR[[#]]} &, 1000], AxesLabel → {"# primes", "# single precision multiplications"},
   AxesOrigin → {0, 0.15 * 10^9}, PlotRange → {0.15 * 10^9, 0.6 * 10^9}, PlotStyle → GrayLevel[0.5]];
AANR = Array[Totalcost24nd[1, # + 2] &, 1000];
HNRD1 = ListPlot[Array[{# + 2, AANR[[#]]} &, 1000], AxesLabel → {"# primes", "# single precision divisions"},
   AxesOrigin → {0, 0}, PlotRange → {0, 9 * 10^6}, PlotStyle → GrayLevel[0.5]];
```

Realistic Reduction

```
ARD = Array[Totalcost3nrealistic[1, # + 50] &, 950];
HRD1 = ListPlot[Array[{# + 50, ARD[[#]]} &, 950], AxesLabel → {"# primes", "# single precision multiplications"},
   AxesOrigin → {0, 0.15 * 10^9}, PlotRange → {0.15 * 10^9, 0.5 * 10^9}, PlotStyle → GrayLevel[0.5]];
AARD = Array[Totalcost3nrealisticd[1, # + 2] &, 1000];
HRDD1 = ListPlot[Array[{# + 2, AARD[[#]]} &, 1000], AxesLabel → {"# primes", "# single precision divisions"},
   AxesOrigin → {0, 0}, PlotRange → {0, 9 * 10^6}, PlotStyle → GrayLevel[0.5]];
```

'Normal' Realistic Reduction

```
ANRRD = Array[Totalcost3nnormalrealistic[1, 2 # + 50] &, 475];
HNRD1 = ListPlot[Array[{2 # + 50, ANRRD[[#]]} &, 475], AxesLabel → {"# primes", "# single precision multiplications"},
   AxesOrigin → {0, 0.15 * 10^9}, PlotRange → {0.15 * 10^9, 0.6 * 10^9}];
AANRD = Array[Totalcost3nnormalrealisticd[1, # + 2] &, 1000];
HNRDD1 = ListPlot[Array[{# + 2, AANRD[[#]]} &, 1000], AxesLabel → {"# primes", "# single precision divisions"},
   AxesOrigin → {0, 0}, PlotRange → {0, 9 * 10^6}];
ANG = Array[Totalcost3nnormalrealistic[1, 20 # + 50] &, 475];
HNG1 = ListPlot[Array[{20 # + 50, ANG[[#]]} &, 475], AxesLabel → {"# primes", "# single precision multiplications"},
   AxesOrigin → {0, 0.15 * 10^9}, PlotRange → {0.1 * 10^9, 0.6 * 10^9}];
AANRDG = Array[Totalcost3nnormalrealisticd[1, 10 # + 2] &, 1000];
HNRDGD1 = ListPlot[Array[{10 # + 2, AANRDG[[#]]} &, 1000], AxesLabel → {"# primes", "# single precision divisions"},
   AxesOrigin → {0, 0}, PlotRange → {0, 9 * 10^6}];
```

E.1.3 Solving $U^U = L(S)$

A 1024 bit RSA modulus:

```
nn =
 173675322711590738683931267916410894090197319389835419651766488635183599756509861062611990251640908486363958296130082354333480625792547712336608322042304379747057399644955445259617239035492777696941437086010226158179549348892401751889452919569848350117622925638114878421587926756118835262808794276372454204431;
```

Bit number check:

```
N[Log[2, nn]]
```

```
1023.95
```

Complexity function for NFS:

```
T[n_, α_] := Exp[(1.923 + α) (Log[n])^(1/3) (Log[Log[n]])^(2/3)]
```

Complexity function for best algorithm for breaking VSSR is u^u, where $u = \frac{\text{Log}[\sqrt{nn}]}{\text{Log}[B]}$. B is the largest small prime.

The algorithm is roughly as follows:

-1) Find $y = x^2 \bmod n$, where $y \sim \sqrt{nn}$

-2) Try to factor y in small primes such that the largest small prime is not bigger than B.

```
NewtonZero[H_, xzero_, ϵ_] := Module[{x, δ, rounds},
  MAXROUNDS = 10^5;
  x = xzero;
  δ = N[Abs[H[x]], 1000];
  rounds = 0;
  While[δ > ϵ && rounds < MAXROUNDS,
   x = N[x - H[x]/H'[x], 1000];
   δ = N[Abs[H[x]], 1000];
   rounds = rounds + 1;
  ];
  {rounds, x}]
```

We want to find the largest B such that $u^u = T[nn, \alpha]$. As u^u is obviously increasing in u (and decreasing in B), we will use the simple Newtons method to find the zero of:

```
F[x_, n_, α_] := x^x - T[n, α]
```

```
NewtonZero[F[#, nn, 0] &, 2, 10^-50]
```

```
General::ovfl : Overflow occurred in computation. More…
General::ovfl : Overflow occurred in computation. More…
General::ovfl : Overflow occurred in computation. More…
General::stop : Further output of General::ovfl will be suppressed during this calculation. More…
{2, Indeterminate}
```

Apparently we get an overflow trying this F. Obviously as F is strictly increasing in x, if we take n and α constant. So we may consider the logarithms i.e.:

```
FF[x_, n_, α_] := Log[x^x] - Log[T[n, α]]
```

```
opt = NewtonZero[FF[#, nn, 0] &, 2, 10^-50]
```

```
{5, 20.0565}
```

The relative error equals:

```
F[opt[[2]], nn, 0]
T[nn, 0]
```

$$2.87596 \times 10^{-15}$$

which is negligible. So from this we calculate $B = \text{Exp}[\frac{\text{Log}[\sqrt{nn}]}{\text{opt}[[2]]}]$

```
B = Exp[Log[Sqrt[nn]]/opt[[2]]]
```

$$4.83381 \times 10^{7}$$

The following module will find the index of the prime that is $\leq$ than B.

