University of Wollongong
Research Online

University of Wollongong Thesis Collection 1954-2016

University of Wollongong Thesis Collections

2001

Aspects of micropayments

Terje Tollisen
University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/theses

University of Wollongong
Copyright Warning
You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.
You are reminded of the following: This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author. Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation

Tollisen, Terje, Aspects of micropayments, Master of Science (Hons.) thesis, School of Information Technology and Computer Science, University of Wollongong, 2001. https://ro.uow.edu.au/theses/2902

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

Aspects of Micropayments

A thesis submitted in the fulfilment of the requirements for the award of the degree

Master of Science (Honours)

from

University of Wollongong

by

Terje Tollisen

Faculty of Informatics,
School of Information Technology and Computer Science,
University of Wollongong,
Wollongong, NSW 2522,
Australia.

August 2001

Certification of Originality

I herby declare that this submission is my own work, and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgment is made in the text.

[Terje Tollisen]

Acknowledgments

I would like to thank Professor Josef Pieprzyk who was my supervisor during the first half of this degree. He was the one who got me interested in cryptography and got me started on the topic of this thesis.

I would like to thank Professor Jennifer Seberry who was my supervisor during the second and mot important part of this degree. She helped me realise the concepts in this thesis and getting the final version written

I would also like to thank the members of the Cryptography research group at the University of Wollongong for inspiration and team spirit.

I also would also like to thank the Internet community and the people in it for all the information, ideas and advice shared within that community.

Finally, I would like to thank my parents for all the support they have given me during my studies.

Abstract

There are many challenges to make micropayment systems on the Internet work as reliably, safely and efficiently as they need to. I have studied many of these problems, and seen how different researchers have tried to solve the challenges. A summary of many of these problems and suggested solutions are presented in this thesis.

A new micropayments system is presented, based on Merkle's authentication tree and Winternitz's one-time signatures. The scheme can add efficiency and flexibility to a range of existing micropayment schemes based on hash chains. Unlike earlier system, hash chains can be made relatively short, since the computational cost of authenticating a new hash chain is made small.

An implementation of suggested micropayment system has been done; this is new. An implementation of the Winternitz signature scheme has also been made. This signature scheme is mostly discussed only in theory in the literature, and only a few implementations exist. Both the Winternitz signature scheme and the new payment system have been tested for time and space requirements and compared favourably to well known signature systems like DSA and RSA. With optimal settings, a Winternitz signature can be done 14 times as fast as DSA (1024) and 28 times as fast as RSA (1024).

Storage requirements are a problem for the Winternitz signatures. A second implementation was therefore made, focusing on this problem. The storage required by the signer was thus reduced by a factor of about 28 by sacrificing some signature speed.

Contents

Chapter 1

Introduction

The introduction of the web browsers in 1993 changed the Internet considerably. All of a sudden everybody with a computer could navigate the World Wide Web, and it was no longer an exclusive club for computer people. It didn't take long before businesses started to use the Internet to promote themselves, and to sell goods and services. This development has exploded in the last few years, and will keep on going at an accelerated speed for years to come.

Being able to perform payments and trust them to be secure is a vital part of a business infrastructure. A lot of research has been undertaken into making payments secure on the Internet. Computationally demanding cryptography, and especially public key cryptography are means by which security is achieved, but there are still a lot of challenges ahead.

One particular category of electronic payments is called micropayments. These are payments with a very small value, typically from less than a cent to a few dollars. A large number of transactions are expected to be made since each payment is worth so little. This means that a higher level of efficiency is required than for other (macro-) payment systems. It is a major challenge to make the micropayment systems computationally efficient enough while keeping the appropriate level of security. It seems to be the view of the crypto-community that public-key signatures are too computationally expensive, and other methods must be found instead. Several schemes use one-way hash functions to obtain the desired speed and efficiency. Both Rivest and Shamir [Ri,Sh'96], and Anderson, Manifavas and Sutherland [An, Ma, Su'97] use hashing to produce the actual tokens. Glassman, Manasse, Abadi, Gauthier and Sobalvarro [Milli'95], use hashing to authenticate the payments by utilising Message Authentication Codes (MACs).

1.1 Contributions made in this thesis

Many papers have been written on micropayments (most of them after 1996). I have studied a variety of these and collected the different problems, concerns and suggested solutions to given problems discussed in different systems. Naturally, several researchers address the same issues, but they address them in different ways with different perspectives. It can be difficult to get a full overview of what has and what has not been done. A summary of how some of these papers intend to solve various problems can therefore be very useful for people intending to study the field of micropayments.

Possible improvements to micropayments based on hash chains are suggested. A framework for a payment system is given, where the Winternitz one-time signatures are used. The system offers flexibility to existing payment systems.

I have made an implementation of the Winternitz one-time signature as a part of the suggested payment system. This signature scheme is often considered to be only of theoretical value, but modifications have been done to make it's use feasible. With the right conditions, a signature can be made about 28 times as fast as an RSA (1024) signature [Ri, Sh, Ad'78], and 14 times as fast as a DSA (1024) signature. DSA (Digital Signature Algorithm) is a defined in the U.S. Federal Information Processing Standard FIPS 186, named the Digital Signature Standard (DSS).

1.2 Outline of this thesis

Chapter 2 discusses the background material. A few of the cryptographic primitives and functions used in this thesis are described. Background on conventional (macro-) payment systems is also provided.

Chapter 3 is the summary of micropayment systems mentioned in Section 1.1. Some of the major issues and their suggested solutions are described and discussed.

Chapter 4 is a description of the Merkle one-time signature, the Winternitz one-time signature and the Merkle authentication tree. These structures are used in the suggested payment system.

Chapter 5 describes the suggested payment system mentioned in Section 1.1. Modifications are done to the Merkle authentication tree and the Winternitz signature scheme. Properties that can be gained by using the suggested improvements are listed.

Chapter 6 describes the details of the implementation mentioned in Section 1.1. Data for signature sizes and operation time are presented.

The material in Chapters 5 and 6 in new and it is intended to be written up for publication.

Chapter 2

Background

2.1 Technical background

2.1.1 Cryptographic functions

One-way functions

A one-way function (Diffie and Hellman, [Di, He'76]) is informally a function that is easy to compute, but hard to invert. If F is a one-way function, then it is easy to compute $y=F(x)$. However, given y and F, it is difficult to compute x.

One-way hash functions

One-way hash functions, OWHF are a special family of one-way functions. A hash function produces a finite length digest of an arbitrarily long message, Merkle [Me'89]. A OWHF is often called a weak one-way hash function in cryptographic literature, Menezes, van Oorschot and Vanstone [Me, Oo, Va'97]. Hash functions are often written with a script font, and $\mathcal{H}$ will be used throughout this thesis.

Collision resistance

Hash functions that are used in cryptography often needs to be collision resistant. Collision resistance can be defined as ([Me, Oo, Va'97]):

"Collision resistance - it is computationally infeasible to find any two distinct inputs x and x' which hash to the same output, i.e., such that $\mathcal{H}(x) = \mathcal{H}(x')$."

Collision-resistant hash functions (CRHF) are often called *strong one-way hash functions* in cryptographic literature [Me, Oo, Va'97].

Digital signatures

Digital signatures (public key) was first suggested by Diffie and Helman, [Di, He'76], and later explored by Rivest, Shamir and Adleman in [Ri, Sh, Ad'78]. Much research has been done in this area. Look in the [Me, Oo, Va'97], for a good covering of the topic.

A signature scheme consists of two algorithms: signing algorithm and verification algorithm. Each participant has a secret and a public key. Using the signature algorithm, user A can use its secret key to sign a message. Anyone can use A's public key and the verification algorithm to verify that the signature was produced by A. Without the knowledge of A's secret key it is infeasible to make a signature that appears to be made by A.

2.1.2 Notation

Binary concatenation

Binary concatenation is represented with $||$. Given two bit strings A and B, the resulting bit string C that is produced by appending B to A is: $C = A \mid\mid B$.

$A=1010 \quad B=1100 \quad C=A||B=10101100$

Public key functions

There are two or more players in a signature scheme. Let them be the signer A, and the verifier B. They have a set of keys each containing a secret and a public key. The secret key of A is named SK_A, and the corresponding public key is PK_A. A message M signed by party A is written as $<M>SK_A$, and a message M' encrypted by B for A's eyes only is written $<M'>PK_A$.

If A sends $<M>SK_A$ to B, it is assumed that B also receives M, unless otherwise noted (this is not true if B sends $<M'>PK_A$ to A).

It is common practice to sign a digest of a message rather then the whole message, to save space. Let $\mathcal{H}$ be a hash function. If the texts says that A sends $<M>SK_A$ to B, then it implies that A actually sends $<\mathcal{H}(M)>SK_A$.

2.2 Introduction to electronic payment systems

There has been a lot of research in the area of electronic payments during the last twenty years or so. Some of the first research and discoveries in this field was done by David Chaum [Ch'82].

Electronic payments can be defined in several ways. In general, an electronic payment protocol aims at making it convenient, safe and cheap to make payments over a network. Damgård , in [Da'88], use a very general, yet fitting definition: " Payment systems and credential mechanisms are protocols that allow individuals to conduct a wide range of financial and social activities while preventing even infinitely powerful and cooperating organizations from monitoring these activities".
Other definitions have been and will be used by other people. In this thesis I will use a very wide definition: any payment made over a computer network will fall into the category of electronic payments. This definition includes all types of computer networks, but the focus in this thesis will be on Internet payment systems (IPS) [Sh, Sw'98].

Several electronic payment schemes have been made; some just as theoretical papers, some with trail implementations, and some that are actually being used in the market today. According to [Milli'95], some of these are: DigiCash, Open Market, CyberCash, First Virtual, NetBill.

2.3 The entities in an electronic payment system

When we focus on the Internet in specific, we talk about an Internet payment system (IPS). Several parties are involved in an IPS, and these will vary depending on the scheme. A Delphi survey done by Shon and Swatman in [Sh, Sw'98] showed six important players. These were financial institutions, IPS providers, merchants, customers, regulators and network providers.

Different schemes focus on different parties, and most schemes do not consider all of these. A trusted third party (TTP) is needed in addition to the six mentioned above, to distribute verified public keys.

Financial institutions. Their main function is to handle the real money transactions. It might be a bank that does money transfers between accounts, or a credit card company that can bill a cardholder.

IPS providers. These are the manufacturer and provider of IPS services. They will most likely work closely with one or more financial institutions, or they might actually be the same entity.
Several IPS systems use on-line verification of payments. In these cases the IPS might work as a clearing server that must approve each payment.

Vendors. These are the merchants that do business on the Internet. It might be a small one-man business, or it might be a large electronic department store. The vendor might sell services, electronic goods that can be transferred over the network, or physical goods that must be sent by normal mail.

Customers. These are the general users of the IPS system that will use the scheme to pay for merchandise. It might be an individual sitting at home at a PC, or it might be a large cooperation. As long as the entity is paying someone it will be identified as a user. Customer and user will be used as the same entities in this thesis.

Regulators. This refers to the legal authority. Their concerns might be the impact of the IPS on the financial system, if it allows payments to be tracked, the protection of users etc. They might establish an entity that will help settle disputes between parties

Network providers. These parties supply the actual network facilities, such as telecommunication capabilities and other necessities to make the Internet work.

Trusted Third Party (TTP). The TTP lets entities look up and verify other entities' public keys.

Broker. Brokers were first introduced in MilliCent [Milli'95], and are mostly used with micropayment systems. The broker is a link between users, financial institutions and vendors. The broker will produce and/or sell valid money, and/or issue certificates to authorise customers as rightful users of the system. Depending on the scheme in question, the broker might do other tasks like account handling and reimbursement.

2.3.1 A payments scenario

A small example of how a payment scheme is provided to give the reader an idea of how it can work. The scenario given is not taken from a particular scheme. Details about cryptographic techniques have been left out for the sake of simplicity and generality.

Alice has a bank account with Bank One. She wants to withdraw \$40 worth of electronic money from that account, and contacts Bank One online. Bank One provides her with the \$40, and deducts that amount from her account. Each piece of money has been signed with the bank's digital signature, using the bank's private key. Alice can verify this signature by using the bank's public key. This key can be provided and verified by a Trusted Third Party.

Later, Alice finds some music she wants to buy on the site of the music store Tunes. She pays them the \$5 required to buy the two songs she wants. Tunes controls the structure of the money and Bank One's signature to make sure the money is authentic. Then they give Alice the passwords to download the music she has bought.

At some point during the day, Tunes contacts Bank One to get redeemed for the payment Alice made. (Any other payments made to Tunes by Bank One customers are settled at the same time.) A record of the transaction between Alice and Tunes is sent to Bank One, who controls the electronic money. Tunes get their \$5, and Bank One keeps a record of the payment for future reference.

2.4 Classifications of payment systems.

There are several ways to identify and classify payment systems. Ferrira and Dahab [Fe, Da'98] have written a paper that focuses on this particular issue. Some of the major classifications will be mentioned here.

2.4.1 On-line or off-line

Systems that are on-line differ fundamentally from those that are off-line. Normally, it is the financial institution's on-line status we are referring to, but it could also be a broker, or the IPS provider and its payment server (the entity in question will be referred to as a payment authority in this section).

If the system is on-line, the vendor will contact a payment authority in real time while a transaction is being done. The payment authority will check if the payment is valid, if the user has tried to spend this money before, if the user has enough funds to make the payment and so on. Only after the payment authority has accepted the payment will the vendor go through with the transaction.

In an off-line scheme the contact between the vendor and the payment authority is much less frequent. The vendor will normally contact the payment authority at regular interval (e.g. once a day) to clear the payments he has received and to get reimbursed for them.

An on-line scheme has the advantage of increased security, as the payment authority approves every transaction before it is made. The drawback is that it increases communication costs, system costs (systems must be available at all times) transaction times etc. The off-line schemes do not need all this, but more complex (and more expensive) computations will normally be involved to preserve the required level of security.

2.4.2 Hardware or software based

Several systems have been proposed that take the advantage of using a piece of specialized hardware. This might be a PCMCIA (Personal Computer Memory Card International Association) card to a PC, a smart card or other devices. Dedicated hardware is often referred to as an electronic wallet. The wallet might consist of several parts, and one of these must be tamper resistant to safeguard against backward engineering and other attacks.

An early electronic wallet was proposed by Even and Goldreich in [Ev, Go'83]. Further work has been done in several papers by Chaum , Pedersen, Brands and others. Some of the most current and detailed work can bee seen in the ESPRIT project CAFE [CAFE]. Mondex (www.mondex.com) is an up and running service that uses smart cards to store information, and special devices (including phones) to make transfers between cards.

Hardware based systems have several advantages over software based ones. It is assumed that the hardware in question is tamper resistant, and thus it is infeasible for the user to change the data in it by physically opening the device. This makes it easier to control things like double spending, as the device will prevent a user spending the same money twice or tampering with the registers that hold the monetary values.
The main drawback with a hardware based system is increased costs. Every user and point of sale (POS) must have one or more pieces of specialised hardware (both the wallet and a device to read the data). Also, if the user does not have access to his device, then he cannot perform and transaction.

2.4.3 Value of payments

The security of the payment system needs to be better when the vale of each payment gets increases. Ferreira and Dahab, [Fe, Da'98], defines three broad size groups as large, medium to small and micro. Large payments are those of several hundred dollars, and such payments will be on-line for many years to come to make fraud very difficult. Medium to small payments range from a few dollars up to a few hundred. This is the type of payments that most of the research is being focused around. Micropayments does not really have a defined lower threshold, but the upper limit is normally set to a few dollars. More details of micropayments are given in Chapter 3.

2.4.4 Payment types

Two major groups of payments are token based and notational systems, [Fe, Da'98]. Token-based systems operate with specific pieces of digital information often referred to as tokens or electronic coins. These will often have a set face value, and the user will have to pay the vendor with several coins to get the exact amount due. Token-based systems are often called cash like systems.

Most notational payment systems are either cheque based or account based. The user will sign an electronic "cheque" or an account transfer authorisation with a digital signature, and the vendor will show this to the right payment authority to get redeemed.

We also make a distinction between pre-paid and credit based systems. This is tightly linked with the above-mentioned grouping. A pre paid system will normally be token based. A user buys tokens from a payment authority, and these tokens will be presented to a vendor as payments. Most systems use tokens that can only be showed once, but some schemes have proposed tokens that can be used multiple times [Fe'93]. A notational system with normally be credit or debit based.

2.5 Properties of an electronic payment system

Depending on the scheme and the authors behind it, different properties will be considered important. It would be very difficult to make a payment scheme that satisfies every possible need, so choices and priorities must be made. The most common properties found in systems are listed below.

- Anonymity. Several systems focus on the protection of the identity of the user. The idea is that no entity *A* in the system should be able to track how another entity *B* spends his or her money. Even a bank should not be able to make a connection between a customer and the money that has been issued to that customer. A possible exception from this might be regulators, to make it possible to track and stop online criminal activity.

The earliest mechanism for providing anonymity was blind signatures, invented by David Chaum [Ch'82]. These signatures allow a financial institution to sign electronic money without being able to make a connection between the user and the serial numbers on that money.

Ivan Damgård introduced a system using pseudonyms in [Da'88]. This allowed a single user to identify him or her self as different entities to different organizations. A

TTP is needed to keep track of use different pseudonyms. A similar approach is used in the NetBill protocol [Co, Ty, Si'95].

- System security. Security is a very wide term, and could be sub divided into several different properties like integrity and robustness [Fe, Da'98], privacy, confidentiality and non-repudiation.
  This is the most important property of any payment scheme.
  System security makes it intractable for any entity to do anything that entity is not authorized to do. No entity should be able to spend more funds than it is allowed to. No entity should be able to unlawfully assume a different identity than its own. No entity should be able to manipulate another entity's data. It should be possible to detect, prevent and punish unauthorized use.
  Encryption and signatures with public keys are two of the major tools for achieving this. Symmetric encryption and one-way functions are also used.

- Cost. The cost of doing a transaction should be low compared to the value of the actual transaction. The system must be cheap for a customer to use, but it must also be profitable for the IPS provider.

- Prevent double spending. This goes hand in hand with system security, but it is so important that it deserves special mentioning.
  It is very easy to copy electronic data, and thus electronic money. This is a major disadvantage electronic money has compared to normal cash, which is considered to be intractable to copy.
  Double spending is also tightly linked to anonymity. If the system has perfect anonymity, it is difficult and/or expensive to prevent and detect double spending; and double spending is easy to handle if there is no anonymity.
  A system was introduced by Chaum, Fiat and Naor in [Ch, Fi, Na'88], which allow double spenders to be caught. The user's identity is protected as long as no double spending is done. But it is possible for the bank to find the user's identity if the user spends the same electronic coin more then once.
  Other ways to prevent double spending are used in different schemes.

- Divisibility. It must be easy to pay a vendor the exact amount that he asks for. This is not a problem in a notational system, but a token based system must have mechanisms for dealing with this.

Chapter 3

Aspects of micropayments

3.1 Desired properties for micropayment systems

The major differences between properties for macro- and micropayment systems evolve around the value of each individual payment. A lot of work has gone into making the systems faster and more efficient, and thus computationally cheaper. A long list of different properties required or desired can be composed, and it would be favourable to have as many of them in both macro- and micropayment systems. However, some of them need special consideration to make micropayment systems work:

- Minimization of computational requirements for the system. Since each payment is so small, it should not require much use of expensive hardware to make it. This goes for both the creation, verification and depositing of the electronic money. One of the key methods used is to minimize the use of public key operations. Macropayment schemes often use public key signatures to bind a payment to an entity, but this is deemed too expensive for micropayments.
Common techniques are the use of efficient one-way hashing schemes and private key cryptography.

- Minimization of the communication costs. Communications between the parties involved costs both time and money. See Section 3.4 on on-line vs. off-line payments for details.

- Certified delivery. This is a guarantee for both parties involved in a transaction to ensure they both will get what they want. For the customer this means that he or she will have to pay if and only if the goods are delivered. For vendors it means that the customer will only get the goods if the payment is made. This is possible with micropayments, since the goods often will be delivered over the same network as the payments.

- Micromerchants support. Micromerchant is a name used for entities selling only small amounts of electronic goods. They will most likely be individuals without a large support system, but who have goods that people are willing to pay for. Examples might be freelance reporters or artists.

- Handling streaming. Micropayments can be used to pay for media and other services where a payment is good for a time period. This can for example be telephone calls or pay per view movies. The payment system must handle streaming of media and other time dependant service like these.

3.1.1 Other properties wanted in both macro- and micropayment schemes:

- Offer strong security for all parties.
- Minimize the need for special hardware.
- Minimize fraud in the system. Special consideration should be given to double spending.
- All parties must be able to authenticate themselves as valid entities to other parties they are dealing with.
- Fairness.
- Provide users with anonymity, privacy and untraceability.
- Scalability. There should not be any bottlenecks in the system.
- It should be easy to pay any arbitrary value in a transaction.

- Transferability. It should be possible for several parties to make payments with the same piece of electronic money before cashing it with a financial institution
- Interoperability. The system should support multiple currencies. It should also be possible to deposit a piece of electronic money with another financial institution than the one that originally made or issued it.
- Non-repudiation. An entity should not be able to go back on a deal that has been agreed upon through the participation in a transaction.

3.2 Setting for micropayment schemes

The entities in a micropayment scheme are mostly the same as in a macropayment system. However, there are a few differences, and some of the entities can perform different tasks. The main difference probably is the broker.

3.2.1 The broker

The broker was introduced in the Millicent scheme presented in [Milli'95], and is used in many other papers since. It acts as a link between the customers, vendors and financial institutions, and can handle the customers' and vendors' accounts.

The broker will sell or issue electronic money to customers, and will redeem vendors when they contact the broker to return the money. Another option is to let the broker certify the customers to produce electronic money for themselves. This saves communication cost between brokers and customers, and computation costs for the broker. By letting customers create their own payments we move towards a more distributed system, and the chances of bottlenecks become fewer.

If the broker creates the payments, then these will be sold in bulk to the customers. The customer will pay the broker through a macropayment system or with a credit card. If the customer pays the broker with an anonymous macropayment system, then the micropayment system in question can qualify for anonymity.

If the customer creates the payments, then the system cannot be anonymous. The customer will pay the vendor, and the vendor will accept the payments because the customer has a certificate from a trusted broker. To receive redemption, the vendor sends this broker the money received from the customer. The broker checks the payments, and if they are valid the vendor is paid and the customer is billed for the purchase.

A third option is that the vendors produce the payments. This can be done in certain systems where the payments are vendor specific, and was proposed in the Millicent system [Milli'95]. The brokers will buy the payments from the vendors in large bulk to get a good price, and sell them is smaller quantities to customers for a higher price. The benefit of this system is that the vendor does not need to contact the broker for redemption. But the downside is that the vendors need large hardware capacity to produce the payments. The communication during the bulk purchases will be relatively intensive, but no extra communication is produced. The payments will have to go from the vendor to the broker either way, be it before or after the customer has spent them.

3.3 Anonymity

In electronic payment systems anonymity refer to the property of protecting the actual identity of the entities in the system. The main focus is to protect the identity of the customers.

There are several motivations to protect the customers' anonymity, but they all evolve around the ability to hide the customers' spending patterns. If these are not hidden, data can be collected and profiles can be made to match each individual user's habits in the digital environment. This can easily lead to what is called intrusive profiling [Br'99]. The most obvious aspect of intrusive profiling is directed advertising, where the user is 'bombarded' with ads and offers that have been custom made to his or her interests and shopping routines. But more serious consequences like discrimination and political assault are also quite likely

But it is not enough to protect the identity of each individual. We also need to make it difficult to see of two payments have been made by the same entity or not. This property is called unlinkability, and is tightly bound to anonymity.

Strong anonymity can be a problem in macropayment systems due to the potential of criminal use. Authorities are not interested in creating new payment systems that are attractive to criminals. If the anonymity is too strong, it will be impossible to trace illegal transactions to either of the parties involved. This can be used for blackmail, money laundering and other unlawful actions.

Criminal usage is much less of a problem with micropayment systems, as the values are so low. After all, it will be somewhat difficult making much money with illegal use of payments worth only a few cents each. This would lead one to think that micropayment systems could have stronger, and perhaps absolute, anonymity implemented. However, there are technical problems doing this:

As was mentioned in Chapter 2, one of the main ways of achieving anonymity is using one-show blind signatures, first introduced in [Ch, Fi, Na'88]: These are a special type of public key signatures that lets the signer sign a blinded message. Thus, a customer can get a signature of the bank on electronic money, and the bank will not be able to link that money (through its serial number) to that particular customer. This scheme can work well for macropayments, but becomes too expensive for micropayments. In general, the number of public key operations should be minimized in a micropayment system, and one public key signature on each payment seems to be too computationally expensive.

3.3.1 Properties of anonymous schemes

If a payment scheme is anonymous, it is infeasible to make the connection between a user and a payment that have been made. That is, it should not be possible to find out who was the paying party in a given transaction.

A consequence of anonymity is that the payment system must be pre-paid. If it is not pre paid, then the customer must be sent a bill after the electronic money has been spent. To do this, the bank must be able to make a connection between money spent and the user that spent them.

3.3.2 Batch signatures

An option to validate electronic payments is to use batch signatures. This way, one public key signature can be used to authenticate several payments, and the computational cost can be spread out amongst them.

This method is used with several proposed systems based on chain and tree structures. The names of some of there schemes (and the papers they are described in) are PayWord[Ri, Sh'96], NetCard [An, Ma, Su'97], Pedersen's proposal [Pe'96], μiKP[Ha, St, Wa'96] and PayTree [Ju, Yu'96]. A series of payments are made from a single tree or chain, and the root of the tree or chain is authenticated through a signature. Each payment can be linked back to the signed root, and thus be verified as authentic.

A problem with batch signatures is the innate linkability between each single payment. An aspect of the anonymity property is that it should not be possible to identify two payments as coming form the same entity. This property is naturally violated if each payment is verified by linking it back to another payment. However, this can be acceptable for some payments, and can thus be used in certain micropayment systems. Examples are phone calls or pay per view movies, where several payments are made to pay for the same product (e.g. a payment every minute for the duration of the movie). We still want to protect the customer's anonymity, but it is acceptable that the vendor knows that each payment comes form the same customer.

Another problem with batch signatures is how to handle partially spent batches. Since anonymous schemes must be pre-paid, the money in a signed batch has already been debited from the customer's account. Then the question arises what to do about the money that is left in a partially used batch. Theoretically the vendor can give back change, but this will violate the customer's anonymity. Another way to handle this is by the bank refunding the customer at a later time, but this involves a fairly long delay.

3.3.3 Pseudonyms

A non-cryptographic technique used to protect the anonymity of the customers is to let them use pseudonyms instead of their real identity. This can be done in different ways, but it will normally involve some type of anonymity server.

A user can register with an organization, and this will issue a public/private key pair, not giving away the anonymity of the user. When the user deals with a vendor, remailers and other services can be used to deliver the electronic goods to the customer without revealing the identity. A problem with this type of anonymity is that the financial institutions in the system might be unwilling to let anonymous customers establish accounts. This can also interfere with tax laws and other regulations.

The payment service provider might also offer pseudonyms. This way, the customers can identify themselves with different names to different vendors, and their spending patterns will thus be hard to map. Banks can offer similar services, and some of the above-mentioned problems can be avoided.

Using pseudonyms, every payment can be traced back to an identity. The security lies in the infeasibility of linking that identity to an actual customer. However, if this can be done once, than all other payments done by that customer using that pseudonym can also be traced. This is a general problem with systems using pseudonyms. Allowing customers to use several pseudonyms helps, but the linkability between payments is still a problem.

Another problem with pseudonyms is that the anonymity is protected by trust rather than the computational infeasibility of revealing the customer's identity. This can prove to be a problem, but it can also fit nicely into the existing trust model. For example, we already assume there exists a trusted party to issue public/private key pairs.

3.4 On-line and off-line systems

Some macropayment systems use on-line verification of payments to strengthen the security of the transactions. This is a useful technique to prevent fraud like double spending and counterfeit money, but comes at a cost; namely the communication overhead required. The vendor will contact the bank to make sure a payment is authentic and in order before the transaction with the customer is executed.

This type of on-line verification (also called on-line payment) is considered to be too expensive for micropayment schemes. However, several micropayment systems propose to use of communications with the bank on other occasions than for withdrawal and deposit. This will be discussed Section 3.5 about probabilistic payments.

There are several drawbacks with on-line verification, besides the fact that it produces time-delays due to communications. The overall cost of the systems also increases, as does the chance for bottlenecks to occur.

If a vendor is to contact a payment authority for every payment, then that authority must be on-line at all times. The cost for being able to handle high traffic even at peak times will be considerable for the payment authority and thus for the system as a whole. This might be bearable for a macropayment system, but may be too expensive for a micropayment scheme. However, there are the cases where a payment system is hybrid, handling both macro- and micropayments. The cost of the on-line system will be spread out over a larger user and payment group, and this might make the system economically feasible.

Related to the financial cost of the on-line system is the number of available on-line payment authority servers. These servers can easily form bottlenecks in the system, making delays too long. A network of servers is needed to handle the load, pushing the price up further.

3.5 Probabilistic payment systems

Some of the problems with micropayments have tried to be addressed by adding the property of chance to the system.

3.5.1 On-line verification

As mentioned, on-line verification of payments is considered too expensive for micropayment systems. But what happens if only a few of the payments are verified on-line and the rest are verified in batches at a later time (i.e. off-line verification).

The main reason for using on-line verification is to prevent double spending by users, and also to make sure a customer does not overspend.

A vendor can accept most payments off-line to save communication costs and time delays. However a small number of the payments picked at random will be checked on-line before the transaction is completed. This will allow a few illegal payments to be stopped before they are made, but more importantly it will discourage customers from making fraudulent payments. If a user knows there is a chance of being caught, this might stop him or her from cheating.

The probability for doing an on-line check should be proportional with the value of the payment made. The greater the value, the bigger the chance that the payment will be checked. This ensures that cheating becomes increasingly harder and more risky as the intended fraud gets bigger.

However, it is not only cheating and fraud that can be controlled with probabilistic on-line verification. It is also possible to monitor and to a certain degree control the customers credit limit.

When a customer makes a payment to a vendor, this payment is checked on-line no matter what. The bank will then know that the given customer is active with the vendor, and will keep an eye on the credit limit of the customer. Whenever the vendor sends a new payment for verification, the customer's usage is updated at the bank. If the customer spends more then his or her limit, or shows signs of doing so, the bank can contact all vendors dealing with that customer to stop all transactions. This effectively stops the customer from overspending any further.

The cost of such a system grows with the values of the transactions done. If the scheme is used with relatively high valued payment, then the cost will get closer to that of an on-line payment system. If all or most transactions stay small, then the system will be closer to the cost of an off-line payment system: The vendor in every payment systems needs to contact the bank at least once to deposit the electronic money received from customers. With probabilistic verification the vendor will have to contact the bank at least once per customer it deals with in addition to the communication needed for deposit.

3.5.2 Probability of payments

A technique for cutting down on both communication- and computational costs is to not pay every vendor every time a service is bought. It sounds a bit odd, but we can add a probabilistic chance to see if a vendor will be paid or not. It can be seen as using a specialized type of lottery tickets rather than electronic money as payments.

The idea is that it costs less to pay a few vendors than paying many. The few vendors that gets paid will be paid a lot more than what a normal micropayment is worth, and the law of large numbers will make sure the values evens out.

Normally, if a given vendors deals with a thousand customers in a day, it will receive several payments from each of them and several thousand payments must be processed. With this type of probability added only a handful of customers will actually make a payment to the vendor in question. Both the communication costs for deposit, and the bank's computational costs for checking the payments can thus be greatly reduced.

There are several ways of achieving this type of probabilistic payments. The customer can issue a 'payment' where the chance for getting paid is described. This chance can be based on a number of things, and the question if an actual payment will be made or not can be resolved instantaneous or there might be a delay.

For delayed decisions on who will and who will not get paid, an external source can be used. Examples mentioned can be numbers form the state lottery.

Protocols that settled the question of payment in real time can make use of the knowledge of the pre-images of one-way hash functions. The customer will choose random winning numbers, and commit to these with a 'payment'. The vendor will generate random numbers, trying to match the numbers of the customer. If the vendor guess right it gets paid, otherwise the vendor provides the service for free.

