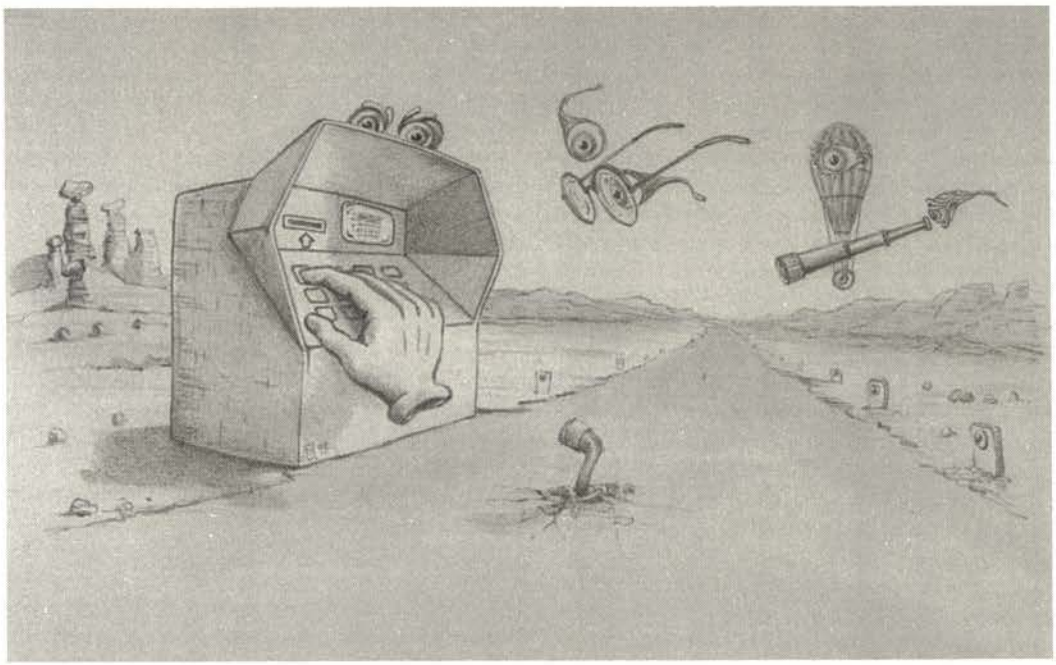
PRACTICAL PRIVACY



Jurjen N.E. Bos

Practical Privacy

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J. H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op dinsdag 24 maart 1992 om 16.00 uur

door

Jurjen Norbert Eelco Bos

geboren te Leiden.

Dit proefschrift is goedgekeurd
door de promotoren
prof. dr. ir. H. C. A. van Tilborg
en
prof. dr. J. H. van Lint,
en copromotor
dr. D. Chaum.

ISBN 90 6196 405 9

Table of Contents

1

Introduction

Cryptography

Cryptography is the branch of mathematics that deals with the protection of information. We distinguish *classical* and *modern* cryptography. We start explaining the two terms in more detail.

Classical cryptography

Until about twenty years ago, cryptography was the art of making and breaking codes. The codes are used to transfer messages over an unsafe channel. The channel should be protected against intruders who read, insert, delete or modify messages (see Figure 1). Transmission of a message is done using an *encryption function* **E**, that converts the message or *plain text* using the *key* into a *cipher text*. The receiver does the reverse of this operation, using the *decryption function* **D** and key to recover the plain text from the cipher text. The key is distributed in advance over a safe channel, for example by courier.

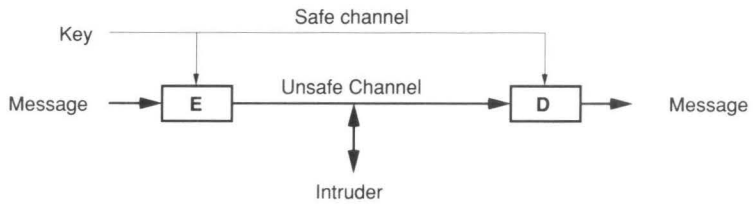


Figure 1: Classical cryptography.

Normally, the encryption and decryption function, but not the key, are considered known to the enemy, so the protection of the information depends on the key only. If the enemy knows the key, the whole system is useless until a new key is distributed.

Systems like this are still widely in use for secret communication. The American National Security Agency (NSA) proposed a public encryption function, called Data

Encryption Standard (DES), that is used for many applications nowadays.

Modern cryptography

This thesis covers aspects of modern cryptography. Modern cryptography deals with communication between many different parties, without using a secret key for every pair of parties. In 1976, Whitfield Diffie and Martin Hellman published an invited paper in the "IEEE Transactions on Information Theory" titled "New Directions in Cryptography" [DH76]*. This paper can be considered as the beginning of modern cryptography.

The authors explain the notion of a "trapdoor one-way function". To explain what this is, we first introduce the *one-way function*. A one-way function is a function that is (relatively) easy to compute, but of which the inverse function is much harder to compute. Of course, all input values can be tried; but there are so many input values that this is impractical.