```
PrimeIndex[X_] := Module[{i}, i = 1; While[Prime[i] < X, i = i + 1]; i - 1]
```

So the index of the largest prime $\leq B$ is

```
PI = PrimeIndex[B]
```

```
2907207
```

check:

```
Prime[PI] < B < Prime[PI + 1]
```

```
True
```

```
N[Log[2, PI]]
```

```
21.4712
```

In conclusion: The best algorithm known for breaking VSSR on nn has about the same running time as the Number Field Sieve to factor nn when VSSR is concerned with $2907207 \approx 2^{21.4712}$ small primes. In other words to have 1024-bit RSA security in VSH we can use at most 2907207 small primes.

It seems (I have to look closely to it) that these calculations are accurate for all B such that $\sqrt{n} \geq B \geq \mathrm{Log}[\sqrt{n}]^{1+\epsilon}$.

```
PrimeIndex[N[Log[√nn]]]
```

```
71
```

So (taken $\epsilon = 0$) from the 71-th small prime the calculation is accurate. Therefore we may conlcude that these calculations are accurate. (See bottom of Page 6 of Igors and Ians Paper :"*Statistical Distribution and Collisions of the VSH*")

E.2 C-source code

This Appendix presents the source codes of the different variants of VSH. Let A and B be multiple precision integers. So A and B are declared as `mpz_t` variables. In our discussion we used the following notation:

- $A = B *_{\text{gmp}} C \Leftrightarrow$`mpz_mul(A,B,C)` if C is a multiple precision (`mpz_t`) integer and $A = B *_{\text{gmp}} C \Leftrightarrow$`mpz_mul_ui` if C is declared as a single precision integer (`int`,`short` or `long`).
- $A = B \mod_{\text{gmp}} C \Leftrightarrow$`mpz_mod(A,B,C)`.
- $B >_{\text{gmp}} C \Leftrightarrow$`mpz_cmp(B,C)>0`.
- The bit-size of a `mpz_t` value A is returned by `mpz_sizeinbase(A,2)`.

See section 5 of [14] for a complete list of all integer functions of GMP.

E.2.1 Source Codes of the Ideas of Chapter 4

This section provides the source codes of the iteration functions of VSH based on the ideas discussed in Chapter 4. Here, `counter` denotes the iteration number, say j. Then, `input` denotes x_j and `output` denotes x_{j+1}. The primes $p_1, \cdots, p_k$ are stored in `primes[]`, k is denoted by `k` and n, the modulus, by `n`. Finally, `temp` denotes the temporary value t and `starting_word` and `starting_bit` denote the position of the message bit, where this iteration round should start reading the message.

Classic-VSH

Code of Pseudo code 4.0.2

```
void iterate(mpz_t output, mpz_t input, unsigned char *message, int
counter, mpz_t n, unsigned short k, int *primes, mpz_t temp){
    int i, j;
    int starting_word = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;
    mpz_set_ui(temp,1);

    /*Calculate the product of small primes*/
    j = starting_bit;
    i = starting_word;

    while((8*i+j) < ((counter+1)*k)) {
        if( ((message[i] >> (7-j)) & 1)==1)
           mpz_mul_ui(temp,temp,primes[8*(i-starting_word)+(j-starting_bit)]);
        j++;
        if(j==8){
            j=0;
            i++;
        }
    }
```

```
    /*perform the remaining calculation steps of an iteration*/
    mpz_mul(output,input,input);
    mpz_mod(output, output, n);
    mpz_mul(output,output, temp);
    mpz_mod(output, output, n);
}
```

code of Pseudo code 4.2.2

```
void calculate_primes_product(mpz_t temp, unsigned char *message,
int counter, unsigned short k, int *primes, unsigned int mask)
{
    static unsigned int products[10000];
    int i, j, w;
    unsigned int t;
    int starting_byte = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;
    int end;

    /*Calculate the product of small primes*/
    j = starting_bit;
    i = starting_byte;
    /* number of words: */
    w = 0;
    /* temporary 32-bit product: */
    t = 1;
    end = (counter+1)*k;

    while( (8*i+j) < end ) {

        /* fit as many small prime products as we can into a single word */
        do {
            if ((message[i] >> (0x7^j)) & 1) {
                t = t * primes[8*(i-starting_byte)+(j-starting_bit)];
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
                if (t & mask)
                    /* danger of overflow: break out of loop */
                    break;
            }
            else {
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            }
        } while ((8*i+j) < end );
```

```
            /* store the product word: */
            products[w] = t;
            /* reset small product word: */
            t = 1;
            w++;
        }

        /* now multiply through all the small product words: */
        for(i=0;i<w;i++){
            mpz_mul_ui(temp, temp, products[i]);
        }
    }
```

code of Pseudo code 4.2.3

```
    void calculate_primes_product(mpz_t temp, unsigned char *message,
    int counter, unsigned short k, int *primes, int *primes_length) {
        static unsigned int products[10000];
        int i, j, w, v;
        unsigned int t;
        int starting_byte = ((counter*k) >> 3);
        char starting_bit = (counter*k) & 7;
        int end;

        /*Calculate the product of small primes*/
        j = starting_bit;
        i = starting_byte;
        /* number of words: */
        w = 0;
        /* intermediate bit-length */
        v = 0;
        /* temporary 32-bit product: */
        t = 1;
        end = (counter+1)*k;

        while( (8*i+j) < end ) {

            /* fit as many small prime products as we can into a single word */
            do {
                if ((message[i] >> (0x7^j)) & 1) {
                    if ((v+primes_length[8*(i-starting_byte)+(j-starting_bit)])>32)
                        /* danger of overflow: break out of loop */
                        break;
                    t = t * primes[8*(i-starting_byte)+(j-starting_bit)];
                    v+= primes_length[8*(i-starting_byte)+(j-starting_bit)];

                    j++;
                    if(j==8) {
                        j=0;
                        i++;
                    }
                }
```