In an example described in [Ri'97], the vendor generates a random 30-digit decimal number w, and send the customer the hash value $\mathcal{H}(w)$. The customer will send a 'payment' committing to $\mathcal{H}(w)$and a winning condition. The winning condition is that the last three digits of w must match a random number generated by the customer. The vendor can easily check if it wins, and will send w to the customer if it does win.

Several problems arise with these types of probabilistic payments. One is that users of the system might not feel comfortable with the uncertainty of payment. The vendors never know exactly when and how much they will get paid, and the customers does not know exactly how much they spend. Even though this will even out, many people might object to the idea.

Another problem mentioned is that a system like this can conflict with the regulations for lotteries. Even this is not a lottery authorities might see it differently. Also, since lottery laws vary in individual countries, a probabilistic payment system will have to be evaluated but local authorities. There is a danger that an otherwise good payment system will not be accepted in a series of countries, and this will weaken the overall acceptability and usefulness of the payment system.

3.6 Money production

Both macro- and micropayment systems have different types of money. They can be divided into two main groups, namely token based and notational.

In token-based systems the electronic money is represented by specific digital patterns with predetermined values. Tokens are similar to conventional coins and notes, and several tokens might have to be used to pay a particular amount. Token-based systems are often called cash like, and the tokens are often referred to as coins.

Several parties in the system can do the actual production of electronic money. An issuing authority like a bank or a broker will often do it, but the customers can also do it. Either way the vendor receiving the money must be able to verify the authenticity of the payments.

3.6.1 Hash chains

Repeated hashing of a number is a much-used technique to produce the payments. Each link in the hash chain will be a separate payment often referred to as a tick, a coupon or a payword.

Some of the payment systems (and the papers describing them) using hash chains are PayWord (in [Ri, Sh'96]), NetCard (in [An, Ma, Su'97]), Pedersen's proposal (in [Pe'96]) and μiKP (in [Ha, St, Wa'96]).

A hash chain is formed by repeatedly applying a one-way hash function on a randomly generated number. Each link w_i in the chain is the hash value of the next link w_{i+1}.

To make a chain of length $n+1$, a random number w_n must be generated. This will be the last link in the chain.

Let $\mathcal{H}$ be a strong one-way hash function. The hash chain can then be generated in the following manner:

$$w_i = \mathcal{H}(w_{i+1})$$

w_0 is called the root of the chain.

A chain like this has the nice property that, if w_i is made public, only the person who generated the chain will know the value of w_{i+1}. Any other entity must break the one-way function $\mathcal{H}$ to be able to find w_{i+1}.

3.6.2 Hash collisions

Let $\mathcal{H}$ be a strong one-way hash function. It is easy to find $y = \mathcal{H}(x)$, given x and $\mathcal{H}$. It is considered infeasible to find x given y and $\mathcal{H}$, and it is also infeasible to find two values x_1 and x_2 so that $\mathcal{H}(x_1)= \mathcal{H}(x_2)$ and x_1 x_2. The first problem is called reversing the hash function, while the later is called finding a collision. However, given enough time and computing power both of these tasks can be done.

In Micromint, [Ri, Sh'96], a system was presented where a broker with specialized hardware can produce special electronic coins that consist of a k-way hash collision.

That is, a series of numbers $x_1, x_2, \ldots, x_k$ are found such that $\mathcal{H}(x_1)= \mathcal{H}(x_2)=\ldots\ldots= \mathcal{H}(x_k)$. The verification of such a coin is easy for anyone to do given the numbers and the hash function $\mathcal{H}$ but it is infeasible to produce counterfeit coins.

3.6.3 Scrip

The Millicent protocol, [Milli'95], presents a token based system, introducing scrip. Scrip represents an account that a customer has with a given vendor. This way, the money is both vendor and customer specific. A piece of scrip contains several data items, including the identity of the vendor and customer and the monetary value of the scrip. A secret key is adde to the scrip and a hash value produced, giving a certificate in the form of a MAC.

3.7 Fraud and loss of money

Some micropayment schemes are not as concerned about absolute security against loss or fraud as other payment systems with higher values per payment [Milli'95], [Ri, Sh'96], [Mu, Va, Li'97]. Micropayments can be seen as pocket change, and it is no big deal if a few micropayments get lost now and then. Likewise, a few occurrences of fraud are accepted, but it must be possible to detect and stop large-scale frauds.

It is simply too expensive to have the necessary mechanisms to make sure every single micropayment is protected and accounted for at every step of the protocol. It is enough to make fraud hard and detectable, [St, Va'97].

3.7.1 Fraud detection/prevention

The main device for preventing loss in the system is by secure production of the electronic money. Some of these are described in Section 3.6.

The NetBill system [Si, Ty'95] uses digital signatures for the transaction, which of course is a very effective weapon against fraud. However, as mentioned earlier, digital signatures are considered to be too computationally expensive for micropayments.

The market forces are considered to have a large influence in many papers. It is often assumed that customers and the market in general will shun vendors that cheat, forcing them out of business. Some of the papers discussing this are SVP in [St, Va'97], MicroPayments based on iKP in [Ha, St, Wa'96], Micro-Payments via Efficient Coin-Flipping in [Li, Os'97] and PayWord in [Ri, Sh'96].

If the payment scheme is not anonymous, then the chance of fraud goes down. A person is less likely to cheat if his or her identity is known to the parties he or she is cheating. The problem of fraud then becomes tightly linked to the authentication of the entities in the system, and whether or not a person can manage to get a fake identity and thus avoid paying their debts.

Overspending is an issue that may or may not be actual fraud, depending on the system. In a credit based system, an over spender who pays his or her bills has only committed a minor offence against the financial institution in question, and might have to pay an additional fee. If the system is pre paid, overspending will most likely be considered fraud, as the person is spending funds he or she should not have access to.

Some of the most effective methods to prevent fraud and overspending are on-line verification, one-show blind signatures and tamper resistant hardware devices.

3.8 Authentication

In a computer environment where all entities have public/private key pairs, we can use the public key to identify individual, for example through X.509 certificates [Cho, Na, Pu, Un'98].

Macropayment systems often rely on public key signatures to authenticate users, as does some micropayment systems. However, several micropayment systems do not use public key signatures at all, to save costs, and need other ways to identify users of the system.

If public key signatures are used, then this will normally just be used for one payment, or a commitment for the payments to come. Micropayment systems don't use a public key signature on each individual payment, so all systems need other measures to identify separate payments.

3.9 Protecting the customers rights

Customers want to be sure they get what they pay for, and vendors wants to be sure they get paid. This property is easy to fulfil with conventional purchases, as the customer and vendor are in the same room, exchanging goods for money.

It is difficult, and often impossible to get this property working in macropayment systems, as goods often are physical. If they are, then the vendor will ship the ordered goods after the payment is cleared, and the customer will have to wait and hope that he or she receives the merchandise. Macropayment systems must have a mechanism for receipts, since there are room for both fraud by the vendor and difficulties during shipments. Receipts can be handled quite easily with a public key signature on a message containing the purchase details.

Again we run into the problem with public key cryptography and micropayments. It will be too expensive to issue a receipt for each micropayment if a public key signature is needed on each receipt. Receipts are also hard to handle if we want to preserve anonymity in a payment system. After all, the vendor will have to make the receipt out to someone, and that is hard to do if the customer is anonymous. It can be done through pseudonym schemes, but can easily be complicated.

Another problem with receipt is that they are only good for proving that a payment was done for a given product or service. A customer that did not receive what he or she paid for will have a hard time proving this to an arbiter or even the vendor to whom the payment was made.

Quite often, it will not be practical to use receipts in a micropayment system. This is especially true for streamed products like movies and phone calls. In such cases it will be more practical to use one receipt for the whole product, not for each payment.

Most proposed micropayment system does not have mechanisms for receipts or similar safe guards. They assume that vendors that do not deliver will be shunned and go out of business. An option is to have the brokers or banks, or a central authority handle complaint about bad deliveries. If a vendor gets enough complaints, it might be forced out of business by revoking its certificates or through other means.

3.9.1 Certified delivery

A system for certified delivery was presented in the NetBill system, [Si, Ty'95]. This ensures that the payment only goes through if the customer gets the information he or she paid for. NetBill is an on-line system, and any payment system that wants to use this type of certified delivery needs to be on-line as well.

With NetBill's certified delivery, the vendor encrypts the information goods before it is sent to customer. The customer sends the payment to the vendor, and the vendor sends both the decryption key and the payment to the NetBill server. If the payment is approved, the NetBill server keeps a copy of the key, and instructs the vendor to give the key to the customer. If a problem arises with the decryption, then the customer can go directly to the NetBill server to get the key.

Chapter 4

One-time signature schemes with an infinite authentication tree

Merkle [Me'87], [Me'89] proposed a scheme where one-time signatures are used in conjunction with an authentication tree. The one-time signature is based on a system proposed by Lamport and Diffie, and improved by Winternitz and Merkle [Me'87], [Me'89].

The following description is a summary taken mainly from [Me'87], and some from [Me'89]. The reader is referred to [Me'87], [Me'89] and Menezes et al [Me, Oo, Va'97] for more detailed descriptions. Ove Heigre has written about the Merkle-, Winternitz and other one-time signature schemes in his thesis, [He'00].

4.1 The Lamport-Diffie one-time signature

The Lamport-Diffie one-time signature uses one-way functions as the base for their one-time signatures. The signature is first described in [Di, He'76], and later referenced to in [Me'87] as the "Lamport-Diffie one-time signature".

If a signer wants to sign a one-bit message $m=\{0,1\}$, this can be done in the following way: The signer selects two values x_1 and x_2, and computes $y_1=\mathcal{H}(x_1)$ and $y_2=\mathcal{H}(x_2)$, where $\mathcal{H}$ is a one-way hash function. y_i is made public. The message m is signed with x_1 if $m = 1$, and with x_2 if $m = 0$. The verifier can easily check the signature by computing $\mathcal{H}(x_i)=y_i$.

If many x_i and y_i are made, a longer message can be signed. To sign an n bit message, $2n\ x_is$ and $2n\ y_is$ must be made. The $2n\ y_is$ must be public, or the receiver must previously have received them from the signer in an authenticated manner. The $2n\ x_is$ are used to sign the message.

4.2 Merkle's one-time signatures

Merkle improved the Lamport-Diffie scheme by cutting down the size of the signature. Rather than creating $2n\ x_is$ (and $2n\ y_is$), only $n+\log_2 n$ needs to be made. This almost halves the size of the signature.
Instead of making two x's and two y's for each bit, only one is made per bit. Let the message be $M=m_1\ m_2 \ldots m_n,\ m_i=\{0,1\}$. If $m_i=1$, then x_i is released, and if $m_i=0$, then x_i is not released.

This would enable the receiver to cheat, by pretending not to receive certain x's.
To avoid cheating, a check sum must be added to M, where the number of 0's in M is noted. The message to sign is $M'=M||C$, where $||$ is concatenation and C is the binary representation of the number of 0's in M.

Let's say M is an 8 bit message. The length of the check sum C will be $\log_2 8 = 3$, so the length of the message to be signed, M', is $8+3=11$.

To sign a message of length *11*, Alice will need a vector $X = x_1, x_2, \ldots., x_{11}$ and the corresponding vector Y. Y must be known to the verifier Bob.

Let M = *"1001 1101"*. The number of *0's* is 3, which is "11" in binary. Thus C = *"011"*, and M' = *"1001 1101 011"*.

Alice sends the message M' along with $x_1, x_4, x_5, x_6, x_8, x_{10}$ and x_{11} to Bob. Bob cannot modify $M'=M||C$ if he wants to have a valid signature on M'. He cannot change a 0 in M into a 1, since he cannot create any $x_i s$ that he has not received from Alice. He can change a 1 in M into a 0, but that is going to make the count C wrong. He would have to change C too, but again he cannot produce the needed x_i.

Example 4.2.1

Alice wants to send the 8 bit message M to Bob:

$M = 1001\ 0101$

The number of 0's in M is 4, which is 100 in binary. This is the check sum for M.

$C = 100$

Append C to M to produce M'

$M' = M||C = 1001\ 0101\ 100$

Alice must produce X and Y, with length 11 (8 for M and 3 for C).

$X = x_1, x_2, \ldots., x_{11}$

$Y = y_1, y_2, \ldots., y_{11}$

Y is made public, so Bob can verify that Alice produced it.

The signature on M' is

x_1, x_4, x_6, x_8, x_9

Alice sends the message M' along with the signature to Bob. Bob cannot modify $M'=M||C$ if he wants to have a valid signature on M'.
He cannot change a 0 in M into a 1, since he cannot create any x_is that he has not received from Alice.

If he tried to modify M to 1001 1101, then he will need x_5 to produce a valid signature.

Bob can change a 1 in M into a 0, but that is going to make the count in C wrong. He would have to change C too, but again he cannot produce the needed x_i.
He can modify M to 1001 0001, since he can pretend he did not receive x_6. However, this would change C from 100 to 011. Bob would need to produce x_{10} and x_{11} to make a valid signature on M'.

4.3 Winternitz's one-time signatures

Winternitz proposed a variant to Merkle's signature that reduces the signature size, but it requires more computations [Me'87].

The idea is to reduce the number of x and y values needed to sign a message. Rather than making $y=\mathcal{H}(x)$, $\mathcal{H}$ is applied repeatedly to x. Repeated applications of $\mathcal{H}$ will have this notation: $\mathcal{H}(\mathcal{H}(\mathcal{H}(\mathcal{H}(x))))$ is written as $\mathcal{H}^4(x)$, $\mathcal{H}(\mathcal{H}(\mathcal{H}(x)))$ is written as $\mathcal{H}^3(x)$ etc., and thus $\mathcal{H}^0(x)$ is equal to x.

This way, a single x and y value can be used to sign several bits. Let $M=m_1, m_2$ be a 2 bit message and $n=4$ be the message space (4 possible messages with a 2 bit message). The public y is equal to $\mathcal{H}^n(x)$; $y= \mathcal{H}^4(x)$. The signature on M is $\mathcal{H}^m(x)$ and $\mathcal{H}^{n-m}(x)$. The signature can be verified be applying repeated hash functions to reach y.

The Winternitz scheme can be used to sign longer messages as well. To do this, the message is split into *t* sub elements of equal length *k*, and each of these elements will be signed with an *x* and *y* pair.

$M=m_1||m_2||\dots||m_t$

A checksum *C* must be added to the message in a similar fashion as in Merkle's scheme (Section 4.2). The checksum is the sum of each sub elements minus *n*.

$$C=\sum_{i=1}^{t}(2^k-m_i)\le t2^k \qquad \text{[Me, Oo, Va'97]}$$

Example 4.3.1

Alice wants to sign an 8 bit message *M.*

$M=m_1\ m_2=\underbrace{1001}_{m_1}\ \ \underbrace{0101}_{m_2}$ (9 and 5 in decimal)

Alice will use one *x* and *y* pair to sign four bits, making

$k=4$

and

$n=2^4=16$

$C=(n-m_1)+(n-m_2)=(16-9)+(16-5)=18$

$C=c_1\ c_2=\underbrace{0001}_{c_1}\ \underbrace{0010}_{c_2}$

The message *w* to sign is then

$w=M\ ||\ C=m_1\ ||\ m_2\ ||\ c_1\ ||\ c_2=\underbrace{1001}_{m_1}\underbrace{0101}_{m_2}\underbrace{0001}_{c_1}\underbrace{0010}_{c_2}$

The signature *S* consists of four hash values as follows

$$S = s_1 s_2 s_3 s_4 = H^{m_1}(x_1)\ H^{m_2}(x_2)\ H^{c_1}(x_3)\ H^{c_2}(x_4) = H^9(x_1)\ H^5(x_2)\ H^1(x_3)\ H^2(x_4)$$

Given the public Y

$$Y = y_1 y_2 y_3 y_4 = \mathcal{H}^{n}(x_1)\ \mathcal{H}^{n}(x_2)\ \mathcal{H}^{n}(x_3)\ \mathcal{H}^{n}(x_4)$$

Signature S can easily be verified by checking each s_i:

$$y_i = \mathcal{H}^{n-m_i}(s_i)$$

4.4 Merkle's authentication tree

A problem with one-time signatures is that each signature requires a new entry in a public record. This amounts to a large exchange of information that might be work if only two parties are involved, but it becomes unwieldy as a general signature scheme.

Merkle proposed a scheme where one-time signatures form a tree structure. The root of the tree is entered into a public record, held by a TTP. Each node in the tree is used to sign a message, but also to verify the authenticity of its children.

When a one-time signature is used, it must be authenticated. This is done by recursively showing the ancestors of the node to the verifier, all the way up to the root.

Each signature still has a private array x, and a public array y which is a function of x. A binary tree is used as an example for simplicity, but in theory any K-array tree can be used.

Each node in the tree has three functions:

1) sign off the left child,
2) sign off the right child and
3) sign off a message.

Thus, each node contains three separate signatures.

Two three-dimensional arrays, x and y, are needed to form the tree. The three fields in each index of the arrays are:

x[<node number>,<left, right or message>,<index within the one-time signature>].

<node number> is simply the node's index within the tree structure. *<left, right or message>* indicates if this signature is used to sign off the left child, the right child or a message. *<index within the one-time signature>* is the index of the bit this particular x-values is going to sign.

Assume p *x's* are needed to make a child signature and q *x's* needed to make a message signature. The private part of the signature in node i would then look like this:

x[i, left, 1], x[i, left, 2], ..., x[i, left, p]
x[i, right,1], x[i, right, 2], ..., x[i, right, p]
x[i, message,1], x[i, message, 2], ..., x[i, message, q] [4.1]

Let *x[i, message, *]* be all *x's* needed to sign a message (and the same for left and right children) using the signature in node i. Let also *x[i, *, *]* be all *x's* for both left, right and the message in node i.

The public part i has exactly the same structure.

The public part y of the root must be authenticated by a TTP, much like a public signature. To sign a message m, the signer uses the one-time signature described above, with the secret parameters *x[i, message, *]*. All public parameters *y[i, message, *]* are given to the verifier. Then certain *x's* in *x[i, message, *]* are shown to the verifier as well, who can now see that the signature is indeed made by the right person.

The verifier must then make sure *y[i, message, *]* are actually a valid set of parameters. This can be done with the parameters *y[j, left or right, *]*, where j is the index of parent node to node i ($j=\lfloor i/2 \rfloor$). Verification for the *y's* can be done recursively up to the root, which in turn has been authenticated by a TTP. The signer must keep sending the parent nodes $y[k, \mathit{left\ or\ right}, *]$, $0 <= k < j$, and some other information to the verifier until the root is reached. This is often referred to as the authentication path.

If we use Merkle's signature scheme (see Section 4.2), each $y[r,s,t]$ (r,s and t are the indexes as indicated in [4.1]) is computed from the corresponding $x[i,j,k]$:

$$y[i,j,k] = \mathcal{H}(x[i,j,k])$$

Each node i has a unique identifying number called $HASH(i)$, which is a collection of all the public parameters for node i.

$$HASH(i) = \mathcal{H}(\mathcal{H}(y[i, \mathit{left}, *]) \mid\mid \mathcal{H}(y[i, \mathit{right}, *]) \mid\mid \mathcal{H}(y[i, \mathit{message}, *]))$$

Chapter 5

New improvements for micropayment schemes based on hash chains

5.1 Introduction to the scheme

A payment scheme based on hash chains is presented in this chapter. These are suggestions for new extensions and improvements to existing micropayment systems based on hash chains.

The payments are structured in a Merkle authentication tree, and any one-time signature scheme can be used, although Merkle's and Winternitz's schemes have been the main focus during this research.

The new improvements offer more flexibility and opens up for time saving for both payment and verifications of already executed transactions. These properties are discussed in Section 5.5.

5.2 Related work

5.2.1 Hash chains

Diffie and Helman used repeated one-way functions in a password authentication scheme, [Di, He'76], and Winternitz used repeated hash functions to design the one-time signature scheme described in Section 3.3. [Me'87].

Several micropayment schemes use hash chains to make payments. The idea is that a chain is created be applying repeated hash functions, and the security is based on the difficulty of reversing a cryptographic hash function. Some of the first to use hash chains for payment systems were Rivest and Shamir in their PayWord system, [Ri,Sh'96], Anderson, Manifavas and Sutherland in the NetCard scheme, [An, Ma, Su'97], and Pedersen's proposal in [Pe'96].

A short description of PayWord is provided here to illustrate how a hash chain can be used for micropayments.

5.2.2 A short summary of PayWord

Let $\mathcal{H}$ be a secure one-way hash function.
n is the length of the hash chain that will be made.
The user selects a random number s_n.
A chain of values is then produced in the following manner:

$$s_i = \mathcal{H}(s_{i+1}), \quad 0 \leq i < n$$

The user ends up with a chain

$$s_0, s_1, \ldots\ldots s_{n-1}, s_n$$

where

$$s_0 = \mathcal{H}(s_1),\ s_1 = \mathcal{H}(s_2), \ldots\ldots, s_{n-1} = \mathcal{H}(s_n),\ s_n$$

Each link s_i in the chain is a payment token. s_0 is considered the root of the chain, and the user must authenticate the root so the vendor knows he will get paid for tokens related to s_0. The user authenticates s_0 by signing a certificate issued by a broker or bank.

Each payment consists of the next token from the chain, and the token's index i, thus the payment is (s_i, i). The vendor have received the previous token s_{i-1} before, and can thus verify this payment by checking that $s_{i-1} = \mathcal{H}(s_i)$ (unless i=0, in which case it is the root which is signed with a public signature).

5.3 The new payment scheme

The system is built around an authentication tree with one-time signatures that is reduced to a chain structure. Each node in the tree will contain signatures to authenticate its child, and a hash chain will be attached to the node. The hash chain will be authenticated through the one-time signatures.

5.3.1 Using a chain rather than a tree structure

An authentication tree can be arbitrarily large (or small), and only the root needs to be created initially; any other node can be made later on. The user will create a new tree for every vendor he does business with. Making a new tree is not more computation ally expensive the making a node in an exciting tree. Thus making a new tree for each vendor does not produce any extra work.

Normally a tree structure is used in order to need as few recursive calls as possible to get up to the root to authenticate a signature. This makes each signature cheap, since only log n signature authentications are needed, where n is the tree depth of the signature to authenticate. But we can shorten the authentication path to only one step here, since a new tree is made for each vendor. A child can be authenticated by its parent, and the parent has already been authenticated by the vendor in question.

Therefore, a chain will be used rather than a tree. This saves space, since each node only have to contain two signatures; one for the child and one for the message, rather than one signature for each of its k children plus one for the message. An authentication tree reduced to a chain will be referred to as a signature chain.

The message signature in a node *i* can be used to authenticate a hash chain and the monetary value pf each link in that chain like this:

$\mathit{Hash\ chain\ authentication} = <s_{0,0},\ i,\ value> \mathit{Message\ signature}_i$

Where $s_{0,0}$ is the root of the hash chain, *i* is the node depth and *value* is the value of each link in the chain.

5.3.2 Further size improvement on the signature nodes

A node in an authentication tree (or a signature chain) normally contains signatures to authenticate its children or child, and also a signature to authenticate a message. With this structure, the identification of node *i* is based on the public values of all the signatures (that is, all the *y* matricis). With a signature chain, each node will need two signatures, and the node identification *HASH(i)* is:

$HASH(i) = \mathcal{H}(\mathcal{H}(y[i, child, *]) \,||\, \mathcal{H}(y[i, message, *]))$

The idea behind this is to make sure an attacker or fraudulent user cannot insert false signature node, since each node is identified with public values that in turn is authenticated with a signature of a parent.

However, rather than using a message signature to sign the root of a hash chain, we can use the root to as an integral part of the node identification. This is done by including the hash chain root in the identic fiction HASH(i). The value of each link in the hash chain must also be included, since we no longer have a message signature to authenticate the value.

Each node will always be associated with only one hash chain with one value, and this connection does not need to be made until a payment from that hash chain is needed. Therefore, we don not lose any flexibility by dropping the message signature from the signature node. The new identification for a signature node will now be:

$HASH(i) = \mathcal{H}(\mathcal{H}(y[i, child, *]) \,||\, Hash\ Chain\ Root \,||\, Value\ per\ link)$

All three components are public values just like in the original Merkle tree. None of these values can be replaced by a fraudulent user, since the vendor will detect this when computing $HASH(i)$. The vendor cannot replace any of these values either, since the bank will detect this. Therefore, we do not compromise security with this new structure.

We have now gone from 2 to just 1 Winternitz signatures per node, so the size of each signature node is reduced to almost half (on top of the reduction achieved by going from a tree to a chain structure as described in Section 5.3.1).

5.3.3 Assumptions

The scheme is credit based. A discussion about adding hardware to allow for a pre paid version is provided later.
The bank is off-line.
Multiple currencies are supported.
Divisibility is not a problem since denominations are chosen on demand.
No anonymity is provided, unless some kind of anonymity server is used. Again, hardware can help solve this.

Three parties are involved: user *U*, bank *B* and vendor *V*. The user has an account with the bank. It might be useful to have a broker that acts as an intermediary between users, vendors and banks, but it does not make a difference for the principle of the scheme.

U establishes an account with *B*, and *B* issues *U* with a certificate. It might look something like this:

$Cert_U = \langle ID_U, ID_B, PK_U, Epx, Stat, Info \rangle SK_B$ [Ri, Sh'96]

The certificate might contain a number of things, but above are listed: user's and bank's ID, users public key, expiration date, credit status or limitation, and other information. The *Info* field might contain maximum spending limits per vendor per day for the user. The certificate is signed by the banks secret key.

The certificate authorizes the user to produce micropayments. A vendor can verify a certificate through the bank's signature, and can thus trust to be redeemed by the bank. The certificate needs to be reissued with certain (fairly frequent) intervals.

The user contacts the vendor to make a purchase. The vendor sends purchase information back, including pricing and currency. The user will send a commitment to the vendor:

$$Commitment = <ID_V, Cert_U, Time, Curr, Root>SK_U$$

The commitment contains: vendor's ID, the user's certificate (including user ID), a timestamp, the currency for the payments and the public parts of the root of a signature chain. *Root* refers to the identification of the root of the signature chain:

$$HASH(i) = \mathcal{H}(\mathcal{H}(y[i, child, *]) \| s_{0,0} \| value_0)$$

The commitment is signed by the user's secret key.

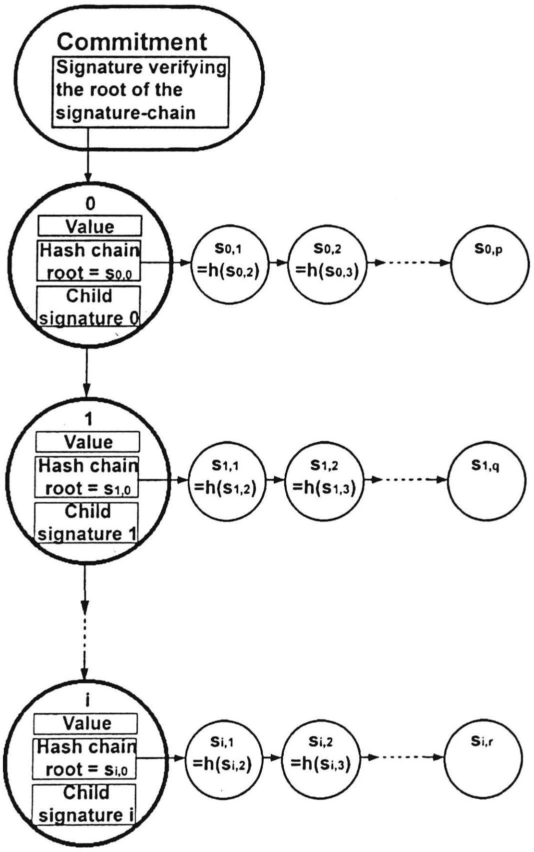


Figure 5.1:
Each signature node has a hash chain attached to it. The node is identified by its public values, the public part of the child signature, *y[i, child, *]*, the root of the hash chain, $s_{o,o}$, and monetary the value per link $value_i$.

5.3.4 Payment

After the commitment has been given to the vendor, the user is ready to start sending payments. The public *y's* in node 0 in the signature chain

is sent to the vendor, along with the root of the hash chain attached to that signature node.
The vendor can verify these vales computing *HASH(0)* can compare this to the value *Root* in the Commitment.

A new signature node with a corresponding hash chain can easily be created if a hash chain runs out, or the user needs to change denomination per payment.

The signature nodes are easy to make (and can be pre made, if this suits the particular application). This allows the user to produce many hash chains of different denominations and lengths as needed. Each link in a specific hash chain will have the same value, but each chain can have different values per link. Thus the user can send links from different hash chains depending on the payment that is to be made.

Each hash chain has the same index as the signature node it belongs to, and each link in the hash chain will have a second index, internal to the hash chain.
Signature node i will be used to sign the root $s_{i,0}$ of the hash chain $s_{i,0}, s_{i,1}, \ldots s_{i,p}$. i is the depth of the node in the signature chain, and p is the length of the hash chain attached to node i.

5.3.5 A payment example

User U contacts vendor V to purchase information on web pages. The user needs to produce a hash chain to make payments. This chain can be made in advance, before U contacts V, since the hash chain is independent of the vendor and the token values in question. Let the chain of length p be $s_{0,0}, s_{0,1}, \ldots, s_{0,p}$. $s_{0,0}$ is the root of the hash chain and is authenticated through the commitment

V wants payment in US$, and each web page costs $0.05. U sends the public parts of the signatures in the root node: $y[0, child, *]$, $s_{0,0}$ and $value_0=0.05$. U then sends a commitment:

$Commitment = <ID_V, Cert_U, Time, US\$, HASH(0)>SK_U$

HASH(0) is the identifying number for the root of the signature chain. *V* can verify the commitment with *U*'s signature SK_U, and the bank's signature SK_B on the certificate $Cert_U$.

U can now do a series of payments worth five cents each. This is done by sending the next link in the hash chain, together with that link's index to *V*.

$Payment = (s_{0,j},\ 0,\ j)\ , j \leq p$

V can verify this payment by checking that $s_{0,j} = \mathcal{H}(s_{0,j+1})\ , j<p$.

If *U* then wants to buy a piece of information goods worth 2 cents, he can easily make a new hash chain worth two cents per link. A new node in the signature chain must be made, and this node must be authenticated by its parent, namely node 0.*U* sends *y[1, child, *]* , $s_{1,0}$ and $value_1=0.02$ to *V*. V can verify the authenticity of node 1 by checking *y[0, child, *]* up against *HASH(1)*.

A hash chain must be made with a finite length. This is one of the common criticisms used against hash chains: A user must make a chain of a chosen length, and if he does not use the whole hash chain he will have done unnecessary computations. On the other hand, if the hash chain proves to be too short, another hash chain must be made, and another public signature used to sign a new commitment containing the new chain. This problem will not occur with this scheme. Each hash chain can be made relatively short. If a hash chain of a certain denomination is exhausted, the user can just make a new hash chain signed by the next node in the signature chain.

5.3.6 Redemption

The vendor contacts the bank and sends the following data: The commitment, the public part of each signature *y[i, child, *]*, all hash chain roots *si,0* and every link value $value_i$. The bank can verify the authenticity of each payment in the same way the vendor did upon payment. The vendor will be redeemed if the bank finds the payments to be authentic, and the user's account will be charged for the same amount.

5.3.6 A problem with size

A problem with this scheme is that the one-time signatures can be relatively large. As will be shown in Section 6.6.2, the public y of a typical Winternitz signature will be 880 bytes long, as will the signature. This can prove a problem, as the vendors will have to store a lot of information.

Adding a closing protocol

A step to end a series of payments can be added to reduce the size of the data stored by the vendor.

Whenever a user is finished dealing with a vendor he will make a new commitment, signing off the amount he as actually spent with the vendor. This commitment will be very much like the starting commitment:

$Commitment = <ID_V, Cert_U, Time, Curr, Root, Value>SK_U$

The only difference is that a last data field, *Value*, has been added to the end of the commitment. It will be impossible for the vendor to cash in both the opening- and the ending commitment, since both of them contain the same *Root*.