A nice example of the use of a one-way function is "flipping a coin over the telephone" [Blu82], shown in Figure 2. Two parties, Alice and Bob, want to flip a fair coin during a phone conversation. Of course, neither Alice nor Bob wants to believe the other's claims just like that. To solve this, they use two coin flips and a one-way function f (in this case, a function on two arguments). First, Alice flips a coin, computes the one-way function of the coin value and some random bits† and tells the result to Bob. Then, Bob flips another coin, and tells the value to Alice. In the last step, Alice tells her coin value to Bob, and the result of the protocol is whether the two coin flips are equal or not. This behaves like a fair coin if at least one of the two uses a fair coin.

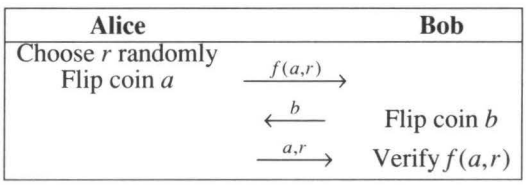


Figure 2: Coin flipping over the telephone.

A *trapdoor one-way function* (see Figure 3) is like a one-way function for everybody except the owner of a certain piece of secret information. This extra information, called again the *key*, allows to compute the inverse of the function quickly.

An interesting application of trapdoor one-way functions is *public-key cryptography*. This is a communication channel, as in classical cryptography, where both ends use different keys instead of the same key. A trapdoor one-way function is used as encryption function, and the function is assumed to be public. The description of the function is called the *public key*. The corresponding *private key* is the secret

* Literature references are given at the end of each chapter.

† The extra random information r prevents Bob from trying out values to see what Alice's value was.

information that allows to invert the function. This system allows sending of secret messages by all parties, while they cannot decrypt each other's messages.

The trapdoor one-way function is the basis of modern cryptography. Almost all modern cryptographic protocols use such functions in one way or another. At the time the notion was introduced, no functions were available that had the needed properties. A few years later Rivest, Shamir and Adleman [RSA78] introduced the first trapdoor one-way function that looked useful for practical use. Nowadays, this is the most common trapdoor one-way function. The RSA function is explained later on.

Figure 3: Trapdoor one-way function.

Protocols

A cryptographic system that solves a certain problem is called a *protocol.* For example, the coin flipping example above is a protocol. Protocols with more than two parties are called *multiparty protocols.* Using modern cryptography, protocols exist for many different problems. Actually, it is possible to do "anything" with modern cryptography in the sense that if a computation can be performed by a computer trusted by all parties, it can also be done using cryptography and no trusted computer [BCC88, CDG88]. Although this result gives an explicit protocol for a problem, this protocol is so inefficient that it cannot be used in practice.

Many practical protocols are developed nowadays that address all kinds of problems: electronic cash, elections, digital signatures, and so on.

Complexity of a Computation

Until now, we said a computation was "easy" or "hard" to perform (or, equivalently, a problem is easy or hard to solve). Of course, to make accurate statements about the security of a system, the notion must be stated more precisely. To show that a computation is "easy", one can show an algorithm that performs it using a certain number of steps, showing explicitly how much work it is. On the other hand, it is not possible to show that a problem is "hard" by showing an algorithm, since there might always be algorithms that do the computation in less steps. Unfortunately, for most "hard" problems it is not known how much time the most efficient algorithm takes.

To show that a problem is hard to solve, we use a *reduction* from another problem. To show that problem A is hard, we show that a solution to problem A can be used to solve a problem B, where problem B is a well-known problem for which no efficient solution has been found yet. If we assume that B is hard, we conclude that A must be hard as well. This is called *reduction from B to A.* Note that a problem is reduced to a

"harder" problem.

If a problem is so hard that finding a solution for given values is very expensive and time consuming, we call it *unfeasible*. How much computation exactly is called unfeasible, depends on the application. For example, for an encryption, the cost of finding the key must be compared to the cost of stealing or buying it.

Table 1 shows an overview of hard problems that are used in cryptography. There are algorithms to solve all these problems, but they are unfeasible for the parameter values used in practice:

- The largest number that was factored nowadays using an algorithm that does not use the special form of the number has 106 digits [LM89]; this computation needed the spare time of 400 computers over the world during a few months.
- The largest discrete log computation performed had a modulus p of 58 digits [McC89, McC90, MO91].
- It is an open problem whether RSA can be solved more efficiently than by factoring the modulus.
- Finding a DES key given both plain text and cipher text (a *known-plain text attack*) of DES is considered possible using a specially built computer. This makes breaking DES not unfeasible for some applications.

Assumption	Given	Find
Factoring	$n = p \cdot q$	p,q
Discrete Log	$a^b \pmod p, a, p$	b
RSA	$a^b \pmod n, b, n$	a
Breaking DES	$\mathrm{DES}(k,m), m$	k

Table 1: Cryptographic assumptions.

There is one problem of which it is easy to compute how hard it is: guessing a random value. Cryptosystems based on this have provable security, and those systems are called *unconditionally secure*. For these systems, no information about the contents of the message can be obtained without knowing the key. These systems depend on the existence of a perfect random number generator. If a simulated random generator is used, the security of the system depends on the qualities of the generator.