```
            else {
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            }
        } while ((8*i+j) < end );

        /* store the product word: */
        products[w] = t;
        /* reset small product word: */
        v = 0;
        t = 1;
        w++;
    }

    /* now multiply through all the small product words: */
    for(i=0;i<w;i++){
        mpz_mul_ui(temp, temp, products[i]);
    }
}
```

Code of Pseudo code 4.3.1

```
#define     MAX_PRIMES      1000

void calculate_primes_product(mpz_t temp, unsigned char *message,
int counter, unsigned short k, int *primes) {
    static unsigned int array[MAX_PRIMES];
    int i, j, num;
    unsigned int t;
    int starting_byte = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;
    int end;
    static int firstcall = 1;
    static mpz_t prods[MAX_PRIMES/2];

    if (firstcall) {
        /* initialize products */
        firstcall = 0;
        for (i = 0; i < MAX_PRIMES/2; ++i)
            mpz_init(prods[i]);
    }

    /*Calculate the product of small primes*/
    j = starting_bit;
    i = starting_byte;
    /* number of array elements: */
    num = 0;
    end = (counter+1)*k;
```

```
        while( (8*i+j) < end ) {
            /* store the primes we need in  array  */
            if ((message[i] >> (0x7^j)) & 1) {
                array[num++] = primes[8*(i-starting_byte)+(j-starting_bit)];
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            } else{

                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
        }
        }

        if (num == 0)
            return;

        /* process the first round of multiplies by pairing the big primes
           with smaller ones, and saving the result within  prods .  Assume
           that there is no overflow.
         */

        for (i=0; i < num/2; ++i){
            mpz_set_ui(prods[i], array[i]*array[num-1-i]);
        }
        if (num&1)
            mpz_set_ui( prods[i], array[i] );
        /* number of elements in  prods  is: */
        num = (num+1)/2;

        /* now perform remaining products: */
        while (num > 1) {
            for (i=0; i < num/2; ++i) {
                mpz_mul(prods[i], prods[i], prods[num-1-i]);

            }
            num = (num+1)/2;

        }

        mpz_set(temp, prods[0]);
    }
```

E.2.2 Fast-VSH

Code of Pseudo code 4.0.3

```
void iterate(mpz_t output, unsigned char *m, mpz_t n, unsigned short
size, int *primes, int message_length, int i){
  static int firstcall=1;
  static mpz_t temp;
  if (firstcall) {
     mpz_init(temp);
     firstcall = 0;
  }
    long j;

    mpz_mul(output, output, output);
    mpz_mod(output, output, n);

    mpz_set_si(temp,1);
    for (j=0;j<size;j++) {
      if (((i+j)<(message_length))) {
        mpz_mul_ui(temp,temp,primes[((j<<8) + m[i+j])]);
        if (mpz_cmp(temp,n)>=0) {

      mpz_mul(output, output, temp);
      mpz_mod(output, output, n);

          mpz_set_si(temp,1);
        }
      }
    }
      if (mpz_cmp_si(temp,1) > 0)
      mpz_mul(output, output, temp);
      mpz_mod(output, output, n);
}
```

Code of Pseudo code 4.2.4

```
void iterate(mpz_t output, mpz_t input, unsigned char *message, int
counter, mpz_t n, unsigned short k, unsigned short cs, int *primes){
    int i, j, w, current_byte;
    static int firstcall = 1;
    static mpz_t temp;
    static mpz_t temp_array[ARRAYLENGTH];

    if(firstcall) {
       firstcall = 0;
       mpz_init(temp);
       for(i = 0; i < ARRAYLENGTH; i++){
        mpz_init_set_ui(temp_array[i],1);
       }
       limit = (mpz_sizeinbase(n,2)/18) + 1;
    }

    w = 0;
    j = 0;
        mpz_set_ui(temp,1);
```

```
    current_byte = k*counter;

    /*Calculate the product of small primes*/

    for(i=0; i < k; i++) {
       mpz_mul_ui(temp, temp, primes[(i << cs)+message[current_byte+i]]);

       if(j++ == limit) {
        j = 0;
        mpz_mod(temp_array[w++], temp, n);
        mpz_set_ui(temp,1);
       }

    }

    for(i = 0; i < w; i++) {
       mpz_mul(temp, temp, temp_array[i]);
       mpz_mod(temp, temp, n);
    }


    /*perform the remaining calculation steps of an iteration*/
    mpz_mul(output,input,input);
    mpz_mod(output,output,n);
    mpz_mul(output,output, temp);
    mpz_mod(output,output,n);
}
```

Code of Pseudo code 4.3.1

```
#define     MAX_PRIMES      1000 #define     ARRAYLENGTH     1000

void calculate_primes_product(mpz_t temp, unsigned char *message,
int counter, unsigned short k, unsigned short cs, int *primes, mpz_t
n) {
    int i, w, num, current_byte;
    unsigned int t;
    static int firstcall = 1;
    static mpz_t prods[MAX_PRIMES];
    static mpz_t temp_array[ARRAYLENGTH];

    if (firstcall) {
        /* initialize products */
        firstcall = 0;
        for (i = 0; i < MAX_PRIMES; ++i)
            mpz_init(prods[i]);
        for (i = 0; i < ARRAYLENGTH; ++i)
            mpz_init(temp_array[i]);
    }

    /*Calculate the product of small primes*/
    current_byte = counter*k;
```

```
    i = 0;
    w = 0;
    /* number of array elements: */
    num = k;

    for(i = 0; i<k; ++i) {
           /* store the primes we need in  array  */
           mpz_set_ui(prods[i], primes[(i << cs) + message[current_byte + i] ]);
    }