If the user tries to cheat by making the value in the closing commitment smaller than the amount he has spent, the vendor can choose to ignore the closing commitment and show the individual payments to the bank. The same will be done if the user for some reason does not send a closing commitment at all. If the vendor finds the closing commitment to contain the right amount and to be authentic, he can delete the opening commitment and all individual payments, using only the closing commitment to be redeemed by the bank.

5.3.7 Further possible improvements

Security improvement.

Each one time signature can sign the number of links released in the hash chain belonging to its parent node.

Bootstrapping

This payment system can be boot strapped to a full-blown macropayment system, much like μiKP, [Ha, St, Wa'96]. The commitment will then be replaced with a regular payment in the macropayment system, except the value of the macropayment will contain the root of the signature chain rather than just a monetary value.

Probabilistic poling

Probabilistic polling as in [Ja, Od'97] can be used to discourage overspending. This way, the bank can keep better control of the users potential overspending.

5.4 Properties of the payment scheme

These are properties any hash chain based payment system canl gain by using the scheme described in this chapter. The main improvement is flexibility in several areas.

- The payment scheme trades time saving and flexibility in several areas for signature sizes.

- It becomes easier and quicker to change denomination per payment. In most payment systems based on hash chains, the user can simply skip several links to make a larger payment. However, this technique cannot be used to make a smaller payment. The system presented here makes it easy to change denomination due to quick Winternitz signatures.
  This type of down grading of link values can be useful if the customer starts buying information that costs less, for example cheaper articles, or the cost of a phone call changes from peak to off-peak price.

- It is very easy to handle multiple currencies. Each commitment can have a different currency, as is the case with most payment systems. Additionally, each separate hash chain can actually have a new currency, although this might not have an immediate real world use at the moment.

- The same signature chain can be used within the same electronic warehouse, where there is a degree of trust amongst the vendors, much like described in PayTree, [Ju, Yu'96]. A party trusted by all vendors (this might be any vendor in the warehouse) verify the commitment, and a new hash chain is made for each vendor in the warehouse. The vendors contact each other to verify separate nodes in the signature chain.

- More flexibility is offered with regards to the length of the hash chains. A new chain can be made more often, since the computational cost of making a new hash chain is cheaper.

- The system is flexible with regards to the available hardware. Computations can be done ahead of time, or during run time, depending on memory and processor power. All the links in a hash chain can be stored, or each link can be computed from the secret s_n each time a payment is made.
  Signature nodes can be made and sorted for later, since no part of the one-time signature is revealed before it is used, and a node in a signature chain is completely independent of its parent until it is used.

5.5 Further work and open questions

5.5.1 Anonymity

Anonymity is a general problem in micropayment schemes, and the system described above does not solve this problem. The two most obvious solutions to anonymity is making the system on-line, of use special hardware, both of which are quite expensive. The reader is referred to Chapter 3 for a more thorough discussion on anonymity in micropayment schemes.

Providing anonymity with hardware

With the use of hardware, this system can provide anonymity with a few extensions, much like described in Brands' paper on Electronic Cash, [Br'99]. First of all, each user must have a piece of hardware that they use when they make payments.

The user will withdraw money from his bank account, and this cash value will be stored in the tamper resistant piece of hardware. The payment system has thus changed from being credit based to being pre paid.

The bank will issue blank cheques that are signed with one-show blind signatures. Each cheque will contain the root of a signature chain, and a maximum spending limit for that cheque. When a user contacts a vendor, he will send a cheque rather than a commitment.

The rest of the payment protocol proceeds as described above, except that the hardware device keeps track of the user's spending. This prevents the user from spending more money than what was withdrawn from the account.
During redemption, the bank needs to verify its own signature on the cheques sent in by the vendor.

5.5.2 Overspending

Users have the opportunity to overspend in this system. That is, they may spend more money than their credit limit or account balance. However, they will find it difficult escaping the bill since the bank knows the identity of the user. We can assume that the situation will be similar to when people overspend their credit card limits today.

Unless the scheme makes use of special hardware, or the scheme is on-li-e, it is very difficult (or impossible?) to prevent this kind of behaviour. Penalties can be used against overspenders, but they cannot be stopped altogether. Again, this is a general problem for software based and off-line schemes. Over- and double spending is possible, but the person doing so will be caught, and most likely made to pay by a bank or other financial institution.

5.5.3 Other signature schemes

There is no reason why this scheme can't be used with another one-time signature scheme. The more efficient and secure the signature scheme is, the more efficient and secure the payment scheme will be. This is particularly true with regards to the size of the signature scheme.

5.5.4 Further areas for study

-Wenbo Mao describes a system using Schnorr signatures described in [Mao'96]. This system lets the bank reveal the user's secret key SK_U after double spending. This might be a good way to improve security for the scheme described here.

-Transferability between users should be made possible. In theory, this can be done by letting the payee act as a vendor and receive payments as per the normal protocol. Alternatively, as long as the system is off-line and credit based it might be sufficient with a digitally signed commitment. IOU.

Chapter 6

Implementation of the proposed improvements to hash chain based payment systems

The implementation has been done in C++, using the Microsoft Visual Studio development environment. Crypto++, a library provided by Wei Dai, [Dai'01], has been included to provide the cryptographic functions.

The implementations could have been done in either C++ or Java. There are two main reasons why C++ was chosen.
First, the cryptographic libraries available for Java are still a bit limited, even after the US export laws on cryptography have changed. The standard JCA and JCE provided by SUN seemed a bit limiting, even though the implementation does not use a lot of cryptographic functions.
Second, C++ it is the preferred language of the author.

Notation in this chapter

Until now, symbols and letters have been written in italic; for example the x matrix of a Winternitz signature. It is common to use courier when writing source code, as it makes it easier to read. All letters and symbols referring directly to a variable in the code will now be written in courier as well; for example the private member `x` of class Winternitz. However, when referring to general concepts like "the public y values" italic will still be used.

6.1 Outline of the programs

An implementation has been done to get hands on experience with the micropayment system described in Chapter 5.

Three classes have been implemented. They are:

```
class Winternitz
class WinternitzShort
class Node
class Tree
```

`class Winternitz` is an implementation of Wintermitz's improvement on Merkle's signature scheme. It takes a SHA digest as an argument, and makes a Winternitz signature on it.

`class WinternitzShort` is a re-make of `class Winternitz` optimised for producing smaller signatures. The main change is the most of the private members are re-computed every time they are needed rather than stored. Some of the functions have been modified for this purpose, and there are several variations of some of the functions in order to avoid unnecessary computations. Due to the class's similarity to `calss Winternitz`, the source code will not be discussed in detail below. See Appendix C for the source.

`class Node` is the implementation of the signature node described in Chapter 5. Its main contents are the root of a hash chain, the face value of each link in the hash chain, and a Winternitz signature to authenticate it's child node.

`class Tree` is the specialized version of the Merkle authentication tree, called a signature chain in Chapter 5. It is a "tree" structure build up of instances of `class Node`, but each node has only one child, making it into a chain.

6.2 class Winternitz

The author has implemented the Winternitz signature scheme described in Chapter 4. This scheme (as well as the original Merkle scheme) is often considered a theoretical signature system, due to the size of the authentication path involved in verifying a signature. However, with the custom application and modifications done in Chapter 5, it can become useful in practice. This prospect needs to be explored, and this implementation has become a significant part of this thesis.

The Winternitz improvement to the Merkle one-time signature scheme can reduce the signature size with a factor of about 4 to 8 [Me'89]. It can be used to reduce the size more, but this will make the scheme too computationally expensive.

This implementation makes it possible to choose if that factor should be 4 or 8 (see next Section: 6.2.1). That way, signature size or computation speed can be chosen as first priority, depending on the situation.

`class Winternitz` makes a Winternitz signature on a message of length 160 bits. It is assumed that this is a SHA-1 digest of the message to be signed. What is described in Section 4.3 as the message, is thus always expected to be a digest of the message. The actual information to sign is of no interest to `class Winternitz`, only a digest of that information. In this chapter, "message" literally means "digest of the message".

6.2.1 Global values

These constants are defined at the beginning of winternitz.h.

As mentioned above, the size saving factor of the Winternitz scheme can be chosen. Setting the value of these two global constants before compilation does this:

```
const short unsigned int elementLen=4;
const short unsigned int elementPerByte=2;
```

`elementLen` describes how many bits will be signed by each y value. This corresponds to the value k in Section 4.3.

`elementPerByte` says how many sub elements there will be in each byte. The signatures will be smaller and slower if `elementLen` is set to 8 and `elementPerByte` to 1.

```
const short unsigned int digestLen=SHA::DIGESTSIZE;
```

`digestLen` is simply the length of the message to sign, which is the length of a SHA-1 digest. At the time of implementation this is 160 bits.

A checksum is appended to the message to sign, as described in Section 4.3. The length of this checksum is described by

```
const short unsigned int checkLen=sizeof(short unsigned);
```

The value of the check sum is set by $C = \sum_{i=1}^{t}(2^k - m_i) \leq t2^k$.

t is the number of sub elements of the message.

$t = 2^{\texttt{elementLen}} * \texttt{elementPerByte} * \texttt{digestLen}$

C gets the largest value if each $m_i=0$, so C gets a maximum value:

$C_{max} = 2^8 * 1 * 160 = 40960.$

A variable of length 16 bits is needed to hold this number, so the checksum can be represented by a `short unsigned int`.

6.2.2 Private members

The data type `byte` is used for several of the members. Byte is defined as `unsigned char` in the crypto++ library.

```
byte **x
```

These are the secret values of the signature. They are generated at random.

`byte **y`

Generated from `x` by applying multiple hash functions. This matrix is public.

`short unsigned int xyLen`

The number of x's and y's that is needed for the signature. `xyLen` will normally be set equal to `(digestLen+checkLen)*elementPerByte`. This will be 22 or 44, depending on the value chosen for `elementPerByte`.

`byte *subVal`

These are the sub elements described in Section 4.3.

`short unsigned int subLen`

This is the number of sub elements in `subVal`. If `subLen` is set to a value, it will be the same as `xyLen`.

`byte *m`

The message to be signed. Again, this is assumed to be a SHA-1 digest of length 160 bits.

`short unsigned int mLen`

The length of `m`. It is assumed to be 160 bits (since a SHA-1 digest is 160 bits).

`short unsigned int n`

This is the maximum value of a sub element. In Section 4.2 this is described as 2^k, which is equal to $2^{\texttt{elemtnLen}}$ in this implementation.

`byte **signature`

This is the signature matrix described as *S* in Section 4.3.

6.2.3 Constructors

`Winternitz();`

The default constructor calls `initialise()`, and makes the `x` and `y` matrices.

`Winternitz(byte messDigest[], short unsigned int messDigestLen);`

Makes a signature object, and creates a signature on the message `messDigest`. `messDigest` is assumed to be a SHA-1 digest, and `messDigestLen` is thus assumed to be 20, since a SHA-1 digest is 20 bytes.

`Winternitz(byte messDigest[], short unsigned int messDigestLen, byte **yTest);`

This constructor does not make the `x` matrix, and the y matrix is sent to it as an argument. The `subVal` matrix is made, and a signature can later be sent to the Winternitz object to see if it the signature corresponds with `messDigest` and `yTest`.

~Winternitz();

Standard destructor that deletes the arrays made by calls to `new`.

6.2.4 Private functions

`void initialize();`

Creates the secret `x` matrix and the corresponding public `y` matrix. A few other data members are also given proper values.

```
void Winternitz::computeSubVal()
{
	if(elementPerByte==1)
		for(short unsigned int i=0; i<mLen; i++)
			subVal[i]=m[i];
	else
		for(short unsigned int i=0; i<mLen; i++)
			splitByte(m[i],&subVal[i*elementPerByte]);
}
```

Makes the matrix `subVal`. If each sub element is 1 byte long, then `subVal` will be equal to the message `m`. Otherwise, each byte in `m` must be split into two bytes, padding the high order bits with 0.

```
void Winternitz::splitByte(const byte val, byte * splitArray)
{
	byte mask = 128;
	for(short unsigned int i=0; i<elementPerByte; i++)
	{
		splitArray[i]=0;
		for(short unsigned int j=0; j<elementLen; j++)
		{
```

```
                    splitArray[i]<<=1;
                    if(val & mask)//Push 1, else push 0
                        splitArray[i]=splitArray[i]|1;
                    mask>>=1;
                }
        }
}
```

This function takes a byte `val`, and splits it into several bytes that are put into the array `splitArray`. The functionality is easiest explained with an example.

Let `elementPerByte` be 2 and `elementLen` be 4.

```
val=1001 0110

mask=10000000

splitArray[0]=00000000

splitArray[1]=00000000
```

The inner for-loop tests if a 1 or a 0 should be pushed into `splitArray[0]`. The four rounds in this for loop will produce these value (after the if-statement, but before `mask>>=1`):

mask =1000 0000 (i splitArray[0]=0000 0001	mask =0100 0000 (ii splitArray[0]=0000 0010
mask =0010 0000 (iii splitArray[0]=0000 0100	mask =0001 0000 (iv splitArray[0]=0000 1001

The next four rounds will produce `splitArray[1]` in a similar fashion:

mask =0000 1000 (v splitArray[1]=0000 0000	mask =0100 0010 (vi splitArray[1]=0000 0001
mask =0010 0100 (vii splitArray[1]=0000 0011	mask =0001 0001 (viii splitArray[1]=0000 0110

The byte `val` has thus been split in two, in the same way as described in Section 4.3. This corresponds to the message *M* being split into m_1 and m_2 in example 4.3.1:

```
val=1001 0110

splitArray[0]=0000 1001

splitArray[1]=0000 0110
```

```
void Winternitz::makeCheckSum()
{
        short unsigned c=0;//the integer value of the checksum
        short unsigned int i=0;//loop counter
        int j=0;//loop counter
        short unsigned cLen=checkLen;
```

```
        byte * cVal = new byte[checkLen];//Binary representation of c

        //Compute the check sum
        for(i=0; i<subLen-(checkLen*elementPerByte); i++)
              c+=(n-subVal[i]);

        for(i=0; i<cLen; i++)
              cVal[i]=0;

        cVal=(byte*)&c;

        int tempVal=subLen-checkLen*elementPerByte;
        short unsigned int k=cLen-1;//Last index of cVal
        if(elementPerByte==1)//cVal can be copied straight into subVal
              for(j=tempVal; j<subLen; j++)
                    subVal[j]=cVal[k--];
        else
              for(j=tempVal; j<subLen; j+=2)
                    splitByte(cVal[k--], &subVal[j]);
}
```

This makes the checksum of the message `m`. The equation of the check sum is found in Section 4.3, and the first for-loop does this calculation. The next for-loop just initialises the elements in `cVal` to zero.

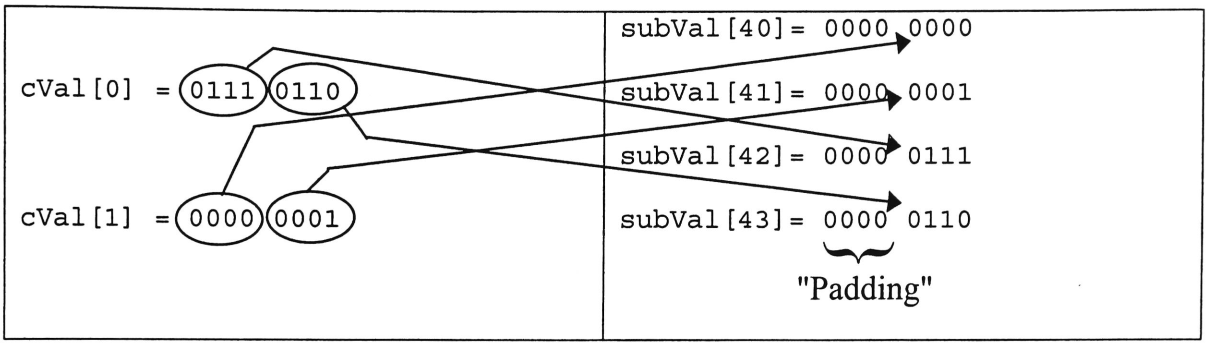
The line `cVal=(byte*)&c;` casts `c` to the byte array `cVal`. (At first it can seem like the casting "reverses" the two bytes in `c` when they are put into `cVal`, so a small example is in order).

A quick example:

```
c = 374 = 00000001 01110110
cVal[0] = 01110110
cVal[1] = 00000001
```

In the last if-statement the checksum `cVal` is appended to the array subVal. This is done in a similar fashion to what is done in `computeSubVal`. If `elementPerByte` is 1, then a simple copy can be used. Otherwise, `splitByte` must be used.
As is shown in the example above, the copying from `cVal` to `subVal` must be done from the highest index of `cVal`.

If spiltByte is used, then the address to the right index in subVal is sent as the destination of the split. Here is a small example where spiltByte is used to append the checksum above into the last four elements in `subVal`.



```
void Winternitz::produceX()
{
        short unsigned i;
        x = new byte *[xyLen];
        for(i = 0; i < xyLen; i++)
              x[i] = new byte[SHA::DIGESTSIZE];

        AutoSeededRandomPool rng;
        long seed=rng.GetLong();

        RandomPool randPool;
        randPool.Put((byte*)&seed, sizeof(seed));
        for(i=0; i<xyLen; i++)
              randPool.GenerateBlock(x[i], SHA::DIGESTSIZE);
}
```

The values in the x matrix are secret, and created at random. Creating a secret, secure and random seed is a research area in itself, and not a focus of this thesis. A function in the crypto++ library is used to create a seed, and the rest of the x matrix is generated from this seed using a pseudorandom function. If we can assume the seed is secure, then the rest of the matrix will be secure as well.

```
void Winternitz::produceY()
{

        short unsigned i, j;
        y = new byte *[xyLen];
        for(i = 0; i < xyLen; i++)
              y[i] = new byte[SHA::DIGESTSIZE];
        SHA hash;
        for(i=0; i<xyLen; i++)
        {
              hash.CalculateDigest(y[i], x[i], SHA::DIGESTSIZE);
              for(j=1; j<n; j++)
                    hash.CalculateDigest(y[i], y[i], SHA::DIGESTSIZE);

        }
}
```

As shown in Section 4.3: $y_k=\mathcal{H}^n(x_k)$. This means that each element in `x` must be hashed `n` number of times to get the corresponding element in `y`. The SHA object `hash` is used for all the hashing operations. Inside the second for-loop, `y[i]` is set equal to the digest of `x[i]`. This is the first hashing.

Then, in the inner for-loop, `y[i]` is set equal to the digest of itself. This is done `n-1` times, giving `n` hashes of `x[i]` to produce `y[i]`.

```
void Winternitz::produceSignature()
{
	short unsigned i, j, k;
	signature = new byte *[subLen];
	for(i = 0; i < subLen; i++)
		signature[i] = new byte[SHA::DIGESTSIZE];

	SHA hash;
	for(i=0; i<subLen; i++)
	{
		if(subVal[i]>0)
		{
		hash.CalculateDigest(signature[i], x[i],
						SHA::DIGESTSIZE);
			for(j=1; j<subVal[i]; j++)
				hash.CalculateDigest(signature[i],
					signature[i], SHA::DIGESTSIZE);
		}
		else
			for(k=0; k<SHA::DIGESTSIZE; k++)
				signature[i][k]=x[i][k];
	}
}
```

The signature is produced by hashing the `x` values a given number of times. The number of hashes being done is set by the value in the corresponding value in `subVal`. `SHA().CalculateDigest(signature[i], x[i], SHA::DIGESTSIZE);` makes `signature[i]` into a hash value of `x[i]`, and then `SHA().CalculateDigest(signature[i], signature[i], SHA::DIGESTSIZE);` hashes `signature[i]` `subVal[i]-1` times more.

If `subVal[i]` is zero, then no hashing is done, and the `x` value can just be copied into `signature`.

6.2.5 Public functions

`bool getSignature(byte **sign, byte **yTemp);`

If either of the two private members `signature` or `y` are not initialised, the function returns false. Otherwise it returns true.

The argument `sign` is set equal to the private `signature`, and `yTemp` is set equal to the private `y`.

`int getxyLen(){return xyLen;}`

Returns the length of the `x` and `y` matrices; the number of x's and y's needed in this signature object.

`bool getY(byte ** yTemp);`

If the private member `y` is not initialised, the function returns false. Otherwise it returns true.

The argument `yTemp` is set equal to the private `y`.

`short verifySignature();`

This is a test function that lets a signature object test it's signature on its own message. The function returns -1 if the private member `signature` is not initialised. It returns 0 if `signature` is not a valid Winternitz signature on the private member `m`. Otherwise it returns 1.

The code is very similar to `verifySignature(byte **testSign)`, so see the description of this function for details.

```
short Winternitz::verifySignature(byte **testSign)
{
	if(!subVal)
		return -1;
	if(!y)
		return -1;

	byte tempCheck[SHA::DIGESTSIZE];
	unsigned short i, j, k, t;
	SHA hash;
	for(i=0; i<subLen; i++)
	{
		for(k=0; k<SHA::DIGESTSIZE; k++)
			tempCheck[k]=testSign[i][k];
		for(j=subVal[i]; j<n; j++)
			hash.CalculateDigest(tempCheck, tempCheck,
							SHA::DIGESTSIZE);
```

```
            for(t=0; t<SHA::DIGESTSIZE; t++)
                        if(y[i][t]!=tempCheck[t])
                              return 0;
      }
      return 1;
}
```

This function tests if `testSign` is a valid signature on the message in the signature object. The function returns -1 if any of the private members `signature` or `y` are not initialised. It returns 0 if `testSign` is not a valid Winternitz signature on the private member `m`. Otherwise it returns 1.

Normally, this function will be called on an object that have been created with the constructor that takes a `y`-matrix as an argument, since such an object does not have a signature on it's own.

Each line in `testSign` is copied into `tempCheck` for verification up against the corresponding line in `y`. Then, `tempCheck` is hashed a number of time equal `n-subVal[i]`. Each element in `tempCheck` should now be the same as the corresponding value in `y`. This is controlled in the last for-loop and if-statement.

`void update(byte messDigest[], short unsigned int messDigestLen);`

Called on signature objects to update the message the object should make a signature on. The object's `x` and `y` are not changed, so update should (out of security reasons) only be called once on each object, and only on objects created with the default constructor.

6.3 class Node

This class represents signature nodes in the signature chain described in Section 5.3. The main content of each are: the root of a hash chain, a face value per link in that chain, and a Winternitz signature used to authenticate the child of the node.

`class Node` is intended to be used in conjunction with `class Tree`. `class Tree` needs access to a few of `class Node`'s private members, and is therefore a `friend` of `class Node`.

6.3.1 Private members

`int depth;`

A node-object is assumed to be in a tree, and this is the node's depth in that tree.

`float face;`

This is the face value of each link in the node's hash chain. `face` is used to make the node's identification number `id`.

`byte chainRoot[SHA::DIGESTSIZE];`

This is the root of the hash chain attached to the node. This value is considered to be public, and is used to make the node's identification number `id`.

`byte chainEnd[SHA::DIGESTSIZE];`

This is the secret base number for the hash chain attached to the node. It is generated at random, and `chainRoot` can be derived from it.

`int chainLen;`

The length of the local hash chain.

`byte id[SHA::DIGESTSIZE];`

This is the identification number of the node. It is made from the member variables `chainRoot`, `face`, and the public `y`-matrix from the Winternitz signature `wChild`.

`int index;`

Current index of the local hash chain.

`Winternitz wChild;`

Signature object for the node's child. This is used to make to node's identification number `id`.

`Node * child;`

Pointer to the node's child node.

6.3.2 Constructors

`Node::Node(int d, float f, int n, Node* c)`

This constructor is the only one implemented, and will often take only the first three arguments. As can be seen in `node.h`, the argument `c` has a default value of NULL, as it will normally be set at a later time. The hash chain for the node is generated, and the node's `id` is calculated.

6.3.3 Private functions

`void Node::computeId()`

```
{
        byte ** childY=new byte *[wChild.getxyLen()];
        for(int i=0; i<wChild.getxyLen(); i++)
              childY[i] = new byte[SHA::DIGESTSIZE];

        wChild.getY(childY);
        SHA hash;
        byte childTemp[SHA::DIGESTSIZE];
        byte chainTemp[SHA::DIGESTSIZE];
        byte faceTemp[SHA::DIGESTSIZE];
        int j=0;

        for(j=0; j<wChild.getxyLen(); j++)
              hash.Update(childY[j], SHA::DIGESTSIZE);
        hash.Final(childTemp);

        hash.Update(chainRoot, SHA::DIGESTSIZE);
        hash.Final(chainTemp);


        hash.Update((unsigned char*)&face, sizeof(float));
        hash.Final(faceTemp);

        hash.Update(childTemp, SHA::DIGESTSIZE);
        hash.Update(chainTemp, SHA::DIGESTSIZE);
        hash.Update(faceTemp, SHA::DIGESTSIZE);

        hash.Final(id);
}
```

The identification number for the signature nodes have been modified a bit from the original Winternitz scheme. See Section 4.3 for details on this.

The `id` consists of three digests that are hashed together. The three digests are:

- A digest of all the *y*-values in the Winternitz signature `wChild`.
- A digest of the root of the hash chain in the node.
- A digest of the face value in the node.

The SHA object `hash` is used for all the hashing operations. In the first for-loop, each line in `childY` is added to `hash`, and the resulting digest is stored in `childTemp`. After the call to `hash.Final(childTemp)`, the SHA object `hash` is reset and ready to start receiving new arguments.

A digest of `chainRoot` is stored in the variable `chainTemp`, and a digest of `face` is stored in `faceTemp`.

Then all three temporary digests are added to `hash`, producing the last digest `id`.

```
void Node::generateChain()
{
        AutoSeededRandomPool rng;
        rng.GenerateBlock(chainEnd, SHA::DIGESTSIZE);
        SHA hash;
        hash.CalculateDigest(chainRoot, chainEnd, SHA::DIGESTSIZE);
        for(int j=1; j<chainLen-1; j++)
              hash.CalculateDigest(chainRoot, chainRoot,
                                                  SHA::DIGESTSIZE);
}
```

The hash chain is generated from a random base number. This number is the private member `chainEnd`, generated by the random function `rng.GrerateBlock`. The rest of the chain is generated in the for-loop, ending with `chainRoot`.

It may seem odd that the last value created is called `chainRoot`, and not the other way around. This is because `chainRoot` is the first element to be sent to a vendor when transaction commences [Ri, Sh'96].

```
void Node::getChainEnd(byte ce[])
{
        for(short unsigned int i=0; i<SHA::DIGESTSIZE; i++)
              ce[i]=chainEnd[i];
}
```

This function is not strictly necessary, but implemented to keep things tidy. It is intended for friends of `class Node`. The argument `ce` is set to the same value as the private member `chainEnd`.

```
Node * Node::getChild();
```

This function is not strictly necessary, but implemented to keep things tidy. It is intended for friends of `class Node`. The node's pointer to its child is returned.

6.3.4 Public functions

`void Node::setChild(Node * c);`

The node's child is set to the node pointed to by `c`.

`void Node::getId(byte ID[]);`

The nodes private member `id` is copied into the argument `ID`.

`int Node::getDepth(){return depth;}`

Returns the depth of the node.

`float Node::getFace(){return face;}`

Returns the face value, `face`, of each link in the node's hash chain.

`int Node::getChainLen(){return chainLen;}`

Returns the length of the node's hash chain.

```
bool Node::getChildSignature(byte **sign, byte **yTemp)
{return wChild.getSignature(sign, yTemp);}
```

The two public parts of the child signature `wChild` are given, through the `getSignature`-function in `class Winternitz`.

`void Node::getChainRoot(byte cr[]);`

The node's private member `chainRoot` is copied into the argument `cr`.

`int Node::getLink(byte link[]);`

The current link in the hash chain is copied into the argument `link`. This is the hash value indexed by the private member `index`. Note that `index` is not updated by this function.

`int Node::getLinkNext(byte link[]);`

The current link in the hash chain is copied into the argument `link`. This is the hash value indexed by the private member `index`. `index` is incremented one step towards `chainEnd`.

6.4 class Tree

`class Tree` is the signature chain described in Section 5.3. It consists of nodes of the type `class Node`. Most of the functions in `class Tree` simply call the corresponding function in `class Node`.

6.4.1 Private members

`Node * rootPtr;`

A pointer to the root node of the tree. This is the first node, and is not the child of any other node in the tree.

`Node * endPtr;`

A pointer to the last node of the tree. This is the last node, and does not have any child.

`Node * currentPtr;`

This pointer points to the current node in the tree. `currentPtr` can be moved up and down in the structure between `rootPtr` and `endPtr`. The current node represents the node from which the user is spending links (making payments).

6.4.2 Constructors

`Tree::Tree()`

Sets all three private members to NULL.

6.4.3 Public functions

Most of the public functions are inline, and quite self-explanatory.

`void insertNode(float face, int n);`

Creates a new node in the tree. Both `currentPtr` and `endPtr` is set to this new node.

`int getDepth(){return endPtr->getDepth();}`

Returns the depth of the tree.

`void getRootId(byte ID[]){rootPtr->getId(ID);}`

Returns the identification number `id` of the tree's root node.

`bool up();`

Moves the current pointer, `currnetPtr`, up one level. That is, the `currentPtr` will point to the parent of the node it just pointed to. If the `currentPtr` is already at the top of the tree, the function returns false. Otherwise it returns true.

`bool down();`

Moves the current pointer, `currnetPtr`, down one level. That is, the `currentPtr` will point to the child of the node it just pointed to. If the `currentPtr` is already at the bottom of the tree, the function returns false. Otherwise it returns true.

`void start();`

Sets the current pointer, `currentPtr`, to point to the root node of the tree (same as `rootPtr`).

`void end();`

Sets the current pointer, `currentPtr`, to point to the last node in the tree (same as `endPtr`).

`int getSignatureSize(){return currentPtr->wChild.getxyLen();}`

Returns the size of the Winternitz signatures used in the tree.

`float getCurrentFace(){return currentPtr->getFace();}`

Returns the face value, `face`, of the node pointed to by `currentPtr`.

`int getCurrentDepth(){return currentPtr->getDepth();}`

Returns the depth (in the tree structure) of the node pointed to by `currentPtr`.

`int getCurrentChainLen(){return currentPtr->getChainLen();}`

Returns the length of the hash chain attached to the node pointed to by `currentPtr`.

```
int getCurrentIndex(){return currentPtr->getIndex();}
```

Returns the index of the current link in the hash chain attached to the node pointed to by `currentPtr`.

```
void getCurrentId(byte ID[]){currentPtr->getId(ID);}
```

The identification number `id` in the node pointed to by `currentPtr` is copied into the argument `ID`.

```
void getCurrentChainRoot(byte cr[]){currentPtr->getChainRoot(cr);}
```

The root of the hash chain attached to the node pointed to by `currentPtr` is copied into the argument `cr`.

```
int getCurrentLink(byte link[]){return currentPtr->getLink(link);}
```

The current link in the hash chain attached to the node pointed to by `currentPtr` is copied into the argument `link`. Note that the index in the hash chain is not updated by this function.

```
int getCurrentLinkNext(byte link[]){return currentPtr-
>getLinkNext(link);}
```

The current link in the hash chain attached to the node pointed to by `currentPtr` is copied into the argument `link`. The hash chain index is incremented one step towards `chainEnd`.

```
bool getCurrentSignature(byte **sign, byte **yTemp)
{return currentPtr->getChildSignature(sign, yTemp);}
```

The `signature`-matrix and the `y`-matrix in the Winternitz signature in the node pointed to by `currentPtr` are copied into the arguments `sign` and `yTemp`.

```
bool getCurrentY(byte **yTemp)
{return currentPtr->wChild.getY(yTemp);}
```

The `y`-matrix in the Winternitz signature in the node pointed to by `currentPtr` is copied into the argument `yTemp`.

`bool currentEmpty(){return currentPtr->index==currentPtr->chainLen;}`

Returns false if the hash chain attached to the node pointed to by `currentPtr` is excused. Returns true otherwise.

6.5 The test programs

A series of tests have been run to find how long some of the key operations for signatures take. These include

Hashing
DSA signatures
RSA signatures
Random number
Winternitz signatures
WinternitzShort signatures

The source code for these tests will not be described in detail here. The reader is referred to the end of Appendix C for the code. The results of some of these tests are discussed in Section 6.6.

Most of the tests for timing have been done on the author's personal computer:

ADM K7
600 MHz
128 MB RAM
Running Windows 2000

The same tests have also been run on different computers to provide more thorough information on the performance. These times are provided in Appendix A.

It would valuable to test the implementations on other operating systems; especially different UNIX flavours. However, this has been left out due to limited access to such systems with the appropriate cryptographic libraries.

6.6 Time requirements and signature sizes

Two different implementations of the Winternitz signature have been done. One is optimised for speed, and the other for minimizing memory requirements for the signer.

Each Winternitz signature has three major components:
The secret matrix `x`.
The public matrix `y`.
The signature matrix `signature`.

In addition, a few other private members are needed to support the classes, the most important being the sub elements that are stored in the matrix `subVal`.

6.6.1 Timing

`class Winternitz` as described above is optimised for quick signing, but can produce rather large signatures. With this implementation a Winternitz signature can be signed about 14 times as fast as a DSA signature and 28 times as fast as an RSA signature. This is after a more complex and time consuming set up of the signature object has already been made, but this set up does not need to be done in real time. If the set up is included, the Winternitz signature is about 4 times as fast as DSA and 7 times as fast an RSA signature.

The verification speeds are the same for both the standard and the short Winternitz signature. Verifying a signature takes about 1.5 longer than an RSA (1024) signature, but it is about 16 faster than the DSA (1024) signature verification.

As mentioned in Section 6.2, the Winternitz size improvement to the Merkle scheme can be adjusted. According to [ME'87], the size can be reduced by a factor between 4 and 8. This can be done in the implementation by changing the value of `elementLen` in `winternitz.h`.

`elementLen` = 8

correspond to a size reduction factor of 8, using one hash sum to sign 8 bits.

`elementLen` = 4

correspond to a size reduction factor of 4, using one hash sum to sign 4 bits.

In the following tables, figures for both of these size factors are given. It is quite clear that a reduction factor of 8 is too much, since the computational times become too large. Using `elementLen=8` with `class Winternitz`, the signature initiation takes about as long as a DSA signature, and the actual signing is only about twice as fast as the DSA signing. Signature verification is also slowed down, but is still to about twice as fast as a DSA signature verification. The figures given where `elementLen=8` are provided to show that larger size reduction factors cannot be used.

The table below show how long it takes to produce each of the private members. Each member takes the same amount of time in the standard and the short version of Winternitz, but the size reduction factor makes a difference. The global constant `elementLen` decides the reduction factor, and times for both 4 and 8 are provided.

	Number of operations needed		Time taken for all operations		
elementLen	4	8	4	8	
Initiate x takes	1 random seeding	1 random seeding	0.71	0.71	ms
Produce x takes	44 random gen.	22 random gen.	1.06	0.53	ms
Produce y takes	704 hash sums	5632 hash sums	4.81	39.5	ms
Produce signature takes	352 hash sums	2816 hash sums	2.41	19.25	ms
Produce subVal	bit shift operations	bit shift operations	0.01	0.01	ms

Table 6.1

The number of operations needed and required time to make the private data members in `class Winternitz` and `class WinternitzShort`.

Table 6.2 shows how long each of the major operations in generating a Winternitz signature takes. The signature objects can be made at any point in time, and even stored for later use if this is convenient. This is the case in the payment system described in Chapter 5. Naturally, both the signing and the signature verifications will be done in real time, as a payee confirms payments from a payer.

		Standard Winternitz		Short Winternitz		
elementLen		4	8	4	8	
Make signature object	Initiate x	0.71	0.71	0.71	0.71	ms
	Produce x	1.06	0.53			ms
	Produce y	4.81	38.5			ms
	Produce subVal	0.01	0.01	0.01	0.01	ms
	Total	6.59	39.75	0.72	0.72	ms
Make a signature	Produce x			1.06	0.53	ms
	Produce y			4.81	38.5	ms
	Produce signature	2.41	19.25	2.41	19.25	ms
	Total	2.41	19.25	8.28	58.28	ms
Verify a signature	Produce subVal	0.01	0.01	0.01	0.01	ms
	Produce signature	2.41	19.25	2.41	19.25	ms
	Total	2.42	19.26	2.42	19.26	ms

Table 6.2
Time required to do the three main operations in `class Winternitz` and `class WinternitzShort`, broken down into each sub operation.

Table 6.3 shows how long it takes to do signing and verification with `class Winternitz`, `class WinternitzShort`, DSA and RSA. A 1024 bit key is used for both DSA and RSA.

elementLen	4	8	
Initialise Winternitz object	6.59	39.75	ms
Make Winternitz signature	2.41	19.25	ms
Verify Winternitz signature	2.42	19.26	ms
Initialise WinternitzShort object	0.72	0.72	ms
Make WinternitzShort signature	8.28	58.28	ms
Verify WinternitzShort signature	2.42	19.26	ms
Make DSA signature	34.18		ms
Verify DSA signature	40.36		ms
Make RSA signature	67.8		ms
Verify RSA signature	1.64		ms

Table 6.3
Time required making and verifying different signatures.

Given the numbers in Table 6.2 and 6.3, we can find a relationship between the key times for the four types of signatures. These are given in Table 6.4.

elementLen	4	8	
Winternitz initiate + sign	3.8	0.58	times faster then DSA signing
Winternitz initiate + sign	7.53	1.15	times faster then RSA signing
Winternitz sign	14.18	1.78	times faster then DSA signing
Winternitz sign	28.13	3.52	times faster then RSA signing
Winternitz verification	16.68	2.1	times faster then DSA verification
Winternitz verification	0.68	0.09	times faster then RSA verification
WinternitzShort initiate + sign	3.8	0.58	times faster then DSA signing
WinternitzShort initiate + sign	7.53	1.15	times faster then RSA signing
WinternitzShort sign	4.13	0.59	times faster then DSA signing
WinternitzShort sign	8.19	1.16	times faster then RSA signing
WinternitzShort verification	16.68	2.1	times faster then DSA verification
WinternitzShort verification	0.68	0.09	times faster then RSA verification

Table 6.4
Relative numbers are given between the times required for signing and verification with `class Winternitz`, `class WinternitzShort`, DSA and RSA.

Lines 3 through 6 in Table 6.4 are of special interest. We can see that a Winternitz signature can be done 14 times faster than a DSA signature and 28 times faster than an RSA signature. Winternitz verification is about 16 times faster than DSA verification, and even though it is slower than the RSA verification, it only takes about 50% longer.

6.6.2 Size

The size of the signatures depends not only on the size reduction factor represented by `elementLen`, but also on the size of the hash digests used, as each sub element is signed with one hash digest. This implementation uses SHA-1 which has digests of size 20 bytes.

Below is a summary of the sizes of the variables used. The data types may vary with the platform and compiler used. The given sizes are for Windows 2000, Microsoft Visual C++.

Data type	Size	
byte*	1	bytes
short unsigned int	2	bytes
long int	4	bytes
*byte is defined as unsigned char		

Table 6.5.
Size of used data types on a PC running Windows 2000, using Microsoft Visual C++.

The value of the global constants listed in Table 6.6 decide the size of the private data members:

	Data member value	
elementLen	4	8
SHA::DIGERSTSIZE	20	20
elementLen	4	8
elementPerByte	2	1
checkLen	2	2

Table 6.6
The values of the global constants, deciding the size of the private arrays and matrices

The tree major matrices (`x`, `y` and `signature`) all has the same size:

`((SHA::DIGESTSIZE +checkLen)*elementPerByte)*SHA::DIGESTSIZE`

= ((20+2) * 2) * 20 = 880 bytes, for `elementPerByte` = 2

= ((20+2) * 1) * 20 = 440 bytes, for `elementPerByte` = 1

The matrix `subVal`'s size:

`sizeof(short unsigned)*((SHA::DIGESTSIZE+checkLen)*elementPerByte)`

= 2 * ((20+2) * 2) = 88 bytes, for elementPerByte = 2

= 2 * ((20+2) * 1) = 44 bytes, for elementPerByte = 1

Given the data in Table 6.5 and 6.6 as well as the two formulas given above, we get the following memory requirements for the `class Winternitz` and `class WinternitzShort`, for both the signer and the verifier, with either 4 or 8 as the size reduction factor.

	Bytes stored by the signer				Bytes stored by the verifier			
	Standard		Short		Standard		Short	
elementLen	4	8	4	8	4	8	4	8
byte **x	880	440						
byte **y	880	440			880	440	880	440
short unsigned int xyLen	2	2			2	2		
byte *subVal	88	44	88	44	88	44		
short unsigned int subLen	2	2			2	2		
byte *m	20	20			20	20	20	20
short unsigned int mLen	2	2	2	2	2	2	2	2
short unsigned int n	2	2			2	2		
byte **signature	880	440			880	440	880	440
long seed			4	4				
Total size	2756	1392	94	50	1876	952	1782	902

Table 6.7:
The number of bytes stored in the private members in class Winternitz and class WinternitzShort.

`class WinternitzShort` reduces the memory requirements while sacrificing computational time. It is a modified version of `class Winternitz`, where most of the private data members are recomputed every time they are needed. The result is quite beneficial for the signer, as the storage requirements are reduced by a factor of about 29. The verifier's storage requirements are reduced only marginally.

6.6.3 Using different hashing algorithms

Hashing is the operation that is done the most in the Winternitz signature. SHA-1 has been chosen simply because it is the standard.

Using MD5 for all hash operations offers advantages regarding both time and space. Depending on what operation we look at, using MD5 cuts down the required time to between 38% and 54% of what is needed while using SHA-1. The signing and verification speeds with `class Winternitz` and `class WinternitzShort` can thus be more then doubled.

MD5 offers smaller signatures as well, since the digests are only 16 bytes as opposed to the 20 bytes digests of SHA-1. Signature sizes can be cut down to about 65% of the size while using SHA-1.

6.6.4 A conclusion

The relative times given in Table 6.4 has a good potential for time saving in payment systems with hash chains.

Since class WinternitzShort offers virtually no size gain for the verifier, it is clear that only the signer should use this class. The signatures produces by the two classes are fully interchangeable, so the signer can use `class WinternitzShort` to sign, and the verifier can use `class Winternitz` for verification. This allows for small storage requirements in small devices like smart cards, while the vendors with larger memory hardware can store more data.

Both classes are optimised to the extreme, one for speed and the other for size. It is quite easy to make a combination, where the required storage will be less than for `class Winternitz`, and the speed will be faster than `class WinternitzShort`.

Bibliography

[An, Be'96]
Anderson, R., Bezuidenhoudt, S., On the Reliability of Electronic Payment Systems. *IEEE Transactions on Software Engineering*, volume 22 number 5, May 1996, pages 294-301.
Available at http://www.cl.cam.ac.uk/~rja14/

[An, Ma, Su'97]
Anderson, R., Manifavas, H., Sutherland, C., NetCard - A practical electronic payment system. In Lomas, M. (ed.), *Proceedings of 1996 International Workshop on Security Protocols*, LNCS, Vol.1189, Springer-Verlag, 1997, Berlin, pages 49-57.
Available at http://www.cl.cam.ac.uk/users/rja14/

[Az'97]
Azbel, I., PayWord Micro-Payment Scheme. Strengths, Weaknesses and Proposed Improvements. BSc (Hons) Thesis, Department of Computer Science, University of Cape Town, South Africa, 1997.
Available at http://www.cs.uct.ac.za/courses/CS400W/NIS/resources.html

[Bl, Io'01]
Blaze, M., Ioannidis, J., Keromytis, A., Offline Micropayments without Trusted Hardware. In *Proceedings of Financial Cryptography, 2001.* To appear.
Available at http://www.crypto.com/papers/

[Bl, Ma'94]
Bleichenbacher, D., Maurer, U., Directed acyclic graphs, one-way functions and digital signatures. In Desmedt, Y. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '94*, LNCS Vol. 839, Springer-Verlag, 1994, Berlin, pages 75-82.
Available at http://www.bell-labs.com/user/bleichen/bib.html

[Br'99]
Brands, S., Electronic Cash. In Atallah, M. (ed.), *Algorithms and Theory of Computation Handbook*, CRC Press LLC, New York, 1999, pages 44.1-44.40.

[CAFE]
Boly, J., Bosselaers, A., Cramer, R., Michelsen, R., Mjølnse, S., Muller, F., Pedersen, T., Pfitzmann, B., de Rooij, P., Schonmaker, B., Schunter, M., Vallée, L. and Waidner, M., The ESPRIT Project CAFÉ - High security digital payment systems. In *Computer Security - Proceedings of ESORICS '94*, LNCS Vol. 875, Springer-Verlag, 1994, Berlin, pages 217-230.

[Ch'82]
Chaum, D., Blind signatures for untraceable payments. In Beth, T. (ed.), Cryptography, Proceedings 1982, LNCS, Vol.149, Springer-Verlag, 1983, Berlin, pages 199-203.

[Ch, Fi, Na'88]
Chaum, D., Fiat, A. and Naor, M., Untraceable electronic cash, In Goldwasser, S. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '88*, LNCS, Vol.403, Springer-Verlag, 1990, Berlin, pages 320-327.

[Chi'97]
Chi, E., Evaluation of Micropayment Schemes. Hewlett Packard Technical Report HPL-97-14, 1997.
Available at http://www.hpl.hp.com/techreports/97/HPL-97-14.html

[Cho, Na, Pu, Un'98]
Chomicki, J., Naqvi, S., Pucci, M., Underwood, R., Decentralised Micropayment Consolidation. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, IEEE, 1998, pages. 332-341.

[Co, Ty, Si'95]
Cox, B., Tygar, J., and Sirbu, M., NetBill security and transaction protocol. In *Proceedings of the First USENIX workshop on electronic commerce*, New York, 1995, pages77-88.
Available from http://www.ini.cmu.edu/NETBILL

[Da'88]
Damgård, I., Payment systems and credential mechanisms with provable security against abuse by individuals. In Goldwasser, S. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '88*, LNCS, Vol.403, Springer-Verlag, 1990, Berlin, pages 328-335.

[Dai'01]
Dai, W., Crypto++ 4.1, a free C++ class library of cryptographic schemes.
Available at http://www.eskimo.com/~weidai/cryptlib.html

[Di, He'76]
Diffie, W., Hellman, M., New Directions in Cryptography. *IEEE Transactions on Information Theory*, IEEE, volume IT-22, number 6, 1976, pages 644-654.
Available at http://www.cs.rutgers.edu/~tdnguyen/classes/cs671/local_papers/diffie-hellman.pdf

[Ev, Go'83]
Even, S., Goldreich, O., Electronic wallet, In McCurley, K. and Ziegler, C. (eds.), *Advances in Cryptology - Proceedings of CRYPTO '83*, LNCS, Vol.1440, Springer-Verlag, 1997, Berlin, pages 383-386

[Fe'93]
Ferguson, N., Extensions of single-term coins. In Stinson, D. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '93*, LNCS, Vol.773, Springer-Verlag, 1994, Berlin, pages 292-301.

[Fe, Da'98]
Ferreira, L., Dahab, R., A scheme for analysing electronic payment schemes. In *Proceedings of the 14th annual Computer Security Applications Conference*, IEEE, 1998, pages 137-146.

[Fi, Od, Si'97]
Fishburn, P., Odlyzko, A., Siders, R., Fixed fee versus unit pricing for information goods: competition, equilibria, and price wars. *First Monday*, volume 2, number 7, July 1997.
Available at http://www.firstmonday.dk/issues/issue2_7/index.html

[Ha, St, Wa'96]
Hauser, R., Steiner, M., Waidner, M., MicroPaymenrs based on iKP. In *Proceedings of the 14th Worldwide Congress on Computer and Communications Security Protection - SECURICOM 96*, 1996, SEDEP, Paris, pages 67-82.
Available at http://www.semper.org/sirene/lit/sirene.lit.html

[He'00]
Heigre, O., One-Time Digital Signatures with Emphasis on Merkle's Authentication Tree. MSc Thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Bergen, Norway, 2000.

[Ho, Pr'98]
Horn, G., Preneel, B., Authentication and Payment in Future Mobile Systems. In *Computer Security - Proceedings of ESORICS '98*, LNCS Vol. 1485, Springer-Verlag, 1998, Berlin, pages 277-293.
Available at http://www.esat.kuleuven.ac.be/cosic/aspect/

[Ja, Od'97]
Jarecki, S., Odylzko A., An efficient micropayment system based on probabilistic polling. In Hirschfeld, R. (ed.), *Proceedings of Financial Cryptography '97*, LNCS, Vol.1318, Springer-Verlag, 1997, Berlin, pages 173-191.

[Ju, Yu'96]
Jutla, C., Yung, M., PayTree: "Amortized-Signature" for Flexible MicroPayments. In *Proceedings of the Second USENIX workshop on electronic commerce*, USENIX, 1996, pages 213-221.

[Ke, Sc'99]
Kelsey, J., Schneier, B., Authenticating Secure Tokens Using Slow Memory Access. USENIX Workshop on Smart Card Technology, USENIX Press, 1999, pages 101-106.
Available at http://www.counterpane.com/slow-memory.html

[Le, Ki'98]
Lee, H., Kim, T., Smart Card Based Off-line Micropayment Framework Using Mutual Authentication Scheme. In the Global Telecommunications Conference - GLOBECOM '98, *The Bridge to Global Integration*, IEEE, volume 4, 1998, pages 2514-2519.

[Li, Os'97]
Lipton, R., Ostrosky, R., Micro-Payments via Efficient Coin-Flipping. In Hirschfeld, R., (ed.), *Proceedings of the 2nd Financial Cryptography Conference - -FC'98*, LNCS Vol. 1465, Springer-Verlag, 1998, Berlin, pages 1-15.
Available at http://www.cl.cam.ac.uk/users/cm213/Project/project_publ.html

[Ma'95]
Manasse, M., The MilliCent Protocols for Electronic Commerce. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, USENIX, 1995, pages 117-123.
Available from http://www.millicent.com/works/details/papers/mcentny.htm

[Mao'96]
Map, W., Lightweight Micro-cash for the Internet. In Bertino, E., Kurth, H., Martella, G., Montolivo, E., (eds.), *Computer Security - Proceedings of ESORICS '96*, LNCS Vol. 1146, Springer-Verlag, 1996, Berlin, pages 15-32.

[Me'87]
Merkle, R., A digital signature based on a conventional encryption function. In Pomerance, C. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '87*, LNCS, Vol.293, Springer-Verlag, 1988, Berlin, pages 369-378.

[Me'89]
Merkle, R., A certified digital signature. In Brassard, G. (ed.), *Advances in Cryptology - Proceedings of CRYPTO '89*, LNCS, Vol.435, Springer-Verlag, 1990, Berlin, pages 218-238.

[Me, Oo, Va'97]
Menezes, A., van Oorschot, P. and Vanstone A., *Handbook Of Applied Cryptography.* CRC Press LLC, 1997.

[Milli'95]
Glassman, S., Manasse, M., Abadi, M., Gauthier, P., Sobalvarro, P.,
The MilliCent Protocol for Inexpensive Electronic Commerce. In *Proceedings of the 4th International World Wide Web Conference*, 1995.
Available at http://www.w3.org/Conferences/WWW4/Papers/246/
http://www.millicent.com/works/details/papers/millicent-w3c4/millicent.html

[Mu, Va, Li'97]
Mu, Y., Varadharajan, V., Lin, Y., New Micropayment Schemes Based on PayWords. In *Proceedings of 2nd Australasian Conference on Information Security and Privacy - ACISP '97*, LNCS, Vol. 1270, Springer-Verlag, 1997, Berlin, pp283-293.
Available at http://www.ics.mq.edu.au/~ymu/pubs.html

[Na, Yu'89]
Naor, M., Yung, M., Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, ACM Press, 1989, New York, pages 33-43.
Available at http://www.wisdom.weizmann.ac.il/~naor/onpub.html

[Ng, Mu, Va'97]
Nguyen, K., Mu, Y., Varadharajan, V., Micro-Digital Money for Electronic Commerce. In *Proceedings of the 13th Annual Computer Security Applications Conference*, IEEE, 1997, pages 2-7.

[Pe'96]
Pedersen, T., Electronic Payments of Small Amounts. In Lomas, M. (ed.), P*roceedings of 1996 International Workshop on Security Protocols*, LNCS, Vol.1189, Springer-Verlag, 1997, Berlin, pages 59-68.

[Po, Hi, St'98]
Poutanene, T., Hinton, H.,Stumm, M., NetCents: A lightweight protocol for secure micropayments. In *Proceedings of the Third USENIX Workshop on Electronic Commerce*, USENIX Association, September 1998, pages 25-36.
Available at
http://www.usenix.org/publications/library/proceedings/ec98/poutanen.html

[Ri'97]
Rivest, R., Electronic Lottery Tickets as Micropayments. In Hirschfeld, R. (ed.), *Proceedings of Financial Cryptography '97*, LNCS, Vol.1318, Springer-Verlag, 1997, Berlin, pages 307-314.

[Ri, Sh'96]
Rivest, R., Shamir, A., PayWord and MicroMint.Two simple micropayment schemes. In Lomas, M. (ed.), *Proceedings of 1996 International Workshop on Security Protocols*, LNCS, Vol.1189, Springer-Verlag, 1997, Berlin, pages 69-87.
Available from http://theory.lcs.mit.edu/~rivest/publications.html

[Ri, Sh, Ad'78]
Rivest, R., Shamir, A., Adleman, L., A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, volume 21, number 2, February 1978, pages 120-126.
Available at http://theory.lcs.mit.edu/~cis/cis-publications.html#1978-1985

[Sc, Mü'97]
Schmidt, C., Müller, R., A Framework for Micropayment Evaluation. First Berlin Internet Economics Workshop, October 1997.
Available at http://mmm.wiwi.hu-berlin.de/~rmueller/paper/

[Si, Ty'95]
Sirbu, M., Tygar, J., NetBill: An Internet Commerce System Optimized for Network Delivered Services. *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers*, 1995, pages 20-25.
and

IEEE Personal Communications , IEEE, volume: 2 issue: 4 , Aug. 1995, pages 34-39.
Available at http://www.ini.cmu.edu/netbill/CompCon.html.
http://www.ini.cmu.edu/NETBILL/pubs.html

[Sh, Sw'98]
Shon, T., Swatman, M., Identifying effectiveness criteria for Internet payment systems, *Internet Research: Electronic Networking Applications and Policy*, Vol. 8, nr. 3, MCB University Press, Bradford, 1998, pages 202-218.

[St, Va'97]
Stern, J., Vaudenay, S., SVP: A Flexible Micropayment System. In Hirschfeld, R. (ed.), *Proceedings of Financial Cryptography '97*, LNCS, Vol.1318, Springer-Verlag, 1997, Berlin, pages 161-171.

Appendix A

A series of time-data have been collected for operations relevant to the system in Chapter 5, Winternitz signatures and implementations. Most of the work has been done on the authors MDA K-7, 600 MHz, but these tests have also been done on two more computers. The three hardware platforms the tests have been done on are:

ADM K-7	Pentium 3	Laptop Celeron
600 MHz	450 MHz	500 MHz
128 MB RAM	128 MB RAM	60 MB RAM
Running Windows 2000	Running Windows NT 4.0	Running Windows 98

		ADM K-7	
	Rounds	Time (ms)	Op. per sec.
Make SHA-1 digest	10000	70	142857.14
Make SHA-1 digest	10000	70	142857.14
Make SHA-1 digest	10000	60	166666.67
Make SHA-1 digest	100000	691	144717.8
Make SHA-1 digest	100000	691	144717.8
Make SHA-1 digest	100000	691	144717.8
Make SHA-1 digest	1000000	6950	143884.89
Make SHA-1 digest	1000000	6920	144508.67
Make SHA-1 digest	1000000	6910	144717.8
Make SHA-1 digest	10000000	68829	145287.6
Make SHA-1 digest	10000000	68799	145350.95
Make SHA-1 digest	10000000	68869	145203.21
Average			146290.62
Make MD5 digest	10000	20	500000
Make MD5 digest	10000	30	333333.33
Make MD5 digest	10000	30	333333.33
Make MD5 digest	100000	260	384615.38
Make MD5 digest	100000	270	370370.37
Make MD5 digest	100000	260	384615.38
Make MD5 digest	1000000	2613	382701.88
Make MD5 digest	1000000	2633	379794.91
Make MD5 digest	1000000	2643	378357.93
Make MD5 digest	10000000	26327	379838.19
Make MD5 digest	10000000	26377	379118.17
Make MD5 digest	10000000	26267	380705.83
Average			382232.06
SHA-1 verification	10000	100	100000
SHA-1 verification	10000	100	100000
SHA-1 verification	10000	100	100000

SHA-1 verification	100000	991	100908.17
SHA-1 verification	100000	1011	98911.97
SHA-1 verification	100000	1031	96993.21
SHA-1 verification	1000000	10015	99850.22
SHA-1 verification	1000000	9975	100250.63
SHA-1 verification	1000000	9965	100351.23
Average			99696.16
SHA "manual" verification	100000	20	5000000
SHA "manual" verification	100000	20	5000000
SHA "manual" verification	100000	20	5000000
SHA "manual" verification	1000000	171	5847953.22
SHA "manual" verification	1000000	161	6211180.12
SHA "manual" verification	1000000	161	6211180.12
SHA "manual" verification	10000000	1592	6281407.04
SHA "manual" verification	10000000	1582	6321112.52
SHA "manual" verification	10000000	1587	6301197.23
Average			5797114.47
Make DSA (1024) key pair	1	9343	0.107
Make DSA (1024) key pair	1	9343	0.107
Make DSA (1024) key pair	1	9343	0.107
Make DSA (1024) key pair	1	9353	0.107
Make DSA (1024) key pair	1	9644	0.104
Average			0.106
Make RSA (1024) key pair	1	1011	0.989
Make RSA (1024) key pair	1	1121	0.892
Make RSA (1024) key pair	1	1011	0.989
Make RSA (1024) key pair	1	1011	0.989
Make RSA (1024) key pair	1	1001	0.999
Average			0.972
Make DSA siganture on a SHA-1 digest	1	30	33.33
Make DSA siganture on a SHA-1 digest	1	40	25
Make DSA siganture on a SHA-1 digest	1	30	33.33
Make DSA siganture on a SHA-1 digest	10	350	28.57
Make DSA siganture on a SHA-1 digest	10	350	28.57
Make DSA siganture on a SHA-1 digest	10	340	29.41
Make DSA siganture on a SHA-1 digest	100	3475	28.78
Make DSA siganture on a SHA-1 digest	100	3475	28.78
Make DSA siganture on a SHA-1 digest	100	3465	28.86
Make DSA siganture on a SHA-1 digest	1000	34680	28.84
Make DSA siganture on a SHA-1 digest	1000	34630	28.88
Make DSA siganture on a SHA-1 digest	1000	34710	28.81
Average			29.26
Verify DSA siganture on a SHA-1 digest	1	40	25
Verify DSA siganture on a SHA-1 digest	1	40	25
Verify DSA siganture on a SHA-1 digest	1	40	25
Verify DSA siganture on a SHA-1 digest	10	411	24.33
Verify DSA siganture on a SHA-1 digest	10	401	24.94
Verify DSA siganture on a SHA-1 digest	10	401	24.94
Verify DSA siganture on a SHA-1 digest	100	4026	24.84
Verify DSA siganture on a SHA-1 digest	100	4076	24.53
Verify DSA siganture on a SHA-1 digest	100	4046	24.72

Verify DSA siganture on a SHA-1 digest	1000	40348	24.78
Verify DSA siganture on a SHA-1 digest	1000	40237	24.85
Verify DSA siganture on a SHA-1 digest	1000	40848	24.48
Average			24.78
Make RSA siganture on a SHA-1 digest	1	70	14.29
Make RSA siganture on a SHA-1 digest	1	70	14.29
Make RSA siganture on a SHA-1 digest	1	70	14.29
Make RSA siganture on a SHA-1 digest	10	671	14.9
Make RSA siganture on a SHA-1 digest	10	661	15.13
Make RSA siganture on a SHA-1 digest	10	701	14.27
Make RSA siganture on a SHA-1 digest	100	6689	14.95
Make RSA siganture on a SHA-1 digest	100	6679	14.97
Make RSA siganture on a SHA-1 digest	100	6669	14.99
Make RSA siganture on a SHA-1 digest	1000	66696	14.99
Make RSA siganture on a SHA-1 digest	1000	66746	14.98
Make RSA siganture on a SHA-1 digest	1000	66756	14.98
Average			14.75
Verify RSA siganture on a SHA-1 digest	100	160	625
Verify RSA siganture on a SHA-1 digest	100	160	625
Verify RSA siganture on a SHA-1 digest	100	170	588.24
Verify RSA siganture on a SHA-1 digest	1000	1652	605.33
Verify RSA siganture on a SHA-1 digest	1000	1643	608.64
Verify RSA siganture on a SHA-1 digest	1000	1653	604.96
Average			609.53
Seed a random pool	1000	310	3225.81
Seed a random pool	1000	310	3225.81
Seed a random pool	1000	290	3448.28
Seed a random pool	10000	3014	3317.85
Seed a random pool	10000	3014	3317.85
Seed a random pool	10000	3034	3295.98
Seed a random pool	100000	31154	3209.86
Seed a random pool	100000	30463	3282.67
Seed a random pool	100000	29893	3345.26
Average			3296.6
Seed a random pool and get a long seed	1000	694	1440.92
Seed a random pool and get a long seed	1000	703	1422.48
Seed a random pool and get a long seed	1000	721	1386.96
Seed a random pool and get a long seed	10000	7030	1422.48
Seed a random pool and get a long seed	10000	7057	1417.03
Seed a random pool and get a long seed	10000	6986	1431.43
Seed a random pool and get a long seed	100000	71230	1403.9
Seed a random pool and get a long seed	100000	70491	1418.62
Seed a random pool and get a long seed	100000	73363	1363.08
Average			1411.88
Make random SHASIZE blocks=	1000	30	33333.33
Make random SHASIZE blocks=	1000	20	50000
Make random SHASIZE blocks=	1000	30	33333.33
Make random SHASIZE blocks=	10000	230	43478.26
Make random SHASIZE blocks=	10000	240	41666.67
Make random SHASIZE blocks=	10000	230	43478.26
Make random SHASIZE blocks=	100000	2344	42662.12

Make random SHASIZE blocks=	100000	2344	42662.12
Make random SHASIZE blocks=	100000	2343	42680.32
Average			41477.16
		elementLen=4	
Make subLen and checkSum	10000	110	90909.09
Make subLen and checkSum	10000	100	100000
Make subLen and checkSum	10000	101	99009.9
Make subLen and checkSum	100000	1131	88417.33
Make subLen and checkSum	100000	1002	99800.4
Make subLen and checkSum	100000	991	100908.17
Average			96507.48
		elementLen=8	
Make subLen and checkSum	10000	60	166666.67
Make subLen and checkSum	10000	40	250000
Make subLen and checkSum	10000	50	200000
Make subLen and checkSum	100000	561	178253.12
Make subLen and checkSum	100000	501	199600.8
Make subLen and checkSum	100000	511	195694.72
Average			198369.22
elementLen=4			
Get (copy) the Winternitz signatures	1000	20	50000
Get (copy) the Winternitz signatures	1000	30	33333.33
Get (copy) the Winternitz signatures	1000	30	33333.33
Get (copy) the Winternitz signatures	10000	221	45248.87
Get (copy) the Winternitz signatures	10000	221	45248.87
Get (copy) the Winternitz signatures	10000	220	45454.55
Average			42103.16
Get (copy) the y matrixes	1000	10	100000
Get (copy) the y matrixes	1000	10	100000
Get (copy) the y matrixes	1000	10	100000
Get (copy) the y matrixes	10000	120	83333.33
Get (copy) the y matrixes	10000	120	83333.33
Get (copy) the y matrixes	10000	120	83333.33
Average			91666.67
Make empty Winternitz object	10	70	142.86
Make empty Winternitz object	10	70	142.86
Make empty Winternitz object	10	60	166.67
Make empty Winternitz object	100	691	144.72
Make empty Winternitz object	100	691	144.72
Make empty Winternitz object	100	691	144.72
Make empty Winternitz object	1000	6869	145.58
Make empty Winternitz object	1000	6879	145.37
Make empty Winternitz object	1000	6869	145.58
Make empty Winternitz object	10000	68418	146.16
Make empty Winternitz object	10000	68418	146.16
Make empty Winternitz object	10000	68578	145.82
Average			146.77
Make Winternitz object (inc. signature)	10	100	100
Make Winternitz object (inc. signature)	10	90	111.11
Make Winternitz object (inc. signature)	10	90	111.11
Make Winternitz object (inc. signature)	100	931	107.41