    /* now perform products: */
    while (num > 1) {
        for (i=0; i < num/2; ++i){
           mpz_mul(prods[i], prods[i], prods[num-1-i]);
           if(mpz_cmp(prods[i],n) > 0){
              mpz_mod(temp_array[w++],prods[i],n);
              mpz_set_ui(prods[i],1);
           }
        }
        num = (num+1)/2;
    }
    mpz_set(temp, prods[0]);

    for(i = 0; i < w; i++){
       mpz_mul(temp,temp,temp_array[i]);
       mpz_mod(temp,temp, n);
    }
}
```

E.2.3 C Source Codes for the Altered Pseudo codes of Classic-VSH w.r.t. Chapter 5

Code of Pseudo code 5.1.4

```
void iterate(mpz_t output, mpz_t input, unsigned char *message, int
counter, mpz_t n, unsigned short k, int *primes, mpz_t temp, mpz_t
temp_array[]){
    int i, j,w;
    int starting_byte = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;
    mpz_set_ui(temp,1);

    /*Calculate the product of small primes*/
    j = starting_bit;
    i = starting_byte;
    w = 0;

    while((8*i+j) < ((counter+1)*k)){
        if( ((message[i] >> (7-j)) & 1)==1) {
            mpz_mul_ui(temp,temp,primes[8*(i-starting_byte)+(j-starting_bit)]);
            if(mpz_cmp(temp, n) > 0) {
                mpz_mod(temp_array[w++],temp,n);
```

```
                mpz_set_ui(temp,1);
            }
        }
        j++;
        if(j==8){
            j=0;
            i++;
        }
    }

    for(i=0;i<w; i++){
        mpz_mul(temp,temp_array[i], temp);
        mpz_mod(temp,temp, n);
    }

    /*perform the remaining calculation steps of an iteration*/
    mpz_mul(output,input,input);
    mpz_mod(output, output, n);
    mpz_mul(output,output, temp);
    mpz_mod(output, output, n);

}
```

code of Pseudo code 4.2.2

```
void calculate_primes_product(mpz_t temp, unsigned char *message,
int counter, mpz_t n, unsigned short k, int *primes, unsigned int
mask, mpz_t temp_array[]) {
    static unsigned int products[50000];
    int i, j, w, v, gmpw;
    unsigned int t;
    int starting_byte = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;
    int end;

    /*Calculate the product of small primes*/
    j = starting_bit;
    i = starting_byte;
    /* number of words: */
    w = 0;
    gmpw = 0;
    /* intermediate bit-length */
    v = 0;
    /* temporary 32-bit product: */
    t = 1;
    end = (counter+1)*k;

    while( (8*i+j) < end ) {

        /* fit as many small prime products as we can into a single word */
        do {
            if ((message[i] >> (0x7^j)) & 1) {
```

```
                t = t * primes[8*(i-starting_byte)+(j-starting_bit)];
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
                if (t & mask)
                    /* danger of overflow: break out of loop */
                    break;
            }
            else {
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            }
        } while ((8*i+j) < end );

        /* store the product word: */
        products[w] = t;
        /* reset small product word: */
        t = 1;
        w++;
    }
    /*fill gmp array of words of same length as n*/
    for(i=0;i<w;i++){
        mpz_mul_ui(temp, temp, products[i]);
        if(mpz_cmp(temp, n) > 0) {
           mpz_mod(temp_array[gmpw++],temp,n);
           mpz_set_ui(temp,1);
        }
    }

    /*now multiply through all the small product words:*/
    for(i=0;i<gmpw; i++){
       mpz_mul(temp,temp_array[i], temp);
       mpz_mod(temp,temp, n);
    }

}

void iterate(mpz_t output, mpz_t input, unsigned char *message, int
counter, mpz_t n, unsigned short k, int *primes, unsigned int
mask, mpz_t temp, mpz_t temp_array[]){
    mpz_set_ui(temp,1);

    calculate_primes_product(temp, message, counter, n, k, primes,
      mask, temp_array);

    /*perform the remaining calculation steps of an iteration*/
    mpz_mul(output,input,input);
    mpz_mod(output, output, n);
```

```
    mpz_mul(output,output, temp);
    mpz_mod(output, output, n);
}
```

code of Pseudo code 4.2.3

```
void calculate_primes_product(mpz_t temp, unsigned char *message,
int counter, mpz_t n, unsigned short k, int *primes, int
*primes_length, mpz_t temp_array[]) {
    static unsigned int products[50000];
    int i, j, w, v, gmpw;
    unsigned int t;
    int starting_byte = ((counter*k) >> 3);
    int end;

    /*Calculate the product of small primes*/
    j = 0;
    i = starting_byte;
    /* number of words: */
    w = 0;
    gmpw = 0;
    /* intermediate bit-length */
    v = 0;
    /* temporary 32-bit product: */
    t = 1;
    end = (counter+1)*k;

    while( (8*i+j) < end ) {

        /* fit as many small prime products as we can into a single word */
        do {
            if ((message[i] >> (0x7^j)) & 1) {
            if ( (v+primes_length[8*(i-starting_byte)+j])>32)
                    /* danger of overflow: break out of loop */
                    break;
                t = t * primes[8*(i-starting_byte)+j];
                v += primes_length[8*(i-starting_byte)+j];

                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            }
            else {
                j++;
                if(j==8) {
                    j=0;
                    i++;
                }
            }
        } while ((8*i+j) < end );
```

```
            /* store the product word: */
            products[w] = t;
            /* reset small product word: */
            v = 0;
            t = 1;
            w++;
        }

        /*fill gmp array of words of same length as n*/
        for(i=0;i<w;i++){
            mpz_mul_ui(temp, temp, products[i]);
            if(mpz_cmp(temp, n) > 0) {
               mpz_mod(temp_array[gmpw++],temp,n);
               mpz_set_ui(temp,1);
            }
        }