Make Winternitz object (inc. signature)	100	981	101.94
Make Winternitz object (inc. signature)	100	1001	99.9
Make Winternitz object (inc. signature)	1000	9243	108.19
Make Winternitz object (inc. signature)	1000	9343	107.03
Make Winternitz object (inc. signature)	1000	9223	108.42
Make Winternitz object (inc. signature)	10000	92192	108.47
Make Winternitz object (inc. signature)	10000	97350	102.72
Make Winternitz object (inc. signature)	10000	96388	103.75
Average			105.84
Make Winternitz test object	1000	120	8333.33
Make Winternitz test object	1000	110	9090.91
Make Winternitz test object	1000	110	9090.91
Make Winternitz test object	10000	1742	5740.53
Make Winternitz test object	10000	1742	5740.53
Make Winternitz test object	10000	1733	5770.34
Average			7294.43
Make/self verity Winternitz signature	10	120	83.33
Make/self verity Winternitz signature	10	120	83.33
Make/self verity Winternitz signature	10	120	83.33
Make/self verity Winternitz signature	100	1182	84.6
Make/self verity Winternitz signature	100	1182	84.6
Make/self verity Winternitz signature	100	1232	81.17
Make/self verity Winternitz signature	1000	11867	84.27
Make/self verity Winternitz signature	1000	11878	84.19
Make/self verity Winternitz signature	1000	11867	84.27
Make/self verity Winternitz signature	10000	131139	76.25
Make/self verity Winternitz signature	10000	120864	82.74
Make/self verity Winternitz signature	10000	120974	82.66
Average			82.9
Update empty Winternitz signature	10	30	333.33
Update empty Winternitz signature	10	20	500
Update empty Winternitz signature	10	20	500
Update empty Winternitz signature	100	250	400
Update empty Winternitz signature	100	300	333.33
Update empty Winternitz signature	100	300	333.33
Update empty Winternitz signature	1000	2334	428.45
Update empty Winternitz signature	1000	2444	409.17
Update empty Winternitz signature	1000	2344	426.62
Update empty Winternitz signature	10000	21351	468.36
Update empty Winternitz signature	10000	26478	377.67
Update empty Winternitz signature	10000	25437	393.13
Average			408.62
Verity Winternitz signature	10	20	500
Verity Winternitz signature	10	20	500
Verity Winternitz signature	10	30	333.33
Verity Winternitz signature	100	230	434.78
Verity Winternitz signature	100	260	384.62
Verity Winternitz signature	100	191	523.56
Verity Winternitz signature	1000	2634	379.65
Verity Winternitz signature	1000	2533	394.79
Verity Winternitz signature	1000	2644	378.21

Verity Winternitz signature	10000	28461	351.36
Verity Winternitz signature	10000	23303	429.13
Verity Winternitz signature	10000	24335	410.93
Average			418.36
element Len=4			
Get (produce) WinternitzShort signature	100	821	121.8
Get (produce) WinternitzShort signature	100	861	116.14
Get (produce) WinternitzShort signature	100	841	118.91
Get (produce) WinternitzShort signature	1000	8813	113.47
Get (produce) WinternitzShort signature	1000	8723	114.64
Get (produce) WinternitzShort signature	1000	8603	116.24
Get (produce) WinternitzShort signature	10000	84932	117.74
Get (produce) WinternitzShort signature	10000	85874	116.45
Get (produce) WinternitzShort signature	10000	86885	115.09
Average			116.72
Get (produce) y matrix	100	611	163.67
Get (produce) y matrix	100	621	161.03
Get (produce) y matrix	100	611	163.67
Get (produce) y matrix	1000	6138	162.92
Get (produce) y matrix	1000	6138	162.92
Get (produce) y matrix	1000	6148	162.65
Get (produce) y matrix	10000	61879	161.61
Get (produce) y matrix	10000	61728	162
Get (produce) y matrix	10000	61789	161.84
Average			162.48
Make empty WinternitzShort object	100	70	1428.57
Make empty WinternitzShort object	100	70	1428.57
Make empty WinternitzShort object	100	70	1428.57
Make empty WinternitzShort object	1000	701	1426.53
Make empty WinternitzShort object	1000	701	1426.53
Make empty WinternitzShort object	1000	701	1426.53
Make empty WinternitzShort object	10000	7020	1424.5
Make empty WinternitzShort object	10000	7010	1426.53
Make empty WinternitzShort object	10000	7060	1416.43
Average			1425.86
Make WinternitzShort signature object	100	70	1428.57
Make WinternitzShort signature object	100	70	1428.57
Make WinternitzShort signature object	100	80	1250
Make WinternitzShort signature object	1000	711	1406.47
Make WinternitzShort signature object	1000	711	1406.47
Make WinternitzShort signature object	1000	701	1426.53
Make WinternitzShort signature object	10000	7130	1402.52
Make WinternitzShort signature object	10000	7161	1396.45
Make WinternitzShort signature object	10000	7190	1390.82
Average			1392.93
Update empty WinternitzShort object	10000	90	111111.11
Update empty WinternitzShort object	10000	80	125000
Update empty WinternitzShort object	10000	90	111111.11
Update empty WinternitzShort object	100000	905	110497.24
Update empty WinternitzShort object	100000	910	109890.11
Update empty WinternitzShort object	100000	101	990099.01

Average			259618.1
Verity WinternitzShort signature	100	290	344.83
Verity WinternitzShort signature	100	240	416.67
Verity WinternitzShort signature	100	260	384.62
Verity WinternitzShort signature	1000	2273	439.95
Verity WinternitzShort signature	1000	2373	421.41
Verity WinternitzShort signature	1000	2483	402.74
Verity WinternitzShort signature	10000	25807	387.49
Verity WinternitzShort signature	10000	24765	403.8
Verity WinternitzShort signature	10000	23704	421.87
Average			402.6
elementLen=8			
Get (copy) the Winternitz signatures	10000	130	76923.08
Get (copy) the Winternitz signatures	10000	130	76923.08
Get (copy) the Winternitz signatures	10000	120	83333.33
Get (copy) the Winternitz signatures	100000	1252	79872.2
Get (copy) the Winternitz signatures	100000	1252	79872.2
Get (copy) the Winternitz signatures	100000	1252	79872.2
Average			79466.02
Get (copy) the y matrixes	10000	70	142857.14
Get (copy) the y matrixes	10000	70	142857.14
Get (copy) the y matrixes	10000	70	142857.14
Get (copy) the y matrixes	100000	701	142653.35
Get (copy) the y matrixes	100000	701	142653.35
Get (copy) the y matrixes	100000	691	144717.8
Average			143099.32
Make empty Winternitz object	10	410	24.39
Make empty Winternitz object	10	400	25
Make empty Winternitz object	10	400	25
Make empty Winternitz object	100	4065	24.6
Make empty Winternitz object	100	4045	24.72
Make empty Winternitz object	100	4045	24.72
Make empty Winternitz object	1000	40488	24.7
Make empty Winternitz object	1000	40468	24.71
Make empty Winternitz object	1000	40388	24.76
Average			24.73
Make Winternitz object (inc. signature)	10	630	15.87
Make Winternitz object (inc. signature)	10	591	16.92
Make Winternitz object (inc. signature)	10	581	17.21
Make Winternitz object (inc. signature)	100	6049	16.53
Make Winternitz object (inc. signature)	100	6409	15.6
Make Winternitz object (inc. signature)	100	5869	17.04
Make Winternitz object (inc. signature)	1000	62119	16.1
Make Winternitz object (inc. signature)	1000	60347	16.57
Make Winternitz object (inc. signature)	1000	56752	17.62
Average			16.61
Make Winternitz test object	1000	60	16666.67
Make Winternitz test object	1000	60	16666.67
Make Winternitz test object	1000	61	16393.44
Make Winternitz test object	10000	615	16260.16
Make Winternitz test object	10000	608	16447.37

Make Winternitz test object	10000	610	16393.44
Average			16471.29
Make/self verity Winternitz signature	10	802	12.47
Make/self verity Winternitz signature	10	801	12.48
Make/self verity Winternitz signature	10	811	12.33
Make/self verity Winternitz signature	100	8022	12.47
Make/self verity Winternitz signature	100	7972	12.54
Make/self verity Winternitz signature	100	7961	12.56
Make/self verity Winternitz signature	1000	79905	12.51
Make/self verity Winternitz signature	1000	79825	12.53
Make/self verity Winternitz signature	1000	79764	12.54
Average			12.49
Update empty Winternitz signature	10	211	47.39
Update empty Winternitz signature	10	180	55.56
Update empty Winternitz signature	10	180	55.56
Update empty Winternitz signature	100	2003	49.93
Update empty Winternitz signature	100	2374	42.12
Update empty Winternitz signature	100	1833	54.56
Update empty Winternitz signature	1000	21591	46.32
Update empty Winternitz signature	1000	19888	50.28
Update empty Winternitz signature	1000	16233	61.6
Average			51.48
Verity Winternitz signature	10	170	58.82
Verity Winternitz signature	10	210	47.62
Verity Winternitz signature	10	220	45.45
Verity Winternitz signature	100	1953	51.2
Verity Winternitz signature	100	1602	62.42
Verity Winternitz signature	100	2113	47.33
Verity Winternitz signature	1000	17716	56.45
Verity Winternitz signature	1000	19548	51.16
Verity Winternitz signature	1000	23003	43.47
Average			51.55
element Len=8			
Get (produce) WinternitzShort signature	10	581	17.21
Get (produce) WinternitzShort signature	10	591	16.92
Get (produce) WinternitzShort signature	10	601	16.64
Get (produce) WinternitzShort signature	100	5998	16.67
Get (produce) WinternitzShort signature	100	6179	16.18
Get (produce) WinternitzShort signature	100	6359	15.73
Get (produce) WinternitzShort signature	1000	59846	16.71
Get (produce) WinternitzShort signature	1000	56431	17.72
Get (produce) WinternitzShort signature	1000	56251	17.78
Average			16.84
Get (produce) y matrix	10	400	25
Get (produce) y matrix	10	400	25
Get (produce) y matrix	10	400	25
Get (produce) y matrix	100	4056	24.65
Get (produce) y matrix	100	4036	24.78
Get (produce) y matrix	100	4066	24.59
Get (produce) y matrix	1000	40468	24.71
Get (produce) y matrix	1000	40468	24.71

Get (produce) y matrix	1000	40488	24.7
Average			24.79
Make empty WinternitzShort object	100	70	1428.57
Make empty WinternitzShort object	100	70	1428.57
Make empty WinternitzShort object	100	60	1666.67
Make empty WinternitzShort object	1000	701	1426.53
Make empty WinternitzShort object	1000	701	1426.53
Make empty WinternitzShort object	1000	691	1447.18
Make empty WinternitzShort object	10000	7108	1406.87
Make empty WinternitzShort object	10000	6911	1446.97
Make empty WinternitzShort object	10000	7058	1416.83
Average			1454.97
Make WinternitzShort signature object	100	80	1250
Make WinternitzShort signature object	100	70	1428.57
Make WinternitzShort signature object	100	70	1428.57
Make WinternitzShort signature object	1000	711	1406.47
Make WinternitzShort signature object	1000	711	1406.47
Make WinternitzShort signature object	1000	701	1426.53
Make WinternitzShort signature object	10000	7014	1425.72
Make WinternitzShort signature object	10000	7119	1404.69
Make WinternitzShort signature object	10000	7102	1408.05
Average			1398.34
Update empty WinternitzShort object	10000	40	250000
Update empty WinternitzShort object	10000	40	250000
Update empty WinternitzShort object	10000	40	250000
Update empty WinternitzShort object	100000	421	237529.69
Update empty WinternitzShort object	100000	440	227272.73
Update empty WinternitzShort object	100000	401	249376.56
Average			244029.83
Verity WinternitzShort signature	10	220	45.45
Verity WinternitzShort signature	10	210	47.62
Verity WinternitzShort signature	10	200	50
Verity WinternitzShort signature	100	1963	50.94
Verity WinternitzShort signature	100	1792	55.8
Verity WinternitzShort signature	100	1602	62.42
Verity WinternitzShort signature	1000	19678	50.82
Verity WinternitzShort signature	1000	23213	43.08
Verity WinternitzShort signature	1000	23263	42.99
Average			49.9

		Pentium 3	
	Rounds	Time (ms)	Op. per sec.
Make SHA-1 digest	10000	90	111111.11
Make SHA-1 digest	10000	90	111111.11
Make SHA-1 digest	10000	80	125000
Make SHA-1 digest	100000	901	110987.79
Make SHA-1 digest	100000	901	110987.79
Make SHA-1 digest	100000	901	110987.79
Make SHA-1 digest	1000000	8952	111706.88

Make SHA-1 digest	1000000	8942	111831.8
Make SHA-1 digest	1000000	8942	111831.8
Make SHA-1 digest	10000000	89338	111934.45
Make SHA-1 digest	10000000	89298	111984.59
Make SHA-1 digest	10000000	89298	111984.59
Average			112621.64
Make MD5 digest	10000	30	333333.33
Make MD5 digest	10000	30	333333.33
Make MD5 digest	10000	40	250000
Make MD5 digest	100000	310	322580.65
Make MD5 digest	100000	310	322580.65
Make MD5 digest	100000	310	322580.65
Make MD5 digest	1000000	3105	322061.19
Make MD5 digest	1000000	3105	322061.19
Make MD5 digest	1000000	3105	322061.19
Make MD5 digest	10000000	31025	322320.71
Make MD5 digest	10000000	31035	322216.85
Make MD5 digest	10000000	31025	322320.71
Average			318120.87
SHA-1 verification	10000	160	62500
SHA-1 verification	10000	160	62500
SHA-1 verification	10000	160	62500
SHA-1 verification	100000	1633	61236.99
SHA-1 verification	100000	1643	60864.27
SHA-1 verification	100000	1633	61236.99
SHA-1 verification	1000000	16323	61263.25
SHA-1 verification	1000000	16323	61263.25
SHA-1 verification	1000000	16323	61263.25
Average			61625.33
SHA "manual" verification	100000	20	5000000
SHA "manual" verification	100000	20	5000000
SHA "manual" verification	100000	10	10000000
SHA "manual" verification	1000000	130	7692307.69
SHA "manual" verification	1000000	130	7692307.69
SHA "manual" verification	1000000	140	7142857.14
SHA "manual" verification	10000000	1362	7342143.91
SHA "manual" verification	10000000	1352	7396449.7
SHA "manual" verification	10000000	1362	7342143.91
Average			7178690
Make DSA (1024) key pair	1	10.936	91.441
Make DSA (1024) key pair	1	10.926	91.525
Make DSA (1024) key pair	1	10.926	91.525
Make DSA (1024) key pair	1	10.929	91.5
Make DSA (1024) key pair	1	10.936	91.441
Average			91.486
Make RSA (1024) key pair	1	1211	0.826
Make RSA (1024) key pair	1	1205	0.83
Make RSA (1024) key pair	1	1201	0.833
Make RSA (1024) key pair	1	1202	0.832
Make RSA (1024) key pair	1	1211	0.826
Average			0.829

Make DSA siganture on a SHA-1 digest	10	420	23.81
Make DSA siganture on a SHA-1 digest	10	410	24.39
Make DSA siganture on a SHA-1 digest	10	410	24.39
Make DSA siganture on a SHA-1 digest	100	4136	24.18
Make DSA siganture on a SHA-1 digest	100	4126	24.24
Make DSA siganture on a SHA-1 digest	100	4146	24.12
Make DSA siganture on a SHA-1 digest	1000	41219	24.26
Make DSA siganture on a SHA-1 digest	1000	41299	24.21
Make DSA siganture on a SHA-1 digest	1000	41700	23.98
Average			24.18
Verify DSA siganture on a SHA-1 digest	10	471	21.23
Verify DSA siganture on a SHA-1 digest	10	481	20.79
Verify DSA siganture on a SHA-1 digest	10	481	20.79
Verify DSA siganture on a SHA-1 digest	100	4817	20.76
Verify DSA siganture on a SHA-1 digest	100	4837	20.67
Verify DSA siganture on a SHA-1 digest	100	4746	21.07
Verify DSA siganture on a SHA-1 digest	1000	48500	20.62
Verify DSA siganture on a SHA-1 digest	1000	46657	21.43
Verify DSA siganture on a SHA-1 digest	1000	48299	20.7
Average			20.9
Make RSA siganture on a SHA-1 digest	10	811	12.33
Make RSA siganture on a SHA-1 digest	10	821	12.18
Make RSA siganture on a SHA-1 digest	10	821	12.18
Make RSA siganture on a SHA-1 digest	100	8181	12.22
Make RSA siganture on a SHA-1 digest	100	8181	12.22
Make RSA siganture on a SHA-1 digest	100	8172	12.24
Make RSA siganture on a SHA-1 digest	1000	81657	12.25
Make RSA siganture on a SHA-1 digest	1000	81668	12.24
Make RSA siganture on a SHA-1 digest	1000	81668	12.24
Average			12.23
Verify RSA siganture on a SHA-1 digest	10	20	500
Verify RSA siganture on a SHA-1 digest	10	20	500
Verify RSA siganture on a SHA-1 digest	10	20	500
Verify RSA siganture on a SHA-1 digest	100	201	497.51
Verify RSA siganture on a SHA-1 digest	100	201	497.51
Verify RSA siganture on a SHA-1 digest	100	211	473.93
Verify RSA siganture on a SHA-1 digest	1000	1993	501.76
Verify RSA siganture on a SHA-1 digest	1000	1992	502.01
Verify RSA siganture on a SHA-1 digest	1000	1992	502.01
Average			497.19
Seed a random pool	1000	22030	45.39
Seed a random pool	1000	22030	45.39
Seed a random pool	1000	21930	45.6
Seed a random pool	10000	222310	44.98
Seed a random pool	10000	222320	44.98
Seed a random pool	10000	222720	44.9
Seed a random pool	100000	2228500	44.87
Seed a random pool	100000	2223790	44.97
Seed a random pool	100000	2225800	44.93
Average			45.11
Seed a random pool and get a long seed	1000	22730	43.99

Seed a random pool and get a long seed	1000	22730	43.99
Seed a random pool and get a long seed	1000	22830	43.8
Seed a random pool and get a long seed	10000	227430	43.97
Seed a random pool and get a long seed	10000	227920	43.88
Seed a random pool and get a long seed	10000	227820	43.89
Seed a random pool and get a long seed	100000	2278380	43.89
Seed a random pool and get a long seed	100000	2274270	43.97
Seed a random pool and get a long seed	100000	2276170	43.93
Average			43.92
Make random SHASIZE blocks=	10000	300	33333.33
Make random SHASIZE blocks=	10000	200	50000
Make random SHASIZE blocks=	10000	300	33333.33
Make random SHASIZE blocks=	100000	2800	35714.29
Make random SHASIZE blocks=	100000	2910	34364.26
Make random SHASIZE blocks=	100000	2900	34482.76
Average			24580.89
elementLen=4			
Get (copy) the Winternitz signatures	1000	30	33333.33
Get (copy) the Winternitz signatures	1000	30	33333.33
Get (copy) the Winternitz signatures	1000	40	25000
Get (copy) the Winternitz signatures	10000	260	38461.54
Get (copy) the Winternitz signatures	10000	260	38461.54
Get (copy) the Winternitz signatures	10000	270	37037.04
Get (copy) the Winternitz signatures	100000	2673	37411.15
Get (copy) the Winternitz signatures	100000	2673	37411.15
Get (copy) the Winternitz signatures	100000	2673	37411.15
Average			35317.8
Get (copy) the y matrixes	1000	20	50000
Get (copy) the y matrixes	1000	20	50000
Get (copy) the y matrixes	1000	10	100000
Get (copy) the y matrixes	10000	150	66666.67
Get (copy) the y matrixes	10000	150	66666.67
Get (copy) the y matrixes	10000	140	71428.57
Average			67460.32
Make empty Winternitz object	10	310	32.26
Make empty Winternitz object	10	300	33.33
Make empty Winternitz object	10	310	32.26
Make empty Winternitz object	100	3054	32.74
Make empty Winternitz object	100	3054	32.74
Make empty Winternitz object	100	3054	32.74
Make empty Winternitz object	1000	30694	32.58
Make empty Winternitz object	1000	30674	32.6
Make empty Winternitz object	1000	30714	32.56
Average			32.65
Make Winternitz object (inc. signature)	10	331	30.21
Make Winternitz object (inc. signature)	10	331	30.21
Make Winternitz object (inc. signature)	10	331	30.21
Make Winternitz object (inc. signature)	100	3355	29.81
Make Winternitz object (inc. signature)	100	3375	29.63
Make Winternitz object (inc. signature)	100	3425	29.2
Make Winternitz object (inc. signature)	1000	33439	29.91

Make Winternitz object (inc. signature)	1000	33969	29.44
Make Winternitz object (inc. signature)	1000	33849	29.54
Average			29.8
Make Winternitz test object	1000	140	7142.86
Make Winternitz test object	1000	140	7142.86
Make Winternitz test object	1000	140	7142.86
Make Winternitz test object	10000	1392	7183.91
Make Winternitz test object	10000	1392	7183.91
Make Winternitz test object	10000	1392	7183.91
Make Winternitz test object	100000	15032	6652.47
Make Winternitz test object	100000	15052	6643.64
Make Winternitz test object	100000	15062	6639.22
Average			6990.63
Make/self verity Winternitz signature	10	370	27.03
Make/self verity Winternitz signature	10	370	27.03
Make/self verity Winternitz signature	10	370	27.03
Make/self verity Winternitz signature	100	3726	26.84
Make/self verity Winternitz signature	100	3726	26.84
Make/self verity Winternitz signature	100	3726	26.84
Make/self verity Winternitz signature	1000	37173	26.9
Make/self verity Winternitz signature	1000	37183	26.89
Make/self verity Winternitz signature	1000	37403	26.74
Average			26.9
Update empty Winternitz signature	10	30	333.33
Update empty Winternitz signature	10	30	333.33
Update empty Winternitz signature	10	30	333.33
Update empty Winternitz signature	100	300	333.33
Update empty Winternitz signature	100	300	333.33
Update empty Winternitz signature	100	340	294.12
Update empty Winternitz signature	1000	2603	384.17
Update empty Winternitz signature	1000	3124	320.1
Update empty Winternitz signature	1000	2994	334
Average			333.23
Verity Winternitz signature	10	40	250
Verity Winternitz signature	10	40	250
Verity Winternitz signature	10	40	250
Verity Winternitz signature	100	350	285.71
Verity Winternitz signature	100	340	294.12
Verity Winternitz signature	100	310	322.58
Verity Winternitz signature	1000	3806	262.74
Verity Winternitz signature	1000	3275	305.34
Verity Winternitz signature	1000	3675	272.11
Average			276.96
element Len=4			
Get (produce) WinternitzShort signature	10	111	90.09
Get (produce) WinternitzShort signature	10	111	90.09
Get (produce) WinternitzShort signature	10	111	90.09
Get (produce) WinternitzShort signature	100	1041	96.06
Get (produce) WinternitzShort signature	100	1061	94.25
Get (produce) WinternitzShort signature	100	1061	94.25
Get (produce) WinternitzShort signature	1000	10405	96.11

Get (produce) WinternitzShort signature	1000	10135	98.67
Get (produce) WinternitzShort signature	1000	10546	94.82
Average			93.83
Get (produce) y matrix	10	80	125
Get (produce) y matrix	10	80	125
Get (produce) y matrix	10	80	125
Get (produce) y matrix	100	771	129.7
Get (produce) y matrix	100	781	128.04
Get (produce) y matrix	100	781	128.04
Get (produce) y matrix	1000	7781	128.52
Get (produce) y matrix	1000	7791	128.35
Get (produce) y matrix	1000	7781	128.52
Average			127.35
Make empty WinternitzShort object	10	220	45.45
Make empty WinternitzShort object	10	220	45.45
Make empty WinternitzShort object	10	220	45.45
Make empty WinternitzShort object	100	2263	44.19
Make empty WinternitzShort object	100	2263	44.19
Make empty WinternitzShort object	100	2263	44.19
Make empty WinternitzShort object	1000	22782	43.89
Make empty WinternitzShort object	1000	22772	43.91
Make empty WinternitzShort object	1000	22822	43.82
Average			44.5
Make WinternitzShort signature object	10	230	43.48
Make WinternitzShort signature object	10	220	45.45
Make WinternitzShort signature object	10	230	43.48
Make WinternitzShort signature object	100	2253	44.39
Make WinternitzShort signature object	100	2263	44.19
Make WinternitzShort signature object	100	2263	44.19
Make WinternitzShort signature object	1000	22853	43.76
Make WinternitzShort signature object	1000	22863	43.74
Make WinternitzShort signature object	1000	22863	43.74
Average			44.05
Update empty WinternitzShort object	1000	10	100000
Update empty WinternitzShort object	1000	10	100000
Update empty WinternitzShort object	1000	10	100000
Update empty WinternitzShort object	10000	100	100000
Update empty WinternitzShort object	10000	90	111111.11
Update empty WinternitzShort object	10000	100	100000
Average			101851.85
Verity WinternitzShort signature	10	30	333.33
Verity WinternitzShort signature	10	30	333.33
Verity WinternitzShort signature	10	30	333.33
Verity WinternitzShort signature	100	381	262.47
Verity WinternitzShort signature	100	341	293.26
Verity WinternitzShort signature	100	351	284.9
Verity WinternitzShort signature	1000	3676	272.03
Verity WinternitzShort signature	1000	3946	253.42
Verity WinternitzShort signature	1000	3535	282.89
Average			294.33
elementLen=8			

Get (copy) the Winternitz signatures	10000	130	76923.08
Get (copy) the Winternitz signatures	10000	130	76923.08
Get (copy) the Winternitz signatures	10000	140	71428.57
Get (copy) the Winternitz signatures	100000	1342	74515.65
Get (copy) the Winternitz signatures	100000	1342	74515.65
Get (copy) the Winternitz signatures	100000	1342	74515.65
Average			74803.61
Get (copy) the y matrixes	10000	80	125000
Get (copy) the y matrixes	10000	80	125000
Get (copy) the y matrixes	10000	70	142857.14
Get (copy) the y matrixes	100000	751	133155.79
Get (copy) the y matrixes	100000	751	133155.79
Get (copy) the y matrixes	100000	761	131406.04
Average			131762.46
Make empty Winternitz object	10	781	12.8
Make empty Winternitz object	10	741	13.5
Make empty Winternitz object	10	751	13.32
Make empty Winternitz object	100	7430	13.46
Make empty Winternitz object	100	7420	13.48
Make empty Winternitz object	100	7400	13.51
Make empty Winternitz object	1000	75037	13.33
Make empty Winternitz object	1000	73896	13.53
Make empty Winternitz object	1000	74598	13.41
Average			13.37
Make Winternitz object (inc. signature)	10	1022	9.78
Make Winternitz object (inc. signature)	10	971	10.3
Make Winternitz object (inc. signature)	10	1032	9.69
Make Winternitz object (inc. signature)	100	9734	10.27
Make Winternitz object (inc. signature)	100	10255	9.75
Make Winternitz object (inc. signature)	100	10876	9.19
Make Winternitz object (inc. signature)	1000	97089	10.3
Make Winternitz object (inc. signature)	1000	99332	10.07
Make Winternitz object (inc. signature)	1000	98254	10.18
Average			9.95
Make Winternitz test object	1000	70	14285.71
Make Winternitz test object	1000	80	12500
Make Winternitz test object	1000	70	14285.71
Make Winternitz test object	10000	721	13869.63
Make Winternitz test object	10000	721	13869.63
Make Winternitz test object	10000	731	13679.89
Make Winternitz test object	100000	7821	12786.09
Make Winternitz test object	100000	7791	12835.32
Make Winternitz test object	100000	7781	12851.82
Average			13440.42
Make/self verity Winternitz signature	10	1251	7.99
Make/self verity Winternitz signature	10	1242	8.05
Make/self verity Winternitz signature	10	1241	8.06
Make/self verity Winternitz signature	100	12478	8.01
Make/self verity Winternitz signature	100	12859	7.78
Make/self verity Winternitz signature	100	12458	8.03
Make/self verity Winternitz signature	1000	124589	8.03

Make/self verity Winternitz signature	1000	124459	8.03
Make/self verity Winternitz signature	1000	124487	8.03
Average			8
Update empty Winternitz signature	10	230	43.48
Update empty Winternitz signature	10	230	43.48
Update empty Winternitz signature	10	280	35.71
Update empty Winternitz signature	100	2073	48.24
Update empty Winternitz signature	100	2774	36.05
Update empty Winternitz signature	100	2534	39.46
Update empty Winternitz signature	1000	23083	43.32
Update empty Winternitz signature	1000	25497	39.22
Update empty Winternitz signature	1000	24587	40.67
Average			41.07
Verity Winternitz signature	10	280	35.71
Verity Winternitz signature	10	271	36.9
Verity Winternitz signature	10	220	45.45
Verity Winternitz signature	100	2955	33.84
Verity Winternitz signature	100	2273	43.99
Verity Winternitz signature	100	2503	39.95
Verity Winternitz signature	1000	27169	36.81
Verity Winternitz signature	1000	24926	40.12
Verity Winternitz signature	1000	24587	40.67
Average			39.27
element Len=8			
Get (produce) WinternitzShort signature	10	711	14.06
Get (produce) WinternitzShort signature	10	781	12.8
Get (produce) WinternitzShort signature	10	741	13.5
Get (produce) WinternitzShort signature	100	7371	13.57
Get (produce) WinternitzShort signature	100	7821	12.79
Get (produce) WinternitzShort signature	100	7601	13.16
Get (produce) WinternitzShort signature	1000	71853	13.92
Get (produce) WinternitzShort signature	1000	78223	12.78
Get (produce) WinternitzShort signature	1000	73686	13.57
Average			13.35
Get (produce) y matrix	10	511	19.57
Get (produce) y matrix	10	511	19.57
Get (produce) y matrix	10	521	19.19
Get (produce) y matrix	100	5117	19.54
Get (produce) y matrix	100	5117	19.54
Get (produce) y matrix	100	5117	19.54
Get (produce) y matrix	1000	51044	19.59
Get (produce) y matrix	1000	51043	19.59
Get (produce) y matrix	1000	51023	19.6
Average			19.53
Make empty WinternitzShort object	100	230	434.78
Make empty WinternitzShort object	100	230	434.78
Make empty WinternitzShort object	100	230	434.78
Make empty WinternitzShort object	1000	2283	438.02
Make empty WinternitzShort object	1000	2293	436.11
Make empty WinternitzShort object	1000	2283	438.02
Make empty WinternitzShort object	10000	22882	437.02

Make empty WinternitzShort object	10000	22872	437.22
Make empty WinternitzShort object	10000	22862	437.41
Average			436.46
Make WinternitzShort signature object	100	230	434.78
Make WinternitzShort signature object	100	230	434.78
Make WinternitzShort signature object	100	230	434.78
Make WinternitzShort signature object	1000	2303	434.22
Make WinternitzShort signature object	1000	2293	436.11
Make WinternitzShort signature object	1000	2293	436.11
Make WinternitzShort signature object	10000	22903	436.62
Make WinternitzShort signature object	10000	22913	436.43
Make WinternitzShort signature object	10000	22913	436.43
Average			435.58
Update empty WinternitzShort object	10000	40	250000
Update empty WinternitzShort object	10000	50	200000
Update empty WinternitzShort object	10000	40	250000
Update empty WinternitzShort object	100000	480	208333.33
Update empty WinternitzShort object	100000	520	192307.69
Update empty WinternitzShort object	100000	490	204081.63
Average			217453.78
Verity WinternitzShort signature	10	301	33.22
Verity WinternitzShort signature	10	231	43.29
Verity WinternitzShort signature	10	271	36.9
Verity WinternitzShort signature	100	2704	36.98
Verity WinternitzShort signature	100	2264	44.17
Verity WinternitzShort signature	100	2484	40.26
Verity WinternitzShort signature	1000	29372	34.05
Verity WinternitzShort signature	1000	22612	44.22
Verity WinternitzShort signature	1000	27129	36.86
Average			38.88