        /*now multiply through all the small product words:*/
        for(i=0;i<gmpw; i++){
           mpz_mul(temp,temp_array[i], temp);
           mpz_mod(temp,temp, n);
        }

    }

    void iterate(mpz_t output, mpz_t input, unsigned char *message, int
    counter, mpz_t n, unsigned short k, int *primes, int *primes_length,
    mpz_t temp, mpz_t temp_array[]) {
        mpz_set_ui(temp,1);

        calculate_primes_product(temp, message, counter, n, k, primes,
          primes_length, temp_array);

        /*perform the remaining calculation steps of an iteration*/
        mpz_mul(output,input,input);
        mpz_mod(output, output, n);
        mpz_mul(output,output, temp);
        mpz_mod(output, output, n);
    }
```

Code of Pseudo code 4.3.1

```
    void calculate_primes_product(mpz_t temp, unsigned char *message,
    int counter, unsigned int k, int *primes, mpz_t n) {
        static unsigned int array[MAX_PRIMES];
        int i, j, num;
        unsigned int t;
        int starting_byte = ((counter*k) >> 3);
        char starting_bit = (counter*k) & 7;
        int end;
        static int firstcall = 1;
```

```
static mpz_t prods[MAX_PRIMES/2];

if (firstcall) {
    /* initialize products */
    firstcall = 0;
    for (i = 0; i < MAX_PRIMES/2; ++i)
        mpz_init(prods[i]);
}

/*Calculate the product of small primes*/
j = 0;
i = starting_byte;
/* number of array elements: */
num = 0;
end = (counter+1)*k;


while( (8*i+j) < end ) {
    /* store the primes we need in  array  */
    if ((message[i] >> (0x7^j)) & 1) {
        array[num++] = primes[8*(i-starting_byte)+(j-starting_bit)];
        j++;
        if(j==8) {
            j=0;
            i++;
        }
    } else{

        j++;
        if(j==8) {
            j=0;
            i++;
        }
}
}

if (num == 0)
    return;

/* process the first round of multiplies by pairing the big
   primes with smaller ones, and saving the result within
   prods. Assume that there is no overflow.
 */

for (i=0; i < num/2; ++i) {
    mpz_set_ui(prods[i], array[i]*array[num-1-i]);
}
if (num&1)
    mpz_set_ui( prods[i], array[i] );
/* number of elements in  prods  is: */
num = (num+1)/2;

/* now perform remaining products: */
```

```
    while (num > 1) {
        for (i=0; i < num/2; ++i){
            mpz_mul(prods[i], prods[i], prods[num-1-i]);
            mpz_mod(prods[i], prods[i], n);
        }
        num = (num+1)/2;

    }

    mpz_set(temp, prods[0]);
}

void iterate(mpz_t output, mpz_t input, unsigned char *message, int
counter, mpz_t n, unsigned int k, int *primes, mpz_t temp) {

    mpz_set_ui(temp,1);

    calculate_primes_product(temp, message, counter, k, primes, n);

    /*perform the remaining calculation steps of an iteration*/
    mpz_mul(output,input,input);
    mpz_mod(output, output, n);
    mpz_mul(output,output, temp);
    mpz_mod(output, output, n);
}
```

E.2.4 Trapdoor-VSH (Chapter 6)

Code of Trapdoor VSH based on Pseudo code 6.1.1

```
void find_powers(mpz_t output[], mpz_t input[], unsigned char
*message, mpz_t phin, int counter, unsigned short k){

    int i, j;
    int starting_word = ((counter*k) >> 3);
    char starting_bit = (counter*k) & 7;

    j = starting_bit;
    i = starting_word;

    while((8*i+j) < ((counter+1)*k)){
       /*shift the exponents 1 bit to left*/
       mpz_mul_2exp(output[8*(i-starting_word)+(j-starting_bit)],
                    output[8*(i-starting_word)+(j-starting_bit)],1);
       /*add one if the corresponding message-bit is equal to one*/
       if( ((message[i] >> (7-j)) & 1)==1)
            mpz_add_ui(output[8*(i-starting_word)+(j-starting_bit)],
                       input[8*(i-starting_word)+(j-starting_bit)],1);
       /*If prime power is larger than phi(n), subtract phi(n)*/
       if(mpz_cmp(output[8*(i-starting_word)+(j-starting_bit)],phin)>0)
              mpz_sub(output[8*(i-starting_word)+(j-starting_bit)],
                       output[8*(i-starting_word)+(j-starting_bit)],phin);
       j++;
```

```
        if(j==8){
            j=0;
            i++;
        }
    }
}

void finalize(mpz_t output, mpz_t prime_power[], mpz_t n, unsigned
short k, int *primes){
    int i, j, power_bit_size;
    mpz_t temp;
    mpz_init_set_ui(temp,1);

    power_bit_size = mpz_sizeinbase(n,2);

    /*Calculate the product of small primes*/
    for(i = 1; i <= power_bit_size; i++){
        for(j = 0; j < k; j++){
           if(mpz_tstbit(prime_power[j], (power_bit_size - i)))
              mpz_mul_ui(temp,temp,primes[j]);
        }
        mpz_mul(output, output, output);
        mpz_mod(output, output, n);
        mpz_mul(output, output, temp);
        mpz_mod(output, output, n);
        mpz_set_ui(temp, 1);
    }
}
```

Code of Trapdoor VSH based on Pseudo code 6.1.2

```
/*
 * This procedure uses the trapdoor information ( phi(n) ) to speed
 * up computation of VSH hash.  The VSH computation can be written
 * as a product of  p_i ^ e_i  for exponents  e_i  which come from
 * the message bits.  This procedure computes the  e_i  modulo  phi(n),
 * and then  finalize()  procedure computes the product of the   p_i
 * modulo the reduced exponents.  Computing the exponents here is done
 * by calling  find_exps()  x times, where x = (message len in bits) / k .
 * The  counter  should be incremented every time.
 * WARNING: THIS PROCEDURE ASSUMES THE MESSAGE LENGTH (IN BITS) IS A
 *          MULTIPLE OF  k .  That should be the case assuming that
 *          padding has been applied as defined in the VSH paper.
 *
 * Description of function inputs and outputs:
 *
 *  prime_exps[]    array holding the current exponents (as much as has
 *                    been read in) for the primes  p_i .  This array
 *                    should be initialized to 0 before the first
 *                    time this procedure is called.  Ths array is
 *                    updated after processing  k  message bits.
 *  message[]       An array holding the entire message.
```

```
 *  phin            phi(n)  where   n  is the modulus.
 *  counter         number of blocks processed so far (so that we can
 *                    index into the right position of  message[] ).
 *                    A block is  k  bits of message.
 *  k               number of primes  p_i .
 *  finalcall       boolean telling whether or not we are processing the
 *                    final message block.
 */
void find_exps(mpz_t prime_exps[], unsigned char *message, mpz_t
phin,
  int counter, unsigned short k, short finalcall)
{ /* The exponents in this procedure are stored in the array
prime_exps[] .
 * In order to speed up computation, we use  tmp_exp[]  to hold up to
 * 32-bits of the current exponents being read in.  Once we have those
 * 32-bits (for all primes), we insert those into the  prime_exps[]
 * array with the proper shifting.  In this way, we only have to perform
 * mpz shift operations once in every 32 iterations (as opposed to every
 * iteration).  We also have to insert the  tmp_exp  exponents in the
 * final call to this procedure.
 *
 * Reduction of the exponents modulo  phi(n)  is done when either those
 * exponents reach approx twice the size of  phi(n) , or else during the
 * final call to this procedure.
 */
   static unsigned int tmp_exps[10000];      /*The intermediate ints*/
   /* number of bits currently stored in (each word of)  tmp_exps : */
   static int te_bits = 0;
   /* firstmpz  is a boolean: does  prime_exps  need to be initialized? */
   static int firstmpz = 1;
   /* When  prime_exps  has  limit  words, then perform mod reduction */
   static int limit;
   /* prime_exps_words  is number of 32-bit words currently stored in
    * each position of prime_exps[]: */
   static int prime_exps_words = 0;
   int i, j, l;
   int starting_word = ((counter*k) >> 3);
   int starting_bit = (counter*k) & 7;

   j = starting_bit;
   i = starting_word;

   if (te_bits == 0) {
       /* initialize  tmp_exps : we are processing a new 32-bits of
        * exponents (for each prime). */
       for(l = 0; l < k; l++)
           tmp_exps[l] = 0;
   }

   /* Calculate tmp_exps for prime 1 to l: */
   for(l = 0; l < k; l++){
       if(((message[i] >> (7-j)) & 1)==1) tmp_exps[l]++;
       j++;
```

```
            if(j==8) {
                j=0;
                i++;
            }
        }
        /* Increase the size of the  tmp_exps  by 1-bit */
        te_bits++;

        /* If we reached 32-bits in the  tmp_exps  array, or if it is the
         * final call, then place this array into  prime_exps  array.
         */
        if (te_bits==32 || finalcall) {
            prime_exps_words++;
            if (firstmpz) {
                firstmpz = 0;
                /* limit  is twice the number of 32-bit words as  phin : */
                limit = (mpz_sizeinbase(phin,2)>>4);
                for(l = 0; l < k; l++)
                    mpz_set_ui( prime_exps[l], tmp_exps[l] );
            }
            else
            {
                for(l = 0; l < k; l++) {
                    /* note -- te_bits  is 32 except when finalcall is 1 : */
                    mpz_mul_2exp( prime_exps[l], prime_exps[l], te_bits );
                    mpz_add_ui( prime_exps[l], prime_exps[l], tmp_exps[l] );
                    if( (prime_exps_words > limit) || finalcall)
                        mpz_mod( prime_exps[l], prime_exps[l], phin );
                }
                if (prime_exps_words > limit)
                    /* after mod reduction, elements of  prime_exps[]  are
                     * the same length as  phin : */
                    prime_exps_words = (mpz_sizeinbase(phin,2)>>5);
            }
            /* next iteration: reset the number of bits in  tmp_exps */
            te_bits = 0;
        }

        if(finalcall) {
            te_bits = 0;
            firstmpz = 1;
            prime_exps_words = 0;
        }
    }


    /*
     * After we have determine the exponents for our VSH computation
     * (from repeatedly calling   find_exps() ), we now compute the final hash.
     *
     * Description of function inputs and outputs:
     *
     *  hash        The final hash value.  Should be initialized to 1 by
```

```
 *                calling procedure.
 *  prime_exps  The exponents for the small primes, which were computed
 *                from repeatedly calling  find_exps() .
 *  n           The modulus.
 *  k           The number of small primes used in VSH.
 *  primes      The small primes used in VSH.
 */
void finalize(mpz_t hash, mpz_t prime_exps[], mpz_t n, unsigned
short k,
  int *primes)
{
    int i, j, exp_bits;
    mpz_t temp;

    exp_bits = mpz_sizeinbase(n,2);

    for(i = 1; i <= exp_bits; i++) {
        mpz_init_set_ui(temp,1);
        /* product of primes for the  i'th  bit from the 'left': */
        for(j = 0; j < k; j++) {
            if (mpz_tstbit(prime_exps[j], (exp_bits - i)))
                mpz_mul_ui(temp,temp,primes[j]);
        }
        mpz_mul(hash, hash, hash);
        mpz_mod(hash, hash, n);
        mpz_mul(hash, hash, temp);
        mpz_mod(hash, hash, n);
    }