		Celeron	
	Rounds	Time (ms)	Op. per sec.
Make SHA-1 digest	10000	160	62500
Make SHA-1 digest	10000	170	58823.53
Make SHA-1 digest	10000	110	90909.09
Make SHA-1 digest	100000	1320	75757.58
Make SHA-1 digest	100000	1320	75757.58
Make SHA-1 digest	100000	1370	72992.7
Make SHA-1 digest	1000000	9170	109051.25
Make SHA-1 digest	1000000	9220	108459.87
Make SHA-1 digest	1000000	9230	108342.36
Make SHA-1 digest	10000000	87770	113934.15
Make SHA-1 digest	10000000	87780	113921.17
Make SHA-1 digest	10000000	87770	113934.15
Average			92031.95
Make MD5 digest	10000	60	166666.67
Make MD5 digest	10000	50	200000
Make MD5 digest	10000	60	166666.67
Make MD5 digest	100000	270	370370.37

Make MD5 digest	100000	270	370370.37
Make MD5 digest	100000	330	303030.3
Make MD5 digest	1000000	3020	331125.83
Make MD5 digest	1000000	3020	331125.83
Make MD5 digest	1000000	3020	331125.83
Make MD5 digest	10000000	30100	332225.91
Make MD5 digest	10000000	30100	332225.91
Make MD5 digest	10000000	30100	332225.91
Average			297263.3
SHA-1 verification	100000	1430	69930.07
SHA-1 verification	100000	1430	69930.07
SHA-1 verification	100000	1370	72992.7
SHA-1 verification	1000000	14010	71377.59
SHA-1 verification	1000000	14010	71377.59
SHA-1 verification	1000000	13950	71684.59
SHA-1 verification	10000000	139900	71479.63
SHA-1 verification	10000000	139840	71510.3
SHA-1 verification	10000000	139780	71540.99
Average			71313.73
SHA "manual" verification	100000	110	909090.91
SHA "manual" verification	100000	110	909090.91
SHA "manual" verification	100000	170	588235.29
SHA "manual" verification	1000000	1370	729927.01
SHA "manual" verification	1000000	1370	729927.01
SHA "manual" verification	1000000	1320	757575.76
SHA "manual" verification	10000000	13290	752445.45
SHA "manual" verification	10000000	13230	755857.9
SHA "manual" verification	10000000	13240	755287.01
Average			765270.81
Make DSA (1024) key pair	1	10.65	93.897
Make DSA (1024) key pair	1	10.65	93.897
Make DSA (1024) key pair	1	10.65	93.897
Make DSA (1024) key pair	1	10.71	93.371
Make DSA (1024) key pair	1	11.15	89.686
Average			92.95
Make RSA (1024) key pair	1	1.65	606.061
Make RSA (1024) key pair	1	1.65	606.061
Make RSA (1024) key pair	1	2.03	492.611
Make RSA (1024) key pair	1	2.03	492.611
Make RSA (1024) key pair	1	2.15	465.116
Average			532.492
Make DSA siganture on a SHA-1 digest	10	770	12.99
Make DSA siganture on a SHA-1 digest	10	770	12.99
Make DSA siganture on a SHA-1 digest	10	770	12.99
Make DSA siganture on a SHA-1 digest	100	4390	22.78
Make DSA siganture on a SHA-1 digest	100	4450	22.47
Make DSA siganture on a SHA-1 digest	100	4450	22.47
Make DSA siganture on a SHA-1 digest	1000	40650	24.6
Make DSA siganture on a SHA-1 digest	1000	40700	24.57
Make DSA siganture on a SHA-1 digest	1000	40700	24.57
Average			20.05

Verify DSA siganture on a SHA-1 digest	10	440	22.73
Verify DSA siganture on a SHA-1 digest	10	490	20.41
Verify DSA siganture on a SHA-1 digest	10	500	20
Verify DSA siganture on a SHA-1 digest	100	4620	21.65
Verify DSA siganture on a SHA-1 digest	100	4670	21.41
Verify DSA siganture on a SHA-1 digest	100	4670	21.41
Verify DSA siganture on a SHA-1 digest	1000	46460	21.52
Verify DSA siganture on a SHA-1 digest	1000	46520	21.5
Verify DSA siganture on a SHA-1 digest	1000	46900	21.32
Average			21.33
Make RSA siganture on a SHA-1 digest	10	760	13.16
Make RSA siganture on a SHA-1 digest	10	820	12.2
Make RSA siganture on a SHA-1 digest	10	830	12.05
Make RSA siganture on a SHA-1 digest	100	7790	12.84
Make RSA siganture on a SHA-1 digest	100	7800	12.82
Make RSA siganture on a SHA-1 digest	100	7800	12.82
Make RSA siganture on a SHA-1 digest	1000	78160	12.79
Make RSA siganture on a SHA-1 digest	1000	78160	12.79
Make RSA siganture on a SHA-1 digest	1000	78220	12.78
Average			12.69
Verify RSA siganture on a SHA-1 digest	100	170	588.24
Verify RSA siganture on a SHA-1 digest	100	220	454.55
Verify RSA siganture on a SHA-1 digest	100	220	454.55
Verify RSA siganture on a SHA-1 digest	1000	1920	520.83
Verify RSA siganture on a SHA-1 digest	1000	1980	505.05
Verify RSA siganture on a SHA-1 digest	1000	1980	505.05
Average			504.71
Seed a random pool	10	550	18.18
Seed a random pool	10	710	14.08
Seed a random pool	10	610	16.39
Seed a random pool	100	4230	23.64
Seed a random pool	100	4230	23.64
Seed a random pool	100	4230	23.64
Seed a random pool	1000	40200	24.88
Seed a random pool	1000	39490	25.32
Seed a random pool	1000	39710	25.18
Average			21.66
Seed a random pool and get a long seed	10	610	16.39
Seed a random pool and get a long seed	10	500	20
Seed a random pool and get a long seed	10	600	16.67
Seed a random pool and get a long seed	100	3850	25.97
Seed a random pool and get a long seed	100	3900	25.64
Seed a random pool and get a long seed	100	3850	25.97
Seed a random pool and get a long seed	1000	43170	23.16
Seed a random pool and get a long seed	1000	42570	23.49
Seed a random pool and get a long seed	1000	42460	23.55
Average			22.32
Make random SHASIZE blocks=	1000	60	16666.67
Make random SHASIZE blocks=	1000	50	20000
Make random SHASIZE blocks=	1000	60	16666.67
Make random SHASIZE blocks=	10000	610	16393.44

Make random SHASIZE blocks=	10000	589	16977.93
Make random SHASIZE blocks=	10000	573	17452.01
Average			11572.97
elementLen=4			
Make empty Winternitz object	10	820	12.2
Make empty Winternitz object	10	770	12.99
Make empty Winternitz object	10	820	12.2
Make empty Winternitz object	100	5110	19.57
Make empty Winternitz object	100	5170	19.34
Make empty Winternitz object	100	5110	19.57
Make empty Winternitz object	1000	48770	20.5
Make empty Winternitz object	1000	48770	20.5
Make empty Winternitz object	1000	48660	20.55
Average			17.49
Make Winternitz object (inc. signature)	10	490	20.41
Make Winternitz object (inc. signature)	10	610	16.39
Make Winternitz object (inc. signature)	10	550	18.18
Make Winternitz object (inc. signature)	100	5000	20
Make Winternitz object (inc. signature)	100	5050	19.8
Make Winternitz object (inc. signature)	100	4940	20.24
Make Winternitz object (inc. signature)	1000	55420	18.04
Make Winternitz object (inc. signature)	1000	54810	18.24
Make Winternitz object (inc. signature)	1000	55200	18.12
Average			18.82
Make Winternitz test object	100	50	2000
Make Winternitz test object	100	60	1666.67
Make Winternitz test object	100	50	2000
Make Winternitz test object	1000	170	5882.35
Make Winternitz test object	1000	110	9090.91
Make Winternitz test object	1000	110	9090.91
Average			4955.14
Make/self verity Winternitz signature	10	550	18.18
Make/self verity Winternitz signature	10	550	18.18
Make/self verity Winternitz signature	10	550	18.18
Make/self verity Winternitz signature	100	5380	18.59
Make/self verity Winternitz signature	100	5390	18.55
Make/self verity Winternitz signature	100	5380	18.59
Make/self verity Winternitz signature	1000	60970	16.4
Make/self verity Winternitz signature	1000	61020	16.39
Make/self verity Winternitz signature	1000	60970	16.4
Average			17.72
Update empty Winternitz signature	10	60	166.67
Update empty Winternitz signature	10	50	200
Update empty Winternitz signature	10	60	166.67
Update empty Winternitz signature	100	270	370.37
Update empty Winternitz signature	100	330	303.03
Update empty Winternitz signature	100	280	357.14
Update empty Winternitz signature	1000	3180	314.47
Update empty Winternitz signature	1000	2690	371.75
Update empty Winternitz signature	1000	2850	350.88
Average			289

Verity Winternitz signature	10	40	250
Verity Winternitz signature	10	50	200
Verity Winternitz signature	10	30	333.33
Verity Winternitz signature	100	330	303.03
Verity Winternitz signature	100	270	370.37
Verity Winternitz signature	100	390	256.41
Verity Winternitz signature	1000	3080	324.68
Verity Winternitz signature	1000	3570	280.11
Verity Winternitz signature	1000	3520	284.09
Average			289.11
element Len=4			
Get (produce) WinternitzShort signature	10	110	90.91
Get (produce) WinternitzShort signature	10	60	166.67
Get (produce) WinternitzShort signature	10	110	90.91
Get (produce) WinternitzShort signature	100	1040	96.15
Get (produce) WinternitzShort signature	100	1050	95.24
Get (produce) WinternitzShort signature	100	1050	95.24
Get (produce) WinternitzShort signature	1000	11370	87.95
Get (produce) WinternitzShort signature	1000	10380	96.34
Get (produce) WinternitzShort signature	1000	10490	95.33
Average			101.64
Get (produce) y matrix	10	50	200
Get (produce) y matrix	10	110	90.91
Get (produce) y matrix	10	110	90.91
Get (produce) y matrix	100	770	129.87
Get (produce) y matrix	100	770	129.87
Get (produce) y matrix	100	770	129.87
Get (produce) y matrix	1000	7800	128.21
Get (produce) y matrix	1000	7690	130.04
Get (produce) y matrix	1000	7640	130.89
Average			128.95
Make empty WinternitzShort object	10	660	15.15
Make empty WinternitzShort object	10	600	16.67
Make empty WinternitzShort object	10	610	16.39
Make empty WinternitzShort object	100	4340	23.04
Make empty WinternitzShort object	100	4290	23.31
Make empty WinternitzShort object	100	4290	23.31
Make empty WinternitzShort object	1000	40750	24.54
Make empty WinternitzShort object	1000	40100	24.94
Make empty WinternitzShort object	1000	40150	24.91
Average			21.36
Make WinternitzShort signature object	10	550	18.18
Make WinternitzShort signature object	10	600	16.67
Make WinternitzShort signature object	10	550	18.18
Make WinternitzShort signature object	100	3850	25.97
Make WinternitzShort signature object	100	3840	26.04
Make WinternitzShort signature object	100	3850	25.97
Make WinternitzShort signature object	1000	43440	23.02
Make WinternitzShort signature object	1000	42790	23.37
Make WinternitzShort signature object	1000	42730	23.4
Average			22.31

Verity WinternitzShort signature	100	330	303.03
Verity WinternitzShort signature	100	330	303.03
Verity WinternitzShort signature	100	330	303.03
Verity WinternitzShort signature	1000	3080	324.68
Verity WinternitzShort signature	1000	3460	289.02
Verity WinternitzShort signature	1000	3350	298.51
Average			303.55
elementLen=8			
Make empty Winternitz object	10	1370	7.3
Make empty Winternitz object	10	1320	7.58
Make empty Winternitz object	10	1370	7.3
Make empty Winternitz object	100	9390	10.65
Make empty Winternitz object	100	9400	10.64
Make empty Winternitz object	100	9450	10.58
Make empty Winternitz object	1000	92940	10.76
Make empty Winternitz object	1000	90290	11.08
Make empty Winternitz object	1000	90850	11.01
Average			9.66
Make Winternitz object (inc. signature)	10	1150	8.7
Make Winternitz object (inc. signature)	10	1210	8.26
Make Winternitz object (inc. signature)	10	1150	8.7
Make Winternitz object (inc. signature)	100	11210	8.92
Make Winternitz object (inc. signature)	100	11200	8.93
Make Winternitz object (inc. signature)	100	11480	8.71
Make Winternitz object (inc. signature)	1000	122260	8.18
Make Winternitz object (inc. signature)	1000	120400	8.31
Make Winternitz object (inc. signature)	1000	113150	8.84
Average			8.62
Make Winternitz test object	1000	60	16666.67
Make Winternitz test object	1000	50	20000
Make Winternitz test object	1000	60	16666.67
Make Winternitz test object	10000	637	15698.59
Make Winternitz test object	10000	619	16155.09
Make Winternitz test object	10000	532	18796.99
Average			17330.67
Make/self verity Winternitz signature	10	1370	7.3
Make/self verity Winternitz signature	10	1370	7.3
Make/self verity Winternitz signature	10	1430	6.99
Make/self verity Winternitz signature	100	13950	7.17
Make/self verity Winternitz signature	100	13950	7.17
Make/self verity Winternitz signature	100	13890	7.2
Make/self verity Winternitz signature	1000	687230	1.46
Make/self verity Winternitz signature	1000	147690	6.77
Make/self verity Winternitz signature	1000	145440	6.88
Average			6.47
Update empty Winternitz signature	10	280	35.71
Update empty Winternitz signature	10	280	35.71
Update empty Winternitz signature	10	220	45.45
Update empty Winternitz signature	100	2300	43.48
Update empty Winternitz signature	100	2310	43.29
Update empty Winternitz signature	100	2470	40.49

Update empty Winternitz signature	1000	27020	37.01
Update empty Winternitz signature	1000	26970	37.08
Update empty Winternitz signature	1000	20320	49.21
Average			40.83
Verity Winternitz signature	10	280	35.71
Verity Winternitz signature	10	170	58.82
Verity Winternitz signature	10	270	37.04
Verity Winternitz signature	100	2690	37.17
Verity Winternitz signature	100	2630	38.02
Verity Winternitz signature	100	2410	41.49
Verity Winternitz signature	1000	22130	45.19
Verity Winternitz signature	1000	28780	34.75
Verity Winternitz signature	1000	26854	37.24
Average			40.6
element Len=8			
Get (produce) WinternitzShort signature	10	720	13.89
Get (produce) WinternitzShort signature	10	720	13.89
Get (produce) WinternitzShort signature	10	770	12.99
Get (produce) WinternitzShort signature	100	7420	13.48
Get (produce) WinternitzShort signature	100	7640	13.09
Get (produce) WinternitzShort signature	100	7250	13.79
Get (produce) WinternitzShort signature	1000	74860	13.36
Get (produce) WinternitzShort signature	1000	74430	13.44
Get (produce) WinternitzShort signature	1000	74970	13.34
Average			13.47
Get (produce) y matrix	10	490	20.41
Get (produce) y matrix	10	490	20.41
Get (produce) y matrix	10	490	20.41
Get (produce) y matrix	100	4990	20.04
Get (produce) y matrix	100	5000	20
Get (produce) y matrix	100	4990	20.04
Get (produce) y matrix	1000	49930	20.03
Get (produce) y matrix	1000	49870	20.05
Get (produce) y matrix	1000	49870	20.05
Average			20.16
Make empty WinternitzShort object	100	770	129.87
Make empty WinternitzShort object	100	600	166.67
Make empty WinternitzShort object	100	720	138.89
Make empty WinternitzShort object	1000	4340	230.41
Make empty WinternitzShort object	1000	4280	233.64
Make empty WinternitzShort object	1000	4400	227.27
Make empty WinternitzShort object	10000	40320	248.02
Make empty WinternitzShort object	10000	40210	248.69
Make empty WinternitzShort object	10000	40150	249.07
Average			208.06
Make WinternitzShort signature object	100	490	204.08
Make WinternitzShort signature object	100	600	166.67
Make WinternitzShort signature object	100	550	181.82
Make WinternitzShort signature object	1000	3900	256.41
Make WinternitzShort signature object	1000	3900	256.41
Make WinternitzShort signature object	1000	3890	257.07

Make WinternitzShort signature object	10000	42890	233.15
Make WinternitzShort signature object	10000	42460	235.52
Make WinternitzShort signature object	10000	42790	233.7
Average			224.98
Verity WinternitzShort signature	10	270	37.04
Verity WinternitzShort signature	10	270	37.04
Verity WinternitzShort signature	10	220	45.45
Verity WinternitzShort signature	100	2420	41.32
Verity WinternitzShort signature	100	2250	44.44
Verity WinternitzShort signature	100	2640	37.88
Verity WinternitzShort signature	1000	24280	41.19
Verity WinternitzShort signature	1000	24220	41.29
Verity WinternitzShort signature	1000	24280	41.19
Average			40.76

Appendix B

Table 6.2 presented times for how long it takes to do some of the operations involved with making `Winternitz` and `WinternitzShort` signatures. Those times, as well as the ones on the left hand side of these tables were taken from individual smaller tests. They are based on timing for producing hash chains (simply repeated hashing), generate random numbers and do a few other calculations. This was done to see where the most time was spent, and thus provide more detailed and vital data.

The values on the right hand side in these two tables are taken from tests run on the `Winternitz` and `WinternitzShort` classes. They are all fraction slower then the ones on the left hand side, as would be expected for an object oriented implementation.

		Standard		Winternitz		
elementLen		4	8	4	8	
Make signature object	Initiate x	0.71	0.71			ms
	Produce x	1.06	0.53			ms
	Produce y	4.81	38.5			ms
	Produce subVal	0.01	0.01			ms
	Total	6.59	39.75	6.81	40.44	ms
Make a signature	Produce x					ms
	Produce y					ms
	Produce signature	2.41	19.25			ms
	Total	2.41	19.25	2.45	19.43	ms
Verify a siganture	Produce subVal	0.01	0.01			ms
	Produce signature	2.41	19.25			ms
	Total	2.42	19.26	2.39	19.4	ms

Table B.1
Detailed time for each sub operation done in `class Winternitz` versus the actual times required to do the actual `class Winternitz` operations

		Short Winternitz				
elementLen		4	8	4	8	
Make signature object	Initiate x	0.71	0.71			ms
	Produce x					ms
	Produce y					ms
	Produce subVal	0.01	0.01			ms
	Total	0.72	0.72	0.72	0.72	ms
Make a signature	Produce x	1.06	0.53			ms
	Produce y	4.81	38.5			ms
	Produce signature	2.41	19.25			ms
	Total	8.28	58.28	8.57	59.38	ms
Verify a siganture	Produce subVal	0.01	0.01			ms
	Produce signature	2.41	19.25			ms
	Total	2.42	19.26	2.48	20.04	ms

Table B.1

Detailed time for each sub operation done in `class WinternitzShort` versus the actual times required to do the actual `class WinternitzShort` operations

Appendix C

class Winternitz

```
/**********************************************************
*winternitz.h
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
**********************************************************/

#ifndef winterntitz_h
#define winterntitz_h
#include <time.h>
#include "sha.h"

USING_NAMESPACE(CryptoPP)


/*Each element to be signed with a hash value. 4 means that each byte
will be divivde into 2*/.
const short unsigned int elementLen=4;
//Nubmer og elements per byte that must be signed with a hash value
const short unsigned int elementPerByte=2;
/*Maximum number of bytes needed for the checksum =
2^elementLen*elementPerByte*digestLen
=2^8*1*160=40960 and takes less then 16 bits
(in binary= 10100000 00000000)
=2^4*2*160=5120 and takes less then 16 bits
(in binary= 00010100 00000000)
*/
const short unsigned int checkLen=sizeof(short unsigned);
const short unsigned int digestLen=SHA::DIGESTSIZE;

class Winternitz
{
      public:
            //Makes a signature object with the x and y values.
            Winternitz();
            ~Winternitz();
            //Makes a signature object, and creates a signature on
the
            //message messDigest
            Winternitz(byte messDigest[], short unsigned int
                              messDigestLen);
            //Produce a signature object used for tesing a signature
            Winternitz(byte messDigest[], short unsigned int
                              messDigestLen, byte **yTest);
```

```
             //Assigns a message to a defined Winternitz object
             bool update(byte messDigest[], short unsigned int
                               messDigestLen);
         //Returns the length of the x and y matrices
             short unsigned getxyLen(){return xyLen;}
             //Returns the object's public y matrix
         bool getY(byte ** yTemp);
         //Returns the public parts of the signature
             bool getSignature(byte **sign, byte **yTemp);
             //Test function. Verifying the objects own signature
             short verifySignature();
             //Tests if the given signature is valid on the object's
             message
             short verifySignature(byte ** testSign);

      private:
            //Creates the x and y matrices
            void initialize();
            void computeSubVal();
            //Make the checksum for the siganture
            void makeCheckSum();
            //Splits a byte into wo bytes, adding 0's as padding
            void splitByte(const byte val, byte * splitArray);
            void produceX();
            void produceY();
        //Makes the signature
            void produceSignature();
            byte **x;
            byte **y;
            short unsigned int xyLen;
            byte *subVal;
            short unsigned int subLen;
            byte *m;
            short unsigned int mLen;
            short unsigned int n;
            byte **signature;
};

#endif



/**********************************************************
*winternitz.cpp
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
**********************************************************/
#include "winternitz.h"
#include <iostream.h>
#include <math.h>
#include <time.h>
#include "osrng.h"


Winternitz::Winternitz()
{
      initialize();
}
```

```
Winternitz::~Winternitz()
{
      short unsigned i=0;//counter
      if(x!=NULL)
      {
            for(i=0; i<xyLen; i++)
                  delete [] x[i];
            delete [] x;
      }
      if(y!=NULL)
      {
            for(i=0; i<xyLen; i++)
                  delete [] y[i];
            delete [] y;
      }
      if(subVal!=NULL)
            delete [] subVal;
      if(signature!=NULL)
      {
            for(i=0; i<subLen; i++)
                  delete [] signature[i];
            delete [] signature;
      }
}

//messDigest is assumed to be a SHA digest
Winternitz::Winternitz(byte messDigest[], short unsigned
messDigestLen)
{
      //Initialize m
      mLen=messDigestLen;
      m=new byte[mLen];
      for(short unsigned int i=0; i<mLen; i++)
            m[i]=messDigest[i];

      n = pow(2,elementLen);//A sub element has a values less then n.

      //Split the bytes in m into shorter elements. Used to find
      //number of hashes needed for signing
      subLen=(digestLen+checkLen)*elementPerByte;
      xyLen=subLen;
      //Long enough to hold the digest plus checksum
      subVal = new byte [subLen];
      computeSubVal();//Computes the sub elements in the digest.
      makeCheckSum();
      produceX();
      produceY();
      produceSignature();
}

//Produce a signature object used for tesing a signature
//No x and y matrices are made.
Winternitz::Winternitz(byte messDigest[], short unsigned int
messDigestLen, byte **yTest)
{
      //Initialize m
      mLen=messDigestLen;
      m=new byte[mLen];
      for(short unsigned int i=0; i<mLen; i++)
            m[i]=messDigest[i];
```

```
        n = pow(2,elementLen);//A sub element has a values less then n.
        subLen=(mLen+checkLen)*elementPerByte;
        xyLen=subLen;
        //Long enough to hold the digest plus checksum
        subVal = new byte [subLen];
        computeSubVal();//Computes the sub elements in the digest.
        makeCheckSum();
        x=NULL;//x is not used in a test object
        signature=NULL;//signature is not used in a test object

        //Allocate memory for y
        y = new byte *[xyLen];
        for(short unsigned int k=0; k<subLen; k++)
        {
                y[k] = new byte[SHA::DIGESTSIZE];
                for(short unsigned int j=0; j<SHA::DIGESTSIZE; j++)
                        y[k][j]=yTest[k][j];
        }
}

/////////////Private functions////////////////////

void Winternitz::initialize()
{
        n = pow(2,elementLen);//A sub element has a values less then n.
        xyLen=(digestLen+checkLen)*elementPerByte;
        subVal=NULL;//The message to sign has not been given yet
        signature=NULL;//The message to sign has not been given yet
        produceX();
        produceY();
}

//Makes the sub elements in subVal. See Witnernitz descriptions in
//chapter 4 for details
void Winternitz::computeSubVal()
{
        short unsigned i;//loop counter
        if(elementPerByte==1)
                for(i=0; i<mLen; i++)
                        subVal[i]=m[i];
        else
                for(i=0; i<mLen; i++)
                        splitByte(m[i],&subVal[i*elementPerByte]);
}

//Creates the check sum for the siganture. See Chapter 4 for details
//This check sum will be appended to subVal
void Winternitz::makeCheckSum()
{
        short unsigned c=0;//the integer value of the checksum
        short unsigned int i=0;//loop counter
        int j=0;//loop counter
        short unsigned cLen=checkLen;
        //The binary representation of the check sum c
        byte * cVal = new byte[checkLen];

        //Compute the check sum
        for(i=0; i<subLen-(checkLen*elementPerByte); i++)
                c+=(n-subVal[i]);
        for(i=0; i<cLen; i++)
```

```
            cVal[i]=0;
      //casts the integer check sum, c, into a byte array, cVal.
      cVal=(byte*)&c;

      int tempVal=subLen-checkLen*elementPerByte;
      short unsigned int k=cLen-1;//Last index of cVal
      if(elementPerByte==1)//cVal can be copied straight into subVal
      //Put cVal into the last indexes of subVal
            for(j=tempVal; j<subLen; j++)
                  subVal[j]=cVal[k--];
      else//byte's in check sum must split like the elements in
subVal
         //Put cVal into the last indexes of subVal
            for(j=tempVal; j<subLen; j+=2)
                  splitByte(cVal[k--], &subVal[j]);
}

//Splits a byte into an array of bytes, padding the high order bits
//with 0
void Winternitz::splitByte(const byte val, byte * splitArray)
{
    //The mask starts out with one 1, and seven 0's: mask=1000000
      byte mask = 128;
    //Once for each element in splitArray
      for(short unsigned int i=0; i<elementPerByte; i++)
      {
            splitArray[i]=0;
            for(short unsigned int j=0; j<elementLen; j++)
            {
                  splitArray[i]<<=1;
                  if(val & mask)//Push 1, else push 0
                        splitArray[i]=splitArray[i]|1;
                  mask>>=1;
            }
      }
}

//Generates a matrix of ramdom numbers.
//These are the secret x values.
//Each x is set to the same length as a SHA digest.
void Winternitz::produceX()
{
      short unsigned i;//loop counter

      //Allocate memory for the x matrix
      x = new byte *[xyLen];
      for(i = 0; i < xyLen; i++)
            x[i] = new byte[SHA::DIGESTSIZE];

      //Create a secret seed
      AutoSeededRandomPool rng;
      long seed=rng.GetLong();

      //Create the x matrix from the seed
    //using a pseudorandom function
      RandomPool randPool;
      randPool.Put((byte*)&seed, sizeof(seed));
      for(i=0; i<xyLen; i++)
            randPool.GenerateBlock(x[i], SHA::DIGESTSIZE);
}
```

```
//Generates a matrix with public y values.
//Each y is a SHA (multiple) digest of the corresponding x.
void Winternitz::produceY()
{

      short unsigned i, j;//loop counters
      //Allocate memory for the y matrix
      y = new byte *[xyLen];
      for(i = 0; i < xyLen; i++)
            y[i] = new byte[SHA::DIGESTSIZE];

      //Each x[i] is hashed n times to produce y[i]
      SHA hash;
      for(i=0; i<xyLen; i++)
      {
            hash.CalculateDigest(y[i], x[i], SHA::DIGESTSIZE);
            for(j=1; j<n; j++)
                  hash.CalculateDigest(y[i], y[i], SHA::DIGESTSIZE);

      }
}

//Creating the signature involves hashing each x[i]
//as many times as the value in subVal[i].
//The result is stored in signature[i]
void Winternitz::produceSignature()
{
      short unsigned i, j, k;//loop conters
      //Allocate memory for the signature matrix
      signature = new byte *[subLen];
      for(i = 0; i < subLen; i++)
            signature[i] = new byte[SHA::DIGESTSIZE];

      //Create the signature
      SHA hash;
      for(i=0; i<subLen; i++)//Once for each x (and y and subVal)
      {
            //The x[i] is hashed subVal[i] times and
        //put into signature[i]
            if(subVal[i]>0)
            {
                  hash.CalculateDigest(signature[i], x[i],
                                                SHA::DIGESTSIZE);
                  //j=0 have been done on the line above
                  for(j=1; j<subVal[i]; j++)
                        hash.CalculateDigest(signature[i],
signature[i],
                                                SHA::DIGESTSIZE);
            }
            //subVal[i] can be 0 in the check sum. x[i] is just
        //copied into signature[i]
            else
                  for(k=0; k<SHA::DIGESTSIZE; k++)
                        signature[i][k]=x[i][k];

      }
}
```

```
////////////Public functions//////////////////////

//Assigns a message to sign to an object. This should only be
//done with an object made with the default constructor. It should
//also only be done once per object for security reasons.
bool Winternitz::update(byte messDigest[], short unsigned int
messDigestLen)
{
      //Security check
      //Comment out if several updates must be called to
    // time the operations
      if(sigantuer!=NULL)
            return false;

      //Initialize m
      mLen=messDigestLen;
      m=new byte[mLen];
      for(short unsigned int i=0; i<mLen; i++)
            m[i]=messDigest[i];
      //Split the bytes in m into shorter elements.
    // Used to find number of hashes needed for signing
      subLen=(digestLen+checkLen)*elementPerByte;
      xyLen=subLen;
      //Long enough to hold the digest pluss checksum
      subVal = new byte [subLen];
      computeSubVal();//Computes the sub values in the digest.
      makeCheckSum();
      produceSignature();
      return true;
}

//Copies the private y-matrix into the argument yTemp
bool Winternitz::getY(byte ** yTemp)
{
      if(!y)
            return false;
      short unsigned i, j;//loop counters
      for(i=0; i<xyLen; i++)
            for(j=0; j<SHA::DIGESTSIZE; j++)
                  yTemp[i][j]=y[i][j];
      return true;
}

//The public parts of the siganture (the y and signature matrices)
//are copied into the argumnts.
//Returns false if no siganture exits, and true otherwise
bool Winternitz::getSignature(byte **sign, byte **yTemp)
{
      if(!signature)
            return false;
      if(!y)
            return false;
      short unsigned i, j;//loop counters
      for(i=0; i<subLen; i++)
            for(j =0; j<SHA::DIGESTSIZE; j++)
            {
                  sign[i][j]=signature[i][j];
```

```
                        yTemp[i][j]=y[i][j];
                }
        return true;
}
//A test function that tests the signature of the signtureobject
//Returns -1 if no siganture exits, 0 if the test fails and 1
//otherwise
short Winternitz::verifySignature()
{
        if(signature==NULL)
                return -1;

        byte tempCheck[SHA::DIGESTSIZE];
        unsigned short i, j, k, t;//loop counters
        SHA hash;
        for(i=0; i<subLen; i++)//Once for each hash value
        {
                for(k=0; k<SHA::DIGESTSIZE; k++)//Copy the siganture to
test
                        tempCheck[k]=signature[i][k];
                //Hash the test siganture as many times as subVal[i]
                for(j=subVal[i]; j<n; j++)
                        hash.CalculateDigest(tempCheck, tempCheck,
                                                         SHA::DIGESTSIZE);
                //It is faster to check the digest "manually" rather
           //then calling hash.VerifyDigest()
           //Test is the test signature is equal to y
                for(t=0; t<SHA::DIGESTSIZE; t++)
                                if(y[i][t]!=tempCheck[t])
                                        return 0;
        }
        return 1;
}

//Tests if the recieved signature is a valid one for the
//signatureobject. Returns -1 if the test cannot be done,
//0 if the test fails and 1 otherwise
short Winternitz::verifySignature(byte **testSign)
{
        if(!subVal)
                return -1;
        if(!y)
                return -1;

        byte tempCheck[SHA::DIGESTSIZE];
        unsigned short i, j, k, t;//loop counters
        SHA hash;
        for(i=0; i<subLen; i++)//Once for each hash value
        {
                for(k=0; k<SHA::DIGESTSIZE; k++)//Copy the siganture to
test
                        tempCheck[k]=testSign[i][k];
                //Hash the test siganture as many times as subVal[i]
                for(j=subVal[i]; j<n; j++)
                        hash.CalculateDigest(tempCheck, tempCheck,
                                                         SHA::DIGESTSIZE);
                //It is fater to check the digests "manually"
           //rather then calling hash.VerifyDigest()
           //Test is the test siganture is equal to y
                for(t=0; t<SHA::DIGESTSIZE; t++)
                                if(y[i][t]!=tempCheck[t])
```

```
                                    return 0;
        }
        return 1;
}
```

class WinternitzShort

```
/********************************************************
*winternitzShort.h
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
*********************************************************/

#ifndef winterntitzshort_h
#define winterntitzshort_h
#include "winternitz.h"//Global constants needed form this class

#include "sha.h"

USING_NAMESPACE(CryptoPP)

class WinternitzShort
{
        public:
              //Makes a signature object with the x and y values.
              WinternitzShort();
              ~WinternitzShort();
              //Makes a signature object, and creates a signature on
          //the message messDigest
              WinternitzShort(byte messDigest[], short unsigned int
                                         messDigestLen);

              //Assigns a message to a defined WinternitzShort object
              bool update(byte messDigest[], short unsigned int
                                 messDigestLen);
              //Returns the length of the x and y matrices
              short unsigned getxyLen(){return xyLen;}
              //Returns the object's public y matrix
              void getY(byte ** yTemp);
              //Returns the public parts of the signature
              bool getSignature(byte **sign, byte **yTemp);
              //Tests if the given siganture is valid on
          //the object's message
              short verifySignature(byte ** testSign);
              //Tests if the given siganture is valid on
          //the object's message
              short verifySignature(byte ** testSign, byte ** testY);
              //Tests if the given siganture is valid on
          //given message
              short verifySignature(byte messDigest[], byte **
testSign,
                                         byte ** testY);

        private:
              void computeSubVal(byte m[]);
              void splitByte(const byte val, byte * splitArray);
              void makeCheckSum();
              void produceX(byte ** xTemp);
```

```
            void produceY(byte ** yTemp);
            //To avoid making the x matrix several times
            void produceY(byte ** yTemp, byte ** x);
            //Makes the signature
            bool produceSignature(byte ** signTemp);
            //To avoid making the x matrix several times
            bool produceSignature(byte ** signTemp, byte ** x);

            long seed;//Secret seed that the x-matrix is based on
            short unsigned int xyLen;
            byte *subVal;
};
#endif


/************************************************************
*winternitzShort.cpp
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
************************************************************/

#include "winternitzshort.h"
#include <iostream.h>
#include <math.h>
#include <time.h>
#include "osrng.h"



WinternitzShort::WinternitzShort()
{
      AutoSeededRandomPool rng;
      seed=rng.GetLong();
      xyLen=(digestLen+checkLen)*elementPerByte;
      subVal=NULL;

}

WinternitzShort::~WinternitzShort()
{
      if(subVal!=NULL)
            delete [] subVal;
}

//messDigest is assumed to be a SHA digest
WinternitzShort::WinternitzShort(byte messDigest[], short unsigned
messDigestLen)
{
      //Make the random seed
      AutoSeededRandomPool rng;
      seed=rng.GetLong();

      xyLen=(digestLen+checkLen)*elementPerByte;
      //Long enough to hold the digest plus checksum
      subVal = new byte [xyLen];
      //Computes the sub values in the digest.
    computeSubVal(messDigest);      makeCheckSum();
}
```

```
/////////////Private functions///////////////////

//Makes the sub elements in subVal. See Witnernitz
//descriptions in Chapter 4 for details
void WinternitzShort::computeSubVal(byte m[])
{
      short unsigned n = pow(2,elementLen);
      short unsigned subLen=(digestLen+checkLen)*elementPerByte;
      short unsigned i;//loop counter
      if(elementPerByte==1)
            for(i=0; i<SHA::DIGESTSIZE; i++)
                  subVal[i]=m[i];
      else
            for(i=0; i<SHA::DIGESTSIZE; i++)
                  splitByte(m[i],
&subVal[i*elementPerByte]);//2==elementPerByte

}

//Creates the check sum for the siganture. See Chapter 4 for details
//This check sum will be appended to subVal
void WinternitzShort::makeCheckSum()
{
      //Find the value of the check sum that will
    //be appended to subVal
      short unsigned c=0;//the integer value of the checksum
      short unsigned int i=0;//counter
      short unsigned cLen=checkLen;
      //The binary reprensentation of the check sum c
      byte * cVal = new byte[checkLen];
      int n=pow(2,elementLen);
      //Compute the check sum
      for(i=0; i<xyLen-(checkLen*elementPerByte); i++)
            c+=(n-subVal[i]);

      for(i=0; i<cLen; i++)
            cVal[i]=0;

      //casts the integer check sum, c, into a byte array, cVal.
      cVal=(byte*)&c;
      int tempVal = xyLen-checkLen*elementPerByte;
      short unsigned int k=cLen-1;//Last index of cVal
      if(elementPerByte==1)//cVal can be copied straight into subVal
            //Put cVal into the last indexes of subVal
            for(int j= tempVal; j<xyLen; j++)
                  subVal[j]=cVal[k--];
      else//bytes the check sum must split like the elements in
subVal
            //Put cVal into the last indexes of subVal
            for(int j= tempVal; j<xyLen; j+=2)
                  splitByte(cVal[k--], &subVal[j]);
}
```

```
//Splits a byte into an array of bytes, padding the high
//order bits with 0
void WinternitzShort::splitByte(const byte val, byte * splitArray)
{
        //The mask starts out with one 1, and seven 0's: mask=1000000
        byte mask = 128;

        //Once for each element in splitArray
        for(short unsigned int i=0; i<elementPerByte; i++)
        {
                splitArray[i]=0;
                for(short unsigned int j=0; j<elementLen; j++)
                {
                        splitArray[i]<<=1;
                        if(val & mask)//Push 1, else push 0
                                splitArray[i]=splitArray[i]|1;
                        mask>>=1;
                }
        }
}

//Generates a matrix of ramdom numbers. These are the
//secret x values.Each x is set to the same length as a SHA digest.
void WinternitzShort::produceX(byte ** xTemp)
{
        //Make the x values form the private seed
        RandomPool randPool;
        randPool.Put((byte*)&seed, sizeof(seed));

        for(short unsigned int j=0; j<xyLen; j++)
                randPool.GenerateBlock(xTemp[j], SHA::DIGESTSIZE);
}


//Generates the public y matrix and puts it in the argument yTemp
//Each y is a SHA digest of the corresponding x.
void WinternitzShort::produceY(byte ** yTemp)
{

        short unsigned int i, j;//loop counters
        short unsigned int n=pow(2,elementLen);
        SHA hash;

        //Need to reproduce the x matrix
        byte **x = new byte *[xyLen];
        for(i = 0; i < xyLen; i++)
                x[i] = new byte[SHA::DIGESTSIZE];
        produceX(x);

        //make the y matrix from the x matrix
        for(i=0; i<xyLen; i++)
        {
                hash.CalculateDigest(yTemp[i], x[i], SHA::DIGESTSIZE);

                for(j=1; j<n; j++)
                        hash.CalculateDigest(yTemp[i], yTemp[i],
                                                        SHA::DIGESTSIZE);
        }
}
```

```
//Generates the public y matrix and puts it in the argument yTemp
//Each y is a SHA digest of the corresponding x.
//The y matrix is based on the x matrix given as an argument
void WinternitzShort::produceY(byte ** yTemp, byte ** x)
{
     short unsigned int i, j;//loop counters
     short unsigned int n=pow(2,elementLen);
     SHA hash;

     for(i=0; i<xyLen; i++)
     {
          hash.CalculateDigest(yTemp[i], x[i], SHA::DIGESTSIZE);

          for(j=1; j<n; j++)
               hash.CalculateDigest(yTemp[i], yTemp[i],
                                       SHA::DIGESTSIZE);
     }
}

//Creating the signature involves hashing each x[i] as many
//times as the value in subVal[i].The result is stored in the
//argument signature[i]. Returns false if the signature
//can not be made, and true otherwise
bool WinternitzShort::produceSignature(byte ** signature)
{
     if(subVal==NULL)//There is no message to produce a signature on
          return false;

     short unsigned int i, j, k;//loop counters
     SHA hash;
     //Need to reproduce the x matrix
     byte **x = new byte *[xyLen];
     for(i = 0; i < xyLen; i++)
          x[i] = new byte[SHA::DIGESTSIZE];
     produceX(x);

     //Create the signature
     for(i=0; i<xyLen; i++)//Once for each x (and y and subVal)
     {
          //The x[i] is hashed subVal[i] times and put
       //into signature[i]
          if(subVal[i]>0)
          {
               hash.CalculateDigest(signature[i], x[i],
                                       SHA::DIGESTSIZE);
               //j=0 have been done on the line above
               for(j=1; j<subVal[i]; j++)
                    hash.CalculateDigest(signature[i],
signature[i],

     SHA::DIGESTSIZE);
          }
          else//subVal[i] can be 0 in the check sum.
               for(k=0; k<SHA::DIGESTSIZE; k++)
                    signature[i][k]=x[i][k];
     }
     return 1;
}

//Does the same as the funtion above, except the x matrix is
//passed as an argument rather the produced by the funtions. This
```

```
//saves time.
//Returns false if the siganture can not be made, and true otherwise
bool WinternitzShort::produceSignature(byte ** signature, byte **x)
{
      if(subVal==NULL)//There is no message to produce a signature on
            return false;

      short unsigned int i, j, k;//loop counters
      SHA hash;



      //Create the signature
      for(i=0; i<xyLen; i++)//Once for each x (and y and subVal)
      {
            //The x[i] is hashed subVal[i] times and put
         //into signature[i]
            if(subVal[i]>0)
            {
                  hash.CalculateDigest(signature[i], x[i],
                                                SHA::DIGESTSIZE);
                  //j=0 have been done on the line above
                  for(j=1; j<subVal[i]; j++)
                        hash.CalculateDigest(signature[i],
signature[i],
                                                SHA::DIGESTSIZE);
            }
            else//subVal[i] can be 0 in the check sum.
                  for(k=0; k<SHA::DIGESTSIZE; k++)
                        signature[i][k]=x[i][k];
      }
      return true;
}

/////////////////Public funtions///////////////////

//Each signature object must only be used on one message.
//Update can not be called on an object that have had x produced
//already. Returns false if x has been produced before.
//Returns true otherwise.
bool WinternitzShort::update(byte messDigest[], short unsigned int
messDigestLen)
{
      //Security check
      //Comment out if several updates must be called to
    // time the operations
      if(subVal!=NULL)
            return false;

      //Split the bytes in m into shorter elements.
    //Used to find number of hashes needed for signing
      xyLen=(digestLen+checkLen)*elementPerByte;
    //Long enough to hold the digest plus checksum
      subVal = new byte [xyLen];
    //Computes the sub elements in the digest.
      computeSubVal(messDigest);
      makeCheckSum();
      return true;
}

//Copies the public y matrix into the argument yTemp
```

```
void WinternitzShort::getY(byte ** yTemp)
{
      produceY(yTemp);
}


//The public parts of the siganture (the y and signature matrices)
//are copied into the arguments. Returns false if the signature can
//not be made. Returns true otherwise.
bool WinternitzShort::getSignature(byte **sign, byte **yTemp)
{
      short unsigned i;
      //Need to reproduce the x matrix
      byte **x = new byte *[xyLen];
      for(i = 0; i < xyLen; i++)
            x[i] = new byte[SHA::DIGESTSIZE];
      produceX(x);

      if(produceSignature(sign, x)==false)
            return false;
      produceY(yTemp, x);
      return true;
}

//Tests if the recieved signature is a valid one for the message
//(subVal) in ths signatureobject. Returns -1 if the signature
//can not be made, 0 if the tet fails and 1 otherwise
short WinternitzShort::verifySignature(byte **testSign)
{
      if(!subVal)//There is no message to produce a signature on
            return -1;
      SHA hash;
      unsigned short i, j;//loop counters
      int n=pow(2,elementLen);
      byte ** y = new byte *[xyLen];

      //Need to make y
      for(i = 0; i < xyLen; i++)
            y[i] = new byte[SHA::DIGESTSIZE];
      produceY(y);

      byte tempCheck[SHA::DIGESTSIZE];
      for(i=0; i<xyLen; i++)//Once for each hash value
      {
            for(j=0; j<SHA::DIGESTSIZE; j++)
                  tempCheck[j]=testSign[i][j];
             //hash tempCheck until it should be the same as
             //the corresponding y
            for(j=subVal[i]; j<n; j++)
                  hash.CalculateDigest(tempCheck, tempCheck,
                                                SHA::DIGESTSIZE);

            //It is faster to check the digest "manually"
         //rather then calling hash.VerifyDigest()
            for(j=0; j<SHA::DIGESTSIZE; j++)
                  if(y[i][j]!=tempCheck[j])
                        return 0;

      }
      return 1;
}

//Tests if the recieved signature and y matrix form a valid signature
```

```
//for the message (subVal) in ths signatureobject. Returns -1 if the
//signature can not be made, 0 if the tet fails and 1 otherwise
short WinternitzShort::verifySignature(byte **testSign, byte **y)
{
    if(!subVal)//There is no messge to produce a signature on
        return -1;
    unsigned short i, j;//loop counters
    int n=pow(2,elementLen);
    SHA hash;
    byte tempCheck[SHA::DIGESTSIZE];



    for(i=0; i<xyLen; i++)//Once for each hash value
    {
        for(j=0; j<SHA::DIGESTSIZE; j++)
            tempCheck[j]=testSign[i][j];
        for(j=subVal[i]; j<n; j++)
            hash.CalculateDigest(tempCheck, tempCheck,
                                        SHA::DIGESTSIZE);
        //It is faster to check the digest "manually" rather then
        //calling hash.VerifyDigest()
        for(j=0; j<SHA::DIGESTSIZE; j++)
                if(y[i][j]!=tempCheck[j])
                    return 0;
    }
    return 1;
}

//Tests if the received signature and y matrix form a valid signature
//for the received message. Used on an empty WinternitzShort object
//made by the default constructor
short WinternitzShort::verifySignature(byte messDigest[], byte
**testSign, byte **y)
{
    short unsigned subLen=(digestLen+checkLen)*elementPerByte;
    subVal = new byte [subLen];

    //Make the subVal matrix based on the messDigest
    computeSubVal(messDigest);
    makeCheckSum();

    unsigned short i, j;//loop counters
    int n=pow(2,elementLen);
    SHA hash;
    byte tempCheck[SHA::DIGESTSIZE];

    for(i=0; i<xyLen; i++)//Once for each hash value
    {
        for(j=0; j<SHA::DIGESTSIZE; j++)
            tempCheck[j]=testSign[i][j];
        for(j=subVal[i]; j<n; j++)
            hash.CalculateDigest(tempCheck, tempCheck,
                                        SHA::DIGESTSIZE);

        //It is faster to check the digest "manually" rather
      //then calling hash.VerifyDigest()
        for(j=0; j<SHA::DIGESTSIZE; j++)
                if(y[i][j]!=tempCheck[j])
                    return 0;

    }
```

```
        return 1;
}
```

class Node

```
/*********************************************************
*node.h
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
*********************************************************/

#ifndef node_h
#define node_h

#include "winternitz.h"
#include "cryptlib.h"

class Tree;
class Node
{
        friend Tree;
        public:
                Node();
                Node(int depth, float face, int n, Node* child=NULL);
                void setChild(Node * c);
                void getId(byte ID[]);
                int getDepth(){return depth;}
                float getFace(){return face;}
                int getChainLen(){return chainLen;}
                int getIndex(){return index;}
                bool getChildSignature(byte **sign, byte **yTemp)
            {return wChild.getSignature(sign, yTemp);}
                void getChainRoot(byte cr[]);
                int getLink(byte link[]);
                int getLinkNext(byte link[]);

        private:
                void getChainEnd(byte cr[]);
                Node * getChild();
                void computeId();
                void generateChain();

                int depth;//Depth in the tree
                float face;//Face value of each link in the local hash
chain
                byte chainRoot[SHA::DIGESTSIZE];//The root of the local
hash chain
                byte chainEnd[SHA::DIGESTSIZE];
                byte id[SHA::DIGESTSIZE];
```

```
        int chainLen;//Length of the local hash chain
        int index;//Current index of the local hash chain
        Winternitz wChild;//Signature object for the child node
        Node * child;//Points to the child node

};
#endif
```

```
/**********************************************************
*node.cpp
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
**********************************************************/

#include "node.h"
#include <iostream.h>
#include "osrng.h"

////////////Constructors//////////////////////

Node::Node(int d, float f, int n, Node* c)
{
    depth=d;
    face=f;
    chainLen=n;
    index=0;
    child=c;
  //Makes the local hash chain. Gives root and chainEnd values
    generateChain();
    computeId();
}

////////////Private functions//////////////////////

void Node::computeId()
{
    byte ** childY=new byte *[wChild.getxyLen()];
    for(int i=0; i<wChild.getxyLen(); i++)
        childY[i] = new byte[SHA::DIGESTSIZE];

  //Get the public y values of the signature on the child
    wChild.getY(childY);

    SHA hash;//A SHA object that will be used for hashing
    //Will hold temporary hash values
    byte childTemp[SHA::DIGESTSIZE];
    byte chainTemp[SHA::DIGESTSIZE];
    byte faceTemp[SHA::DIGESTSIZE];

    int j=0;

    //Put all the public y values into the childTemp array
  //and make a digest (into the same array)
    for(j=0; j<wChild.getxyLen(); j++)
```

```
                hash.Update(childY[j], SHA::DIGESTSIZE);
        hash.Final(childTemp);

        //The face value must be part of the id
        hash.Update(chainRoot, SHA::DIGESTSIZE);
        hash.Final(chainTemp);

        //The face value must be part of the id
        hash.Update((unsigned char*)&face, sizeof(float));
        hash.Final(faceTemp);




        //Make a new digest out of the temporary ones.
      //This new digest is the node id.
        hash.Update(childTemp, SHA::DIGESTSIZE);
        hash.Update(chainTemp, SHA::DIGESTSIZE);
        hash.Update(faceTemp, SHA::DIGESTSIZE);

        hash.Final(id);

}

void Node::setChild(Node * c)
{
        child=c;
        byte childId[SHA::DIGESTSIZE];
        c->getId(childId);
        wChild.update(childId, SHA::DIGESTSIZE);
}


void Node::generateChain()
{
        //Make the random end of the hash chain
        AutoSeededRandomPool rng;
        rng.GenerateBlock(chainEnd, SHA::DIGESTSIZE);
        SHA hash;
      //The root is at least on hash "away" form the end
        hash.CalculateDigest(chainRoot, chainEnd, SHA::DIGESTSIZE);
        //Produce the rest of the links in the hash chain,
      //ending up with root
        for(int j=1; j<chainLen-1; j++)
              hash.CalculateDigest(chainRoot, chainRoot,
SHA::DIGESTSIZE);
}

void Node::getChainEnd(byte ce[])
{
        for(short unsigned int i=0; i<SHA::DIGESTSIZE; i++)
              ce[i]=chainEnd[i];
}

Node * Node::getChild()
{
        return child;
}

////////////Public functions//////////////////////
```

```
void Node::getId(byte ID[])
{
      for(short unsigned int i=0; i<SHA::DIGESTSIZE; i++)
            ID[i]=id[i];
}

void Node::getChainRoot(byte cr[])
{
      for(short unsigned int i=0; i<SHA::DIGESTSIZE; i++)
            cr[i]=chainRoot[i];
}



int Node::getLink(byte link[])
{
      if(index==chainLen)//end of chain
            return -1;

      for(short unsigned int k=0; k<SHA::DIGESTSIZE; k++)
            link[k]=chainEnd[k];

      for(int j=1; j<chainLen-index; j++)
            SHA().CalculateDigest(link, link, SHA::DIGESTSIZE);

      return index;
}

int Node::getLinkNext(byte link[])
{
      if(index==chainLen)//end of chain
            return -1;

      for(short unsigned int k=0; k<SHA::DIGESTSIZE; k++)
            link[k]=chainEnd[k];

      for(int j=1; j<chainLen-index; j++)
            SHA().CalculateDigest(link, link, SHA::DIGESTSIZE);

      return index++;//Return the current index of this link, then
advance the index
}
```

class Tree

```
/***********************************************************
*tree.h
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
***********************************************************/

#ifndef tree_h
#define tree_h

#include "node.h"
```

```
 class Tree
 {
       public:
             Tree();
             void insertNode(float face, int n);//, Node* c=NULL);
             int getDepth(){return endPtr->getDepth();}
             void getRootId(byte ID[]){rootPtr->getId(ID);}
             bool up();
             bool down();
             void start();
             void end();
             int getSignatureSize()
         {return currentPtr->wChild.getxyLen();}


             float getCurrentFace(){return currentPtr->getFace();}
             int getCurrentDepth(){return currentPtr->getDepth();}
             int getCurrentChainLen(){return currentPtr-
 >getChainLen();}
             int getCurrentIndex(){return currentPtr->getIndex();}
             void getCurrentId(byte ID[]){currentPtr->getId(ID);}
             void getCurrentChainRoot(byte cr[])
         {currentPtr->getChainRoot(cr);}
             int getCurrentLink(byte link[])
         {return currentPtr->getLink(link);}
             int getCurrentLinkNext(byte link[])
         {return currentPtr->getLinkNext(link);}
             bool getCurrentSignature(byte **sign, byte **yTemp)
         {return currentPtr->getChildSignature(sign, yTemp);}
             bool getCurrentY(byte **yTemp)
         {return currentPtr->wChild.getY(yTemp);}
             bool currentEmpty()
         {return currentPtr->index==currentPtr->chainLen;}

       private:
             Node * rootPtr;
             Node * currentPtr;
             Node * endPtr;
};
#endif


/************************************************************
*tree.cpp
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
************************************************************/

#include "tree.h"
//#include <iostream.h>

Tree::Tree()
{
      rootPtr=NULL;
      currentPtr=NULL;
      endPtr=NULL;
}

void Tree::insertNode(float face, int n)
```

```
{
      if(rootPtr==NULL)
      {
            rootPtr=new Node(0, face, n);
            currentPtr=rootPtr;
            endPtr=rootPtr;
      }
      else
      {
            Node * temp=new Node(endPtr->getDepth()+1, face, n);
            endPtr->setChild(temp);
            currentPtr=temp;
            endPtr=temp;
      }

}
bool Tree::up()
{
      if(currentPtr==rootPtr)
            return false;
      else
      {
            Node * temp=rootPtr;
            while(temp->getChild()!=currentPtr)
                  temp=temp->getChild();
            currentPtr=temp;
            return true;
      }
}

bool Tree::down()
{
      if(currentPtr==endPtr)
            return false;
      else
      {
            currentPtr=currentPtr->getChild();
            return true;
      }
}

void Tree::start()
{
      currentPtr=rootPtr;
}

void Tree::end()
{
      currentPtr=endPtr;
}
```