}
```

Code of Trapdoor VSH based on Pseudo code 6.1.4

```
/*
 * This procedure uses the trapdoor information ( phi(n) ) to speed
 * up computation of VSH hash.  The VSH computation can be written
 * as a product of  p_i ^ e_i  for exponents  e_i  which come from
 * the message bits.  This procedure computes the  e_i  modulo  phi(n),
 * and then  finalize()  procedure computes the product of the   p_i
 * modulo the reduced exponents.  Computing the exponents here is done
 * by calling  find_exps()  x times, where x = (message len in bits) / k .
 * The  counter  should be incremented every time.
 * WARNING: THIS PROCEDURE ASSUMES THE MESSAGE LENGTH (IN BITS) IS A
 *          MULTIPLE OF  k .  That should be the case assuming that
 *          padding has been applied as defined in the VSH paper.
 *
 * Description of function inputs and outputs:
 *
 *  prime_exps[]    array holding the current exponents (as much as has
 *                    been read in) for the primes  p_i .  This array
 *                    should be initialized to 0 before the first
 *                    time this procedure is called.  Ths array is
```

```
 *                    updated after processing  k  message bits.
 *  message[]       An array holding the entire message.
 *  phin            phi(n)  where   n  is the modulus.
 *  counter         number of blocks processed so far (so that we can
 *                    index into the right position of  message[] ).
 *                    A block is  k  bits of message.
 *  k               number of primes  p_i .
 *  finalcall       boolean telling whether or not we are processing the
 *                    final message block.
 */
void find_exps(mpz_t prime_exps[], unsigned char *message, mpz_t
phin,
  int counter, unsigned short k, short finalcall)
{ /* The exponents in this procedure are stored in the array
prime_exps[] .
 * In order to speed up computation, we use  tmp_exp[]  to hold up to
 * 32-bits of the current exponents being read in.  Once we have those
 * 32-bits (for all primes), we insert those into the  prime_exps[]
 * array with the proper shifting.  In this way, we only have to perform
 * mpz shift operations once in every 32 iterations (as opposed to every
 * iteration).  We also have to insert the  tmp_exp  exponents in the
 * final call to this procedure.
 *
 * Reduction of the exponents modulo  phi(n)  is done when either those
 * exponents reach approx twice the size of  phi(n) , or else during the
 * final call to this procedure.
 */
   static unsigned int tmp_exps[10000];      /*The intermediate ints*/
   /* number of bits currently stored in (each word of)  tmp_exps : */
   static int te_bits = 0;
   /* firstmpz  is a boolean: does  prime_exps  need to be initialized? */
   static int firstmpz = 1;
   /* When  prime_exps  has  limit  words, then perform mod reduction */
   static int limit;
   /* prime_exps_words  is number of 32-bit words currently stored in
    * each position of prime_exps[]: */
   static int prime_exps_words = 0;
   int i, j, l;
   int starting_word = ((counter*k) >> 3);
   int starting_bit = (counter*k) & 7;

   j = starting_bit;
   i = starting_word;

   if (te_bits == 0) {
       /* initialize  tmp_exps : we are processing a new 32-bits of
        * exponents (for each prime). */
       for(l = 0; l < k; l++)
           tmp_exps[l] = 0;
   }

   /* Calculate tmp_exps for prime 1 to l: */
   for(l = 0; l < k; l++){
```

```
            tmp_exps[l] = (tmp_exps[l] << 1) | ((message[i] >> (7-j)) & 1);
            j++;
            if(j==8) {
                j=0;
                i++;
            }
        }
        /* Increase the size of the  tmp_exps  by 1-bit */
        te_bits++;

        /* If we reached 32-bits in the  tmp_exps  array, or if it is the
         * final call, then place this array into  prime_exps  array.
         */
        if (te_bits==32 || finalcall) {
            prime_exps_words++;
            if (firstmpz) {
                firstmpz = 0;
                /* limit  is twice the number of 32-bit words as  phin : */
                limit = (mpz_sizeinbase(phin,2)>>4);
                for(l = 0; l < k; l++)
                    mpz_set_ui( prime_exps[l], tmp_exps[l] );
            }
            else
            {
                for(l = 0; l < k; l++) {
                    /* note -- te_bits  is 32 except when finalcall is 1 : */
                    mpz_mul_2exp( prime_exps[l], prime_exps[l], te_bits );
                    mpz_add_ui( prime_exps[l], prime_exps[l], tmp_exps[l] );
                    if( (prime_exps_words > limit) || finalcall)
                        mpz_mod( prime_exps[l], prime_exps[l], phin );
                }
                if (prime_exps_words > limit)
                    /* after mod reduction, elements of  prime_exps[]  are
                     * the same length as  phin : */
                    prime_exps_words = (mpz_sizeinbase(phin,2)>>5);
            }
            /* next iteration: reset the number of bits in  tmp_exps */
            te_bits = 0;
        }

        if(finalcall) {
            te_bits = 0;
            firstmpz = 1;
            prime_exps_words = 0;
        }
    }


    /*
     * After we have determine the exponents for our VSH computation
     * (from repeatedly calling   find_exps() ), we now compute the final hash.
     *
     * Description of function inputs and outputs:
```

```
 *
 *  hash        The final hash value.  Should be initialized to 1 by
 *                calling procedure.
 *  prime_exps  The exponents for the small primes, which were computed
 *                from repeatedly calling  find_exps() .
 *  n           The modulus.
 *  k           The number of small primes used in VSH.
 *  primes      The small primes used in VSH.
 */
void finalize(mpz_t hash, mpz_t prime_exps[], mpz_t n, unsigned
short k,
  int *primes)
{
    int i, j, exp_bits;
    mpz_t temp;

    exp_bits = mpz_sizeinbase(n,2);

    for(i = 1; i <= exp_bits; i++) {
        mpz_init_set_ui(temp,1);
        /* product of primes for the  i'th  bit from the 'left': */
        for(j = 0; j < k; j++) {
            if (mpz_tstbit(prime_exps[j], (exp_bits - i)))
                mpz_mul_ui(temp,temp,primes[j]);
        }
        mpz_mul(hash, hash, hash);
        mpz_mod(hash, hash, n);
        mpz_mul(hash, hash, temp);
        mpz_mod(hash, hash, n);
    }