Test program

```
/**********************************************************
*test.cpp
*Done as a part of the thesis: "Aspects of Micropayments" by Terje
*Tollisen for his Master of Science (Honours) at University of
*Wollongong.
**********************************************************/
```

```
#include "config.h"
#include "cryptlib.h"
#include "osrng.h"
#include <iostream.h>
#include <iomanip.h>
#include "winternitz.h"
#include "winternitzshort.h"
#include "node.h"
#include "tree.h"
#include <math.h>
#include <time.h>
#include "pch.h"
#include "sha.h"
#include "md5.h"
#include "dsa.h"
#include "rsa.h"
#include "hex.h"
#include "files.h"

USING_NAMESPACE(CryptoPP)
short unsigned int MAX_PHRASE_LENGTH=250;

void help();
void treeTest();
void manualTreeTest();
void nodeTiming(int max, int inc, int len);
void printTree(Tree t);
bool verifyLink(byte root[], byte link[], int i);
bool verifyChild(byte **sign, byte **y, byte*childID);
bool verifyChild(byte **sign, byte **y, byte**childY,
byte*childChainRoot, float face, int signLen);
void hashChainTiming(int max, int inc);
void randomTiming(int max, int inc);
void winternitzTest();
void winternitzTiming(int max, int inc);
void winternitzShortTest();
void winternitzShortTiming(int max, int inc);
void makeKeys();
void signingTest(int max, int inc);

int main(int argc, char* argv[])
{
      int max=0, inc=0, len=0;
      char command[10];

      if(argc==1)
      {
            cout<<endl<<"Enter a command option (h for help):";
            cin>>command;
            if(strcmp(command, "h")==0)
            {
                  help();
                  cout<<endl;
                  return 0;
            }
      }
      else
      {
            strcpy((char*)command,argv[1]);
            if(argc>2)
```

```
                max = atoi(argv[2]);
        if(argc>3)
                inc = atoi(argv[3]);
        if(argc>4)
                len = atoi(argv[4]);
        if(inc<=0)
                inc=max;
}
if(strcmp(command, "h")==0)
{
        help();
        cout<<endl;
        return 0;
}


if(strcmp(command, "tt")==0)
{
        treeTest();
        cout<<endl;
        return 0;
}
if(strcmp(command, "mtt")==0)
{
        manualTreeTest();
        cout<<endl;
        return 0;
}
if(strcmp(command, "wt")==0)
{
        winternitzTest();
        cout<<endl;
        return 0;
}
if(strcmp(command, "wst")==0)
{
        winternitzShortTest();
        cout<<endl;
        return 0;
}
if(strcmp(command, "mk")==0)
{
        makeKeys();
        cout<<endl;
        return 0;
}
if(argc==1)
{
        cout<<"Max nubmer of iterations: ";
        cin>>max;
        cout<<"Size of increments: ";
        cin>>inc;
}

if(strcmp(command, "nt")==0)
{
        if(len==0)
        {
                cout<<"Length of hash chain: ";
                cin>>len;
        }
```

```
                nodeTiming(max, inc, len);
                cout<<endl;
                return 0;
        }
        if(strcmp(command, "hct")==0)
        {
                hashChainTiming(max, inc);
                cout<<endl;
                return 0;
        }
        if(strcmp(command, "rt")==0)
        {
                randomTiming(max, inc);
                cout<<endl;
                return 0;
        }
        if(strcmp(command, "wti")==0)
        {
                winternitzTiming(max, inc);
                cout<<endl;
                return 0;
        }
        if(strcmp(command, "wsti")==0)
        {
                winternitzShortTiming(max, inc);
                cout<<endl;
                return 0;
        }
        if(strcmp(command, "st")==0)
        {
                signingTest(max, inc);
                cout<<endl;
                return 0;
        }

        cout<<endl; return 0;
}



void help()
{
        cout<<endl<<"Program takes 1. 2 or 3 arguments."
                <<endl<<"First argument is a letter code for which "
                <<operation \n\tto perform:"
                <<endl<<"-tt:\tPerform a test on a signature chain as "
                <<"described \n\tin Chapter 5."
                <<endl<<"-mtt:\tPerform a manual test on a signature "
                <<"chain as \n\tdescribed in Chapter 5."
                <<endl<<"+nt:\tTest the time it takes to make a node "
                <<"with a given \n\tnumber size hash chain."
                <<endl<<"+hct:\tTest the time it takes to make and verify
"
                <<"a given \n\tnumber of hashes."
                <<endl<<"+rt:\tTest the time it takes to initialise a "
                <<"random \n\tnumber and do pseudo ramdom operations."
                <<endl<<"-wt:\tPerform tests on the implementation of the
"
                <<"\n\tWinternitz class."
                <<endl<<"+wti:\tTest the time it takes to do operations
on "
```

```
            <<"\n\tthe Winternitz class."
            <<endl<<"-wst:\tPerform tests on the implementation of
the "
            <<"\n\tWinternitzShort class."
            <<endl<<"+wsti:\tTest the time it takes to do operations
"
            <<"on \n\tthe WinternitzShort class."
            <<endl<<"+mk:\tTest the time it takes to make an
RSA(1024) "
            <<"and \n\ta DSA(1024) key pair."
            <<endl<<"+st:\tTest the time it takes to do a given
number "
            <<"of \n\tRSA and DSA operations.";
      cout<<endl<<endl<<"The commands marked with a - takes only one
"
            <<"\nargument (the command option)"
            <<endl<<"The commands marked with a + can take one or two
"
            <<"more options:"
            <<endl<<"1) max nubmer of iterations"
            <<endl<<"2) size of increment";
      cout<<endl<<endl<<"The arguments \"htc 10000\" will casue the "
            <<"program to \ndo tests on 10000 hash chain operatios"
            <<endl<<"The arguments \"htc 10000 5000\" will casue the
"
            <<"program to \ndo tests on 5000 and then 10000 hash
chain "
            <<"operatios"
            <<endl<<"And so on.";
}


void treeTest()
{
      Tree tree;
      tree.insertNode((float)1.1, 2);
      tree.insertNode((float)2.2, 4);
      tree.insertNode((float)3.3, 6);
      tree.insertNode((float)4.4, 8);
      bool test=true;

      int signSize=tree.getSignatureSize();
      int i;

      byte testId[SHA::DIGESTSIZE];
      byte ** testY=new byte*[signSize];
      byte ** testSign=new byte*[signSize];
      byte ** testChildY=new byte*[signSize];
      byte * testChildRoot=new byte[signSize];
      float testChildFace;


      for(i=0; i<signSize; i++)
      {
            testY[i]=new byte[SHA::DIGESTSIZE];
            testSign[i]=new byte[SHA::DIGESTSIZE];
            testChildY[i]=new byte[SHA::DIGESTSIZE];
      }

      tree.start();
      printTree(tree);
```

```
	for(i=1; i<=tree.getDepth(); i++)
	{
		tree.getCurrentSignature(testSign, testY);
		if(!tree.down())
		{
			cout<<endl<<"Unexpexcted end of tree";
			return;
		}

		tree.getCurrentChainRoot(testChildRoot);
		tree.getCurrentY(testChildY);
		testChildFace=tree.getCurrentFace();
		tree.getCurrentId(testId);
		if(!verifyChild(testSign, testY, testChildY,
			testChildRoot, testChildFace, signSize))
		{
			test=false;
			cout<<endl<<"Signature on child "<<i<<" failed";
		}
		else
			cout<<endl<<"Signature on child "<<i<<" ok";

		if(!verifyChild(testSign, testY, testId))
		{
			test=false;
			cout<<endl<<"Signature on child's id "<<i<<"
failed";
		}
		else
			cout<<endl<<"Signature on child's id "<<i<<" ok";
	}

	//Make payments
	cout<<endl<<endl<<"Payment tests";
	byte tempLink[SHA::DIGESTSIZE];
	byte * chainRoot=new byte[signSize];
	int index;
	tree.start();
	while(true)
	{
		printTree(tree);
		while(true)
		{
			tree.getCurrentChainRoot(chainRoot);
			index=tree.getCurrentLinkNext(tempLink);
			if(index==-1)//end of chain
			{
				cout<<endl<<"End of chain";
				break;
			}
			if(!verifyLink(chainRoot, tempLink, index))
			{
				test=false;
				cout<<endl<<"Link verification failed";
			}
			else
				cout<<endl<<"Link verification ok";
		}
		if(!tree.down())
		{
```

```
                    cout<<endl<<"End of tree";
                    break;
                }
        }
        printTree(tree);
        if(test)
                cout<<endl<<endl<<"All tests ran as expected";
        else
                cout<<endl<<endl<<"One or more tests did not go as
expected";
        cout<<endl;

}


void manualTreeTest()
{
        clock_t t1, t2;
        t1 =clock();
        Tree tree;
        int length;//Length of chain to insert
        float face;//Face value for the chain to insert
        t2 =clock();

        //Build the tree
        while(true)
        {
                t1=clock();
                cout<<endl<<"Inserting new node (0 or less to exit):";
                cout<<endl<<"Lenght of hash chain: ";
                cin>>length;
                if(length<=0)
                        break;
                cout<<"Face value per link: ";
                cin>>face;
                t2=clock();
                if(length>0)
                {
                        t1=clock();
                        tree.insertNode(face, length);
                        t2=clock();
                        cout<<"Insert node with a hash chain of
                                        length\t"<<length<<"= "<<
                          (float)(t2-t1)/CLOCKS_PER_SEC
                          <<"\tseconds"<<endl;
                }
        }

        printTree(tree);
        tree.start();

        //Variables needed to test the tree signatures
        int signSize=tree.getSignatureSize();
        int i;

        byte testId[SHA::DIGESTSIZE];
        byte ** testY=new byte*[signSize];
        byte ** testSign=new byte*[signSize];
        byte ** testChildY=new byte*[signSize];
        byte * testChildRoot=new byte[signSize];
        float testChildFace;
```

```
//Allocate memory
for(i=0; i<signSize; i++)
{
      testY[i]=new byte[SHA::DIGESTSIZE];
      testSign[i]=new byte[SHA::DIGESTSIZE];
      testChildY[i]=new byte[SHA::DIGESTSIZE];
}

//Test the signatures on the nodes
cout<<endl<<endl<<"Test the signatures on the nodes:";
tree.start();
while(true)
{
      //Get the siganture of the parent node
      tree.getCurrentSignature(testSign, testY);
      //Move the current point one down; to the child
      if(!tree.down())
            break;

      //Get the three public parts of the child node
      tree.getCurrentChainRoot(testChildRoot);
      tree.getCurrentY(testChildY);
      testChildFace=tree.getCurrentFace();



      //Test the Winternitz signature on the public
  //parts of the child node
      if(!verifyChild(testSign, testY, testChildY,
            testChildRoot, testChildFace, signSize))
            cout<<endl<<"Signature on node "
           <<tree.getCurrentDepth()
           <<" public components failed";
      else
            cout<<endl<<"Signature on node "
            <<tree.getCurrentDepth()
           <<" public components ok";

      //Test the Winternitz signature on the id number
  //of the child node
      tree.getCurrentId(testId);
      if(!verifyChild(testSign, testY, testId))
            cout<<endl<<"Signature on node "
           <<tree.getCurrentDepth()<<" id failed";
      else
            cout<<endl<<"Signature on node "
           <<tree.getCurrentDepth()<<" id ok";
}


//Make payments
cout<<endl<<endl<<"Payment tests";
byte tempLink[SHA::DIGESTSIZE];
byte * chainRoot=new byte[signSize];
int index;
while(true)
{
      printTree(tree);
      cout<<endl<<"Value of next payment: ";
```

```
        cin>>face;
        if(face==0)
            break;
        tree.start();
        while(true)
        {
            if(face == tree.getCurrentFace()
           && !tree.currentEmpty())
            {
                index=tree.getCurrentIndex();
                tree.getCurrentChainRoot(chainRoot);
                tree.getCurrentLinkNext(tempLink);
                if(verifyLink(chainRoot, tempLink, index))
                    cout<<endl<<"Link verification ok";
                else
                    cout<<endl<<"Link verification failed";
                break;
            }
            else
                if(!tree.down())
                {
                    cout<<endl<<"No such value found";
                    cout<<endl<<"To insert a new node with
face
              value "<<face<<",";
                    cout<<endl<<"type length of the new
hash chain,
              (0 to drop insert): ";
                    cin>>length;

                    if(length<=0)
                        break;
                    if(length>0)
                    {
                        t1=clock();
                        tree.insertNode(face, length);
                        t2=clock();
                        cout<<"Insert node with a hash
chain of
                 length\t"<<length<<"= "
                 <<(float)(t2-t1)/CLOCKS_PER_SEC
                 <<"\tseconds"<<endl;
                    }
                    break;
                }
        }
    }
}

void printTree(Tree t)
{
    cout<<endl<<"The tree structure:";
    t.start();
    while(true)
    {
        cout<<endl<<"Depth: "<<t.getCurrentDepth()<<
        "   Face= "<<t.getCurrentFace()<<
        "   Length= "<<t.getCurrentChainLen()<<
        "   Index= "<<t.getCurrentIndex();
        if(!t.down())
            break;
```

```
        }
}

//Use public information to verify a link in a hash chain, compared
to the root of the chain
bool verifyLink(byte root[], byte link[], int i)
{
        byte temp[SHA::DIGESTSIZE];
        for(int k=0; k<SHA::DIGESTSIZE; k++)
                temp[k]=link[k];

        if(i>0)//link is root
                for(int j=0; j<i; j++)
                        SHA().CalculateDigest(temp, temp, SHA::DIGESTSIZE);


        for(int j=0; j<SHA::DIGESTSIZE; j++)
                if(root[j]!=temp[j])
                        return false;
        return true;
}

//Tests if the arguments sign and y makes a valid
//Wintetnitz signature on childID
bool verifyChild(byte **sign, byte **y, byte*childID)
{
        Winternitz testSign(childID, SHA::DIGESTSIZE, y);
        if(!testSign.verifySignature(sign))
                return false;
        return true;
}

//Computes the id of the child node form the arguments
//childY, childChainRoot and face. Tests if the arguments sign
//and y makes a valid Wintetnitz signature on that id
bool verifyChild(byte **sign, byte **y, byte**childY,
byte*childChainRoot, float face, int signLen)
{

        SHA hash;//A SHA object that will be used for hashing
        //Will hold temporary hash values
        byte chainTemp[SHA::DIGESTSIZE];
        byte childTemp[SHA::DIGESTSIZE];
        byte faceTemp[SHA::DIGESTSIZE];
        byte id[SHA::DIGESTSIZE];

        int j=0;

        //Put all the public y values into the
      //childTemp array and make a digest (intot he same array)
        for(j=0; j<signLen; j++)
                hash.Update(childY[j], SHA::DIGESTSIZE);
        hash.Final(childTemp);

        //The denomination of each link must be part of the id
        hash.Update(childChainRoot, SHA::DIGESTSIZE);
        hash.Final(chainTemp);

        //The denomination of each link must be part of the id
        hash.Update((unsigned char*)&face, sizeof(float));
        hash.Final(faceTemp);
```

```
        //Make a new digest out of the temporary ones. This new digest
is the node id.
        hash.Update(childTemp, SHA::DIGESTSIZE);
        hash.Update(chainTemp, SHA::DIGESTSIZE);
        hash.Update(faceTemp, SHA::DIGESTSIZE);
        hash.Final(id);

        Winternitz W(id, SHA::DIGESTSIZE, y);
        if(!W.verifySignature(sign))
              return false;
        return true;
}



//Times how long it takes to make a new signature node
//with a given length of the hash chain
void nodeTiming(int max, int inc, int len)
{
        cout<<endl<<"Node timing"<<endl;
        clock_t t1, t2;
        int i, j;//loop counters
        t1=clock();
        t2 =clock();

        t1=clock();
        Tree tree;
        t2 =clock();



        for( i=inc; i<=max; i+=inc)
        {             t1=clock();
              for(j=0; j<i; j++)
                    tree.insertNode((float)1.1, len);
              t2=clock();
              cout<<"Insert "<<j<<" nodes with lenght\t"
               <<len<<"\tchain=\t"
               <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }

}

//Times how long it takes to produce a hash chain with a
//given number of links
void hashChainTiming(int max, int inc)
{
        cout<<endl<<"Hash chain timing"<<endl;
        clock_t t1, t2;
        SHA hash;
        t1=clock();
        byte m1[SHA::DIGESTSIZE];
        byte m2 [SHA::DIGESTSIZE];
        AutoSeededRandomPool rng;
        rng.GenerateBlock(m1, SHA::DIGESTSIZE);//Make the message
        int i, j;//loop counters
        t2 =clock();
        for( i=inc; i<=max; i+=inc)
        {
              t1=clock();
```

```
		for(j=0; j<i; j++)
			hash.CalculateDigest(m1, m1, SHA::DIGESTSIZE);
		t2=clock();
		cout<<"Make a SHA-1 hash chain of length\t"
		 <<j<<"\t=\t"
		 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
	}

	MD5 md5;
	for( i=inc; i<=max; i+=inc)
	{
		t1=clock();
		for(j=0; j<i; j++)
			md5.CalculateDigest(m1, m1, SHA::DIGESTSIZE);
		t2=clock();
		cout<<"Make a MD5 hash chain of length\t"
		 <<j<<"\t=\t"
		 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
	}


	cout<<endl<<"Hash chain verification timing";
	hash.CalculateDigest(m2, m1, SHA::DIGESTSIZE);
	//m2 is now a digest of m1


	//Test if m2 a digest of m1?
	if (!hash.VerifyDigest(m2, m1, SHA::DIGESTSIZE))
	{
		cout<<endl<<"Hash verification failed"<<endl;
		return;
	}
	else
		cout<<endl<<"Hash verification ok"<<endl;


	for(i=inc; i<=max; i+=inc)
	{
		t1=clock();
		for(j=0; j<i; j++)
			//Test if m2 a digest of m1?
			if (!hash.VerifyDigest(m2, m1, SHA::DIGESTSIZE))
			{
				cout<<endl<<"Hash verification failed while
calling
			hash.VerifyDigest(m2, m1, SHA::DIGESTSIZE)";
				return;
			}
		t2=clock();
		cout<<"Verify\t"<<j<<"\thash values with testing=\t"
		 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
	}

	hash.CalculateDigest(m1, m1, SHA::DIGESTSIZE);
      //m1 and m2 should now be equal
	cout<<endl;

	int k=0;
	for(i=inc*10; i<=max*10; i+=inc*10)
	{
		t1=clock();
```

```
        for(j=0; j<i; j++)
        {
            for(k=0; k<SHA::DIGESTSIZE; k++)
                if(m1[k]!=m2[k])
                {
                    cout<<endl<<"error";
                    return;
                }
        }
        t2=clock();
        cout<<"Verify\t"<<j<<"\thash values manually=\t"
         <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
    }
}

//Times how long it takes to initsialize a random number generator
//and to make a given number of pseudo random numbers
void randomTiming(int max, int inc)
{
    cout<<endl<<"Random number generation";
    clock_t t1, t2;
    t1=clock();
    t2 =clock();
    cout<<endl<<"Test zero time=\t"
      <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;

    int i,j;
    const short unsigned int messLen=SHA::DIGESTSIZE;
    long seed;
    byte mess[messLen];
    AutoSeededRandomPool rng;

    for(i=inc; i<=max; i+=inc)
    {
        t1=clock();
        for(j=0; j<i; j++)
            AutoSeededRandomPool rng;
        t2=clock();
        cout<<"Initsialize\t"<<j
         <<"\trandom number generators=\t"
         <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
    }
    for(i=inc; i<=max; i+=inc)
    {
        t1=clock();
        for(j=0; j<i; j++)
        {
            AutoSeededRandomPool rng;
            seed=rng.GetLong();
        }
        t2=clock();
        cout<<"Initsialize\t"<<j
         <<"\trandom number generator and get a long=\t"
         <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
    }
    for(i=inc; i<=max; i+=inc)
    {
        t1=clock();
        for(j=0; j<i; j++)
                rng.GenerateBlock(mess, SHA::DIGESTSIZE);
        t2=clock();
```

```
              cout<<"Generate\t"<<j<<
            "\trandom numbers of size SHASIZE=\t"
            <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }
}


void winternitzTest()
{
        int i=0;
        bool test=true;

      //A sub element has a values less then n.
        const short unsigned int n = pow(2,elementLen);
        const short unsigned int messLen=SHA::DIGESTSIZE;
        byte mess[messLen];//The message who's digest will be signed

        //Generating a random message
        AutoSeededRandomPool rng;
        rng.GenerateBlock(mess, SHA::DIGESTSIZE);

        //Creating the digest of the message
        byte m [SHA::DIGESTSIZE];//Will hold the digest of the message
        SHA().CalculateDigest(m, mess, messLen);

        Winternitz W1(m, SHA::DIGESTSIZE);
        cout<<endl<<"W1 is a Winternitz sigantue
      object:Winternitz W1(m, SHA::DIGESTSIZE)";
        cout<<endl<<"Self verification on W1 should be ok";
        if(!W1.verifySignature())
        {
              test=false;
              cout<<endl<<"Signature on W1 failed";
        }
        else
              cout<<endl<<"Signature on W1 ok";


        Winternitz W2;
        cout<<endl<<"W2 is an empty Winternitz sigantue object."
              <<endl<<"Calling W2.update(m, SHA::DIGESTSIZE)";
        W2.update(m, SHA::DIGESTSIZE);
        cout<<endl<<"Self verification on W2 should be ok";
        if(!W2.verifySignature())
        {
              test=false;
              cout<<endl<<"Signature on W2 failed";
        }
        else
              cout<<endl<<"Signature on W2 ok";

        //Generate varables need for siganture testing
        short unsigned int signLen=W2.getxyLen();
        byte ** sign=new byte *[signLen];
        byte *subVal=new byte [signLen];
        short unsigned int subLen=W2.getxyLen();
        byte **y=new byte *[signLen];;;
        short testValue;
        //Allocate memory
        for(i=0; i<signLen; i++)
        {
```

```
          y[i]=new byte [SHA::DIGESTSIZE];
          sign[i]=new byte [SHA::DIGESTSIZE];

    }


    cout<<endl<<"Get the public y and sign
  from W2: W2.getSignature(sign, y)";
    W2.getSignature(sign, y);

    cout<<endl<<"W3 is a Winternitz testing
  sigantue: Winternitz W3(m, SHA::DIGESTSIZE, y)";
    Winternitz W3(m, SHA::DIGESTSIZE, y);
    cout<<endl<<"Use W3 to test the sign from W2.
  Signaute test should be ok";
    if(!W3.verifySignature(sign))
    {
          test=false;
          cout<<endl<<"Signature on W3 failed";
    }
    else
          cout<<endl<<"Signature on W3 ok";

    sign[0][0]++;
    cout<<endl<<"Change a nubmer in sign, to make a miss match"
          <<endl<<"Siganture on W3 should now fail";
    if(!W3.verifySignature(sign))
          cout<<endl<<"Signature on W3 failed";
    else
    {
          test=false;
          cout<<endl<<"Signature on W3 ok";
    }
    cout<<endl<<"W4 is an empty Winternitz
  sigantue object: Winternitz W4";
    Winternitz W4;
    cout<<endl<<"Calling W4.verifySignature()"
          <<endl<<"This siganture does not exist,
      the operation and should not be completed.";
    testValue=W4.verifySignature();
    if(testValue==-1)
          cout<<endl<<"Signature W4 could not be completed";
    else
          if(testValue==0)
          {
                test=false;
                cout<<endl<<"Signature on W4 failed";
          }
    else
          if(testValue==1)
          {
                test=false;
                cout<<endl<<"Signature on W4 ok";
          }
    if(test)
          cout<<endl<<endl<<"All tests ran as expected";
    else
          cout<<endl<<endl<<"One or more tests did not
      go as expected";
}
```

```
	{
		t1=clock();
		for(j=0; j<i; j++)
			tempW.getY(tempY);
		t2=clock();
		cout<<"Get (copy)\t"<<j
		 <<"\ty matrixes=\t"
		 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
	}

	Winternitz testW(m, SHA::DIGESTSIZE, tempY);
	if(!testW.verifySignature(tempSign))
	{
		cout<<endl<<"Test signature failed. Abnormal Abort";
		return;
	}

	//Verif the Winternitz siganture on a signature object
	for(i=inc; i<=max; i+=inc)
	{
		t1=clock();
		for(j=0; j<i; j++)
			testW.verifySignature(tempSign);
		t2=clock();
		cout<<"Verity\t"<<j
		 <<"\twintertnitz signatures=\t"
		 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
	}
}



void winternitzShortTest()
{
	//A sub element has a values less then n.
	const short unsigned int n = pow(2,elementLen);
	const short unsigned int messLen=SHA::DIGESTSIZE;
	byte mess[messLen];//The message whos digest will be signed
	bool test=true;

	//Generating a random message
	AutoSeededRandomPool rng;
	rng.GenerateBlock(mess, SHA::DIGESTSIZE);

	//Creating the digest of the message
	byte m [SHA::DIGESTSIZE];//Will hold the digest of the message
	SHA().CalculateDigest(m, mess, messLen);


	WinternitzShort W1(m, SHA::DIGESTSIZE);
	cout<<endl<<"W1 is a WinternitzShort
    object:WinternitzShort W1(m, SHA::DIGESTSIZE).";

	//Generate varables need for siganture testing
	short unsigned int signLen=W1.getxyLen();
	byte ** sign=new byte *[signLen];
	byte *subVal=new byte [signLen];
	byte **y=new byte *[signLen];
	short testValue;
```

```
//Tiems how long it takes to do different actions on a
//Winternitz signature
void winternitzTiming(int max, int inc)
{
      cout<<endl<<"Winternitz timing. lementLen=\t"<<elementLen;
      clock_t t1, t2;
      t1=clock();
      t2 =clock();

      t1=clock();
      //A sub element has a values less then n.
      const short unsigned int n = pow(2,elementLen);
      const short unsigned int messLen=SHA::DIGESTSIZE;
      byte mess[messLen];//The message who's digest will be signed

      //Generating a random message
      AutoSeededRandomPool rng;
      rng.GenerateBlock(mess, SHA::DIGESTSIZE);

      //Creating the digest of the message
      byte m [SHA::DIGESTSIZE];
      SHA().CalculateDigest(m, mess, messLen);

      t2 =clock();
      cout<<"Initsialize time=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;

      int i,j;
      //Makes an empty Winternitz siganture
      for(i=inc; i<=max; i+=inc)
      {
            t1=clock();
            for(j=0; j<i; j++)
                  Winternitz W;
            t2=clock();
            cout<<"Make\t"<<j<<"\tempty wintertnitz signatures=\t"
             <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
      }
      //Updates an empty Winternitz siganture
      for(i=inc; i<=max; i+=inc)
      {
            Winternitz W;
            t1=clock();
            for(j=0; j<i; j++)
                  W.update(m, SHA::DIGESTSIZE);
            t2=clock();
            cout<<"Update\t"<<j<<"\tempty wintertnitz signatures=\t"
             <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
      }
      //Makes a Winternitz siganture
      for(i=inc; i<=max; i+=inc)
      {
            t1=clock();
            for(j=0; j<i; j++)
                  Winternitz W(m, SHA::DIGESTSIZE);
            t2=clock();
            cout<<"Make\t"<<j
             <<"\twintertnitz signatures=\t"
             <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
      }
```

```
  //Makes a Winternitz siganture and does the self
//verification test
  for(i=inc; i<=max; i+=inc)
  {
       t1=clock();
       for(j=0; j<i; j++)
       {
            Winternitz W(m, SHA::DIGESTSIZE);
            if(!W.verifySignature())
            {
                 cout<<endl<<"Signature failed";
                 return;
            }
       }
       t2=clock();
       cout<<"Make/self verity\t"<<j
        <<"\twintertnitz signatures=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
  }
  //Create the temporary variables need to verify a
//signature
Winternitz tempW(m, SHA::DIGESTSIZE);

 short unsigned int len=tempW.getxyLen();
 byte ** tempSign=new byte *[len];
 byte *tempSubVal=new byte [len];
 byte **tempY=new byte *[len];

 for(i=0; i<len; i++)
 {
      tempY[i]=new byte [SHA::DIGESTSIZE];
      tempSign[i]=new byte [SHA::DIGESTSIZE];
 }
 //Gets the public parts of a Winternitz signature
 tempW.getSignature(tempSign, tempY);

 //Make Winternitz test-objects, used to veruty signatures.
 for(i=inc; i<=max; i+=inc)
 {
      t1=clock();
      for(j=0; j<i; j++)
           Winternitz testW(m, SHA::DIGESTSIZE, tempY);
      t2=clock();
      cout<<"Make\t"<<j<<"\twintertnitz test objects=\t"
       <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
 }

 //Gets the public parts of a WinternitzShort signature
 for(i=inc; i<=max; i+=inc)
 {
      t1=clock();
      for(j=0; j<i; j++)
           tempW.getSignature(tempSign, tempY);
      t2=clock();
      cout<<"Get (copy)\t"<<j
       <<"\twintertnitz signatures=\t"
       <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
 }

 //Gets the public parts of a WinternitzShort signature
 for(i=inc; i<=max; i+=inc)
```

```
for(int i=0; i<signLen; i++)
{
      y[i]=new byte [SHA::DIGESTSIZE];
      sign[i]=new byte [SHA::DIGESTSIZE];
}

cout<<endl<<"Calling W1.getSignature(sign, y). Should be ok";
if(!W1.getSignature(sign, y))
{
      test=false;
      cout<<endl<<"Could not get the siganture on W1";
}
else
      cout<<endl<<"getSignature(sign, y) ok";

cout<<endl<<"Calling W1.verifySignature(sign)."
      <<endl<<"Signature on W1 should be ok";
testValue=W1.verifySignature(sign);
if(testValue==-1)
{
      test=false;
      cout<<endl<<"Signature verification on W1
  could not be completed";
}
else
      if(testValue==0)
      {
            test=false;
            cout<<endl<<"Signature on W1 failed";
      }
else
      if(testValue==1)
            cout<<endl<<"Signature on W1  ok";
cout<<endl;

cout<<endl<<"Calling W1.verifySignature(sign, y)."
      <<endl<<"Signature on W1 should be ok";
testValue=W1.verifySignature(sign, y);
if(testValue==-1)
{
      test=false;
      cout<<endl<<"Signature verification on W1
  could not be completed";
}
else
      if(testValue==0)
      {
            test=false;
            cout<<endl<<"Signature on W1 failed";
      }
else
      if(testValue==1)
            cout<<endl<<"Signature on W1  ok";
cout<<endl;

WinternitzShort W2;
cout<<endl<<"W2 is an empty test object. "
      <<endl<<"Calling W2.verifySignature(m, sign, y)."
      <<endl<<"Signature on W2 should be ok";
testValue=W2.verifySignature(m, sign, y);
if(testValue==-1)
```

```
  {
         test=false;
         cout<<endl<<"Signature verification on W2
     could not be completed";
  }
  else
         if(testValue==0)
         {
                test=false;
                cout<<endl<<"Signature on W2 failed";
         }
  else
         if(testValue==1)
                cout<<endl<<"Signature on W2  ok";
  cout<<endl;

  cout<<endl<<"Changing a number in y to
produce a failed signature";
  if(y[0][0]>0)
         y[0][0]--;
  else
         y[0][0]++;
  cout<<endl;

  cout<<endl<<"Calling W2.verifySignature(m, sign, y)."
         <<endl<<"Signature on W2 should fail";
  testValue=W2.verifySignature(m,sign, y);
  if(testValue==-1)
  {
         test=false;
         cout<<endl<<"Signature verification on
     W2 could not be completed";
  }
  else
         if(testValue==0)
                cout<<endl<<"Signature on W2 failed";
  else
         if(testValue==1)
         {
                test=false;
                cout<<endl<<"Signature on W2  ok";
         }
  cout<<endl;


  WinternitzShort W3;
  cout<<endl<<"W3 is an empty test object.";
  cout<<endl<<"Calling W3.getSignature(sign, y).
Should not be able to get it";
  if(!W3.getSignature(sign, y))
         cout<<endl<<"Could not get the siganture on W3";
  else
  {
         test=false;
         cout<<endl<<"Got the siganture on W3. Abnormal behavior";
  }

  if(test)
         cout<<endl<<endl<<"All tests ran as expected";
  else
         cout<<endl<<endl<<"One or more tests did
```

```
           not go as expected";
        cout<<endl;
        return;
}

//Times how long it takes to do different actions
//on a WinternitzShort signature
void winternitzShortTiming(int max, int inc)
{
        cout<<endl<<"WinternitzShort timing."<<endl;
        clock_t t1, t2;

        t1=clock();
        //A sub element has a values less then n.
        const short unsigned int n = pow(2,elementLen);
        const short unsigned int messLen=SHA::DIGESTSIZE;
        byte mess[messLen];//The message whos digest will be signed

        //Generating a random message
        AutoSeededRandomPool rng;
        rng.GenerateBlock(mess, SHA::DIGESTSIZE);

        //Creating the digest of the message
        byte m [SHA::DIGESTSIZE];
        SHA().CalculateDigest(m, mess, messLen);
        t2 =clock();

        int i,j;//loop counters
        //Makes an empty WinternitzShort siganture
        for(i=inc; i<=max; i+=inc)
        {
                t1=clock();
                for(j=0; j<i; j++)
                        WinternitzShort W;
                t2=clock();
                cout<<"Make\t"<<j<<"\tempty wintertnitz signatures=\t"
                 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }


        //Updates an empty WinternitzShort siganture
        //Can normally only be done once on an empty object
        for(i=inc; i<=max; i+=inc)
        {
                WinternitzShort W;
                t1=clock();
                for(j=0; j<i; j++)
                        W.update(m, SHA::DIGESTSIZE);
                t2=clock();
                cout<<"Update\t"<<j
                 <<"\tempty wintertnitz signatures=\t"
                 <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }

        //Makes a WinternitzShort siganture
        for(i=inc; i<=max; i+=inc)
        {
                t1=clock();
                for(j=0; j<i; j++)
                        WinternitzShort W(m, SHA::DIGESTSIZE);
                t2=clock();
```

```
        cout<<"Make\t"<<j<<"\twintertnitz signature objects=\t"
         <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
}


//Create the temporary variables needed to verify a siganture
WinternitzShort tempW(m, SHA::DIGESTSIZE);

short unsigned int len=tempW.getxyLen();
byte ** tempSign=new byte *[len];
byte *tempSubVal=new byte [len];
byte **tempY=new byte *[len];
//Allocate memory
for(i=0; i<len; i++)
{
      tempY[i]=new byte [SHA::DIGESTSIZE];
      tempSign[i]=new byte [SHA::DIGESTSIZE];
}


//Gets the public parts of a WinternitzShort signature
tempW.getSignature(tempSign, tempY);

//A signature testing object
WinternitzShort testW;
short testValue=testW.verifySignature(m, tempSign, tempY);
if(testValue==-1)
{
      cout<<endl<<"Test signature could not be completed.
   Abnormal abort";
      return;
}

else if(testValue==0)
{
      cout<<endl<<"Test signature failed. Abnormal abort";
      return;
}

//Verify the WinternitzShort siganture (tempSign, tempY)
//on the message m
for(i=inc; i<=max; i+=inc)
{
      t1=clock();
      for(j=0; j<i; j++)
            testW.verifySignature(m, tempSign, tempY);
      t2=clock();
      cout<<"Verity\t"<<j
       <<"\twintertnitz signatures=\t"
       <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
}

//Gets the public parts of a WinternitzShort signature.
//This involves producing the signature
for(i=inc; i<=max; i+=inc)
{
      t1=clock();
      for(j=0; j<i; j++)
            tempW.getSignature(tempSign, tempY);
      t2=clock();
      cout<<"Get (produce)\t"<<j
```

```
                <<"\twintertnitz signatures=\t"
                <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }

        //Gets the public y of a WinternitzShort signature.
      //This involves producing the y martrix
        for(i=inc; i<=max; i+=inc)
        {
               t1=clock();
               for(j=0; j<i; j++)
                      tempW.getY(tempY);
               t2=clock();
               cout<<"Get (produce)\t"<<j
                <<"\ty matrices=\t"
                <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;
        }
}


void makeKeys()
{
        clock_t t1, t2;
        t1=clock();
        unsigned int keyLength=1024;
        const char *privRSAFilename="hexrsapriv.txt";
        const char *pubRSAFilename="hexrsapub.txt";
        const char *privDSAFilename="hexdsapriv.txt";
        const char *pubDSAFilename="hexdsapub.txt";
        const char *seed="456erty68ur";
        t2=clock();
        cout<<endl<<"Make keys timing"<<endl;
        //Make RSA keys
        t1=clock();
        RandomPool randPool;
        randPool.Put((byte *)seed, strlen(seed));

        RSAES_OAEP_SHA_Decryptor priv(randPool, keyLength);
        HexEncoder privFile(new FileSink(privRSAFilename));
        priv.DEREncode(privFile);
        privFile.MessageEnd();

        RSAES_OAEP_SHA_Encryptor pub(priv);
        HexEncoder pubFile(new FileSink(pubRSAFilename));
        pub.DEREncode(pubFile);
        pubFile.MessageEnd();
        t2=clock();
        cout<<"Make RSA (1024) key pair=\t"<<(float)(t2-
t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;

        //Make DSA keys

        t1=clock();
        randPool.Put((byte *)seed, strlen(seed));

        DSAPrivateKey dsaPrivate(randPool,keyLength);
        HexEncoder dsaPrivFile(new FileSink(privDSAFilename));
        dsaPrivate.DEREncode(dsaPrivFile);
        dsaPrivFile.MessageEnd();

        GDSAVerifier<SHA> dsaPublic(dsaPrivate);
        HexEncoder dsaPubFile(new FileSink(pubDSAFilename));
```

```
        dsaPublic.DEREncode(dsaPubFile);
        t2=clock();
        cout<<"Make DSA (1024) key pair=\t"<<(float)(t2-
t1)/CLOCKS_PER_SEC<<"\tseconds"<<endl;

}

void signingTest(int max, int inc)
{

        clock_t t1 =clock();
        const char *privRSAFilename="hexrsapriv.txt";
        const char *pubRSAFilename="hexrsapub.txt";
        const char *privDSAFilename="hexdsapriv.txt";
        const char *pubDSAFilename="hexdsapub.txt";
        int i=0;
        const char *seed="375rth5tdy";
        long longseed;
        const int messLen=12;
        byte mess[messLen]="Hello world";
        byte digest[SHA::DIGESTSIZE];

        AutoSeededRandomPool rng;
        longseed=rng.GetLong();
        RandomPool randPool;
        //(byte*)&c;
        //randPool.Put((byte *)seed, strlen(seed));
        randPool.Put((byte*)&longseed, strlen(seed));
        byte * randomMssg=new byte[SHA::DIGESTSIZE];
        randPool.GenerateBlock(randomMssg, SHA::DIGESTSIZE);



        SHA().CalculateDigest(digest, mess, messLen);

        GDSASigner<SHA> dsaSigner(FileSource(privDSAFilename,
                                  true, new HexDecoder));
        GDSADigestSigner dsaDigestSigner(FileSource(privDSAFilename,
                                            true, new HexDecoder));
        GDSADigestVerifier dsaDigestVerifier(FileSource(pubDSAFilename,
                                  true, new HexDecoder));

        RSASSA_PKCS1v15_SHA_Signer rsaDigestSigner(FileSource
                        (privRSAFilename, true, new HexDecoder));
        RSASSA_PKCS1v15_SHA_Verifier rsaDigestVerifier(FileSource
                        (pubRSAFilename, true, new HexDecoder));


        int signDsaDigestLen=dsaSigner.SignatureLength();
        byte * signatureDsaDigest=new byte[signDsaDigestLen];
        int signRsaDigestLen=rsaDigestSigner.SignatureLength();
        byte * signatureRsaDigest=new byte[signRsaDigestLen];


        int   rounds=max;
        clock_t t2 =clock();


        t1=clock();
        for(i=0; i<rounds; i++)
             dsaDigestSigner.SignDigest(rng, digest,
```

```
                     SHA::DIGESTSIZE, signatureDsaDigest);
      t2=clock();
      cout<<"dsaDigestSigner time=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\t"<<rounds<<endl;

      t1=clock();
      for(i=0; i<rounds; i++)
            dsaDigestVerifier.VerifyDigest(digest,
                     SHA::DIGESTSIZE, signatureDsaDigest);
      t2=clock();
      cout<<"dsaDigestVerifier time=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\t"<<rounds<<endl;

      t1=clock();
      for(i=0; i<rounds; i++)
            rsaDigestSigner.SignDigest(rng, digest,
                     SHA::DIGESTSIZE, signatureRsaDigest);
      t2=clock();
      cout<<"rsaDigestSigner time=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\t"<<rounds<<endl;

      t1=clock();
      for(i=0; i<rounds; i++)
            rsaDigestVerifier.VerifyDigest(digest,
                     SHA::DIGESTSIZE, signatureRsaDigest);
      t2=clock();
      cout<<"rsaDigestVerifier time=\t"
        <<(float)(t2-t1)/CLOCKS_PER_SEC<<"\t"<<rounds<<endl;

}
```