}
```

E.2.5 Some gprof Profiles

This Appendix shows some profiles generated by gprof. These profiles reveil only the gmp sub modules that require a significant amount of time.

Profile of a run of the original version of Classic-VSH, where $S = 1234$.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 29.09      2.71     2.71                             __gmpn_submul_1
 13.66      3.98     1.27                             __gmpn_sb_divrem_mn
 12.90      5.17     1.20                             __gmpn_mul_basecase
 11.72      6.26     1.09                             __gmpn_sqr_basecase
 11.29      7.32     1.05                             __gmpn_mul_1
  6.99      7.96     0.65        1   650.00   650.00  VSHtime
```

```
 5.48      8.47     0.51                             __gmpz_mul_ui
 2.04      8.66     0.19                             __gmpn_copyi
 2.04      8.86     0.19                             __gmpn_lshift
 1.08      8.96     0.10                             __gmpz_tdiv_r
 0.75      9.03     0.07                             __gmpn_tdiv_qr
 0.65      9.09     0.06                             __gmpn_mul
 0.54      9.13     0.05                             __gmpn_sub_n
 0.54      9.19     0.05                             __gmpz_mul
 0.32      9.21     0.03                             __gmpn_mul_1c
 0.32      9.24     0.03                             _alloca
 0.22      9.27     0.02                             __gmpn_rshift
 0.22      9.29     0.02                             __gmpz_mod
 0.16      9.30     0.01                             __gmpn_submul_1c
 0.00      9.30     0.00        1     0.00     0.00  get_data
```

Profile of a run of the Tree-based version of Classic-VSH, where $S = 1234$.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 26.66      2.04     2.04                             __gmpn_submul_1
 16.56      3.31     1.27                             __gmpn_mul_basecase
 14.99      4.46     1.15                              __gmpn_sb_divrem_mn
 13.17      5.47     1.01                             __gmpn_sqr_basecase
 10.56      6.29     0.81   640351     0.00     0.00 calculate_primes_product
  5.48      6.71     0.42                             __gmpz_mul
  5.08      7.09     0.39                             __gmpn_copyi
  1.24      7.19     0.10                             __gmpn_mul_1
  1.04      7.27     0.08                             __gmpn_tdiv_qr
  1.04      7.35     0.08                             __gmpz_set_ui
  0.91      7.42     0.07                             __gmpn_mul
  0.78      7.48     0.06        1    60.00   870.00  VSHtime
  0.65      7.53     0.05                             __gmpz_tdiv_r
  0.52      7.57     0.04                             __gmpn_sub_n
  0.39      7.60     0.03                             __gmpz_mod
  0.26      7.62     0.02                             _alloca
  0.20      7.63     0.01                             __gmpn_mul_1c
  0.20      7.65     0.01                             __gmpn_submul_1c
  0.13      7.66     0.01                             __gmpz_set
  0.13      7.67     0.01                             clock
  0.00      7.67     0.00        1     0.00     0.00  get_data
```

Profile of a run of the original version of Fast-VSH, where $S = 1516$.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
```

```
 26.36      0.68     0.68                             __gmpn_mul_basecase
 20.16      1.20     0.52                             __gmpn_submul_1
 17.83      1.66     0.46                             __gmpn_mul_1
 11.24      1.95     0.29                             __gmpn_sb_divrem_mn
  5.43      2.09     0.14                             __gmpz_mul_ui
  4.26      2.20     0.11                             __gmpn_add_n
  3.49      2.29     0.09    40960     0.00     0.00  iterate
  2.71      2.36     0.07                             __gmpn_sqr_basecase
  1.94      2.41     0.05                             __gmpn_kara_mul_n
  1.94      2.46     0.05                             __gmpn_sub_n
  1.16      2.49     0.03                             __gmpn_rshift
  1.16      2.52     0.03                             __gmpn_tdiv_qr
  0.78      2.54     0.02                             __gmpn_copyi
  0.78      2.56     0.02                             __gmpn_lshift
  0.39      2.57     0.01                             __gmpn_dc_divrem_n
  0.39      2.58     0.01                             mpn_dc_div_3_by_2
  0.00      2.58     0.00        1     0.00    90.00  VSHtime
  0.00      2.58     0.00        1     0.00     0.00  get_data
```

Profile of a run of the Tree-based version of Fast-VSH, where $S = 1516$.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 23.33      0.63     0.63                             __gmpn_mul_basecase
 17.04      1.09     0.46                             __gmpn_mul_1
 15.56      1.51     0.42                             __gmpn_submul_1
  9.63      1.77     0.26                             __gmpn_sb_divrem_mn
  6.30      1.94     0.17                             __gmpn_sub_n
  6.30      2.11     0.17                             __gmpz_mul_ui
  4.81      2.24     0.13                             __gmpn_add_n
  4.07      2.35     0.11    40960     0.00     0.00  iterate
  2.96      2.43     0.08                             __gmpn_sqr_basecase
  2.96      2.51     0.08                             __gmpn_tdiv_qr
  2.22      2.57     0.06                             __gmpz_cmp
  1.85      2.62     0.05                             __gmpn_kara_mul_n
  1.11      2.65     0.03                             __gmpn_lshift
  0.37      2.66     0.01        1    10.00   120.00  VSHtime
  0.37      2.67     0.01                             __gmpn_copyi
  0.37      2.68     0.01                             __gmpn_divrem_2
  0.37      2.69     0.01                             __gmpn_rshift
  0.37      2.70     0.01                             _alloca
  0.00      2.70     0.00        1     0.00     0.00  get_data
```