An example of an unconditionally secure system is the "Vernam Cipher", also known as the "one-time pad". This is a classical cryptosystem; a message is encrypted by shifting each letter a random amount in the alphabet. Every letter is shifted separately by an amount specified by the key. The key consists of a list of random shifts (0 to 25 steps) and is used only once. Note that the key has the same length as the cipher text; this is always true for unconditionally secure ciphers. The cipher text is a random string of letters, for which every plain text is equally probable. This makes that an eavesdropper gains no information about the messages by intercepting the cipher text, so the cipher text contains "no information" in information-theoretic sense.

Basic Protocols

Three protocols that occur elsewhere in the thesis are considered standard protocols. They are explained here separately.

Blobs

A *blob* [BCC88], also known as *commitment*, is a one-way function with two arguments. The arguments are known as the *contents* and the *key*. The one-way function used in the "coin flipping by telephone" example is used as a blob. The purpose of a blob is to prove that a value was chosen before a certain time, and that it didn't change since.

A blob between the parties Alice and Bob is used in three steps:

- First, Alice chooses a key, makes the blob, and sends it to Bob.
- Then, Alice and Bob perform a protocol, where Alice may use the secret blob contents, while Bob doesn't know it.
- Then, Alice *opens* the blob by sending contents and key to Bob, proving that she didn't change the contents during the protocol.

There are many different implementations of blobs, some offering extra properties. An overview of implementations is given in [BCC88]. The simplest implementation of a blob is a locked box. Alice puts the contents in the box, locks it, and sends it to Bob. Alice opens the blob by sending the key of the box to Bob, so that he can open the box to check the contents.

Digital signatures

A *digital signature* is a value that can only be computed by one party, while it can be verified by everybody. Like handwritten signatures, they can be used to sign messages. For example, a trap-door one-way function value of a message can be used as a signature.

The most well-known digital signature system is the RSA system [RSA78]. RSA is a public key system. The public key consists of two numbers n and e, and a message m is encrypted as

$$x = m^e \pmod{n}.$$

The private key consists of the number d so that $e \cdot d = 1 \pmod{\lambda(n)}$. Here, λ is the *Carmichael function*[*] that is the highest order of an element in the multiplicative group modulo n. The number n must be composite; it is assumed that d cannot be computed from n and e in this case. The decryption is performed by

$$x^d \equiv m^{ed} \equiv m^{1+k\lambda(n)} \equiv m \cdot 1^k \equiv m \pmod{n};$$

where k is a positive integer; here we use that $m^{\lambda(n)} \equiv 1 \pmod{n}$ for all m.

The two important properties that make RSA so useful are that encryption and decryption are simple computational operations, and that encryption and decryption

[*] Although the Euler function φ is most often used in this context, the Carmichael function is more general. In fact, $\lambda(n)$ is a divisor of $\varphi(n)$.

commute with each other.

It is interesting to note that RSA can be reduced to factoring: if the factorization of n is known, it is possible to extract roots modulo n. Whether the converse is true, is not known; we know that if one can extract a few square roots modulo n (that is, $b = 2$ in Table 1), it is possible to factor. For other values of b, no such algorithm is known.

Zero-knowledge proofs

A *zero-knowledge proof* (see [QGB89] for a witty and clear explanation) is a protocol in which Peggy (the *prover*) convinces Vic (the *verifier*) that something is true without revealing more information than that fact. An example of a zero-knowledge proof is given in Figure 4. Here, Peggy proves to Vic that she knows x, the discrete logarithm of h with respect to g:

$$h \equiv g^x \pmod{p},$$

where g, h, and p are public numbers, and p is a prime. As the secret x cannot be computed from g and h under the discrete log assumption, it will not be revealed.

Peggy		**Vic**
Choose $r \in \{0,\ldots,p-2\}$	$\xrightarrow{m_1 = h^r}$	
	$\xleftarrow{b}$	Choose $b \in \{0,1\}$
If $b = 0$:	$\xrightarrow{m_2 = r}$	Verify $m_1 = h^{m_2}$
If $b = 1$:	$\xrightarrow{m_2 = x \cdot r \pmod{p-1}}$	Verify $m_1 = g^{m_2}$

Figure 4: A zero-knowledge proof.

In the protocol, Peggy first sends the number h^r to Vic, where r is a random number. In Vic's point of view, this is a random number from the range $\{1,\ldots,p-1\}$. Vic then chooses a *challenge bit* b and sends this to Peggy. Finally, Peggy sends r or $x \cdot r \pmod{p-1}$ depending on the challenge bit.

To prove that a protocol is a zero-knowledge proof, three things must be proved: completeness, soundness and zero-knowledge.

A protocol is *complete* if Vic's verification succeeds if Peggy is not cheating. In the example, this can be verified using simple arithmetic.

A protocol is *sound* if Vic's verification fails with high probability if Peggy is lying. The protocol can be repeated to increase this probability. In the example, if Peggy does not know x, she cannot compute a message m_1 for which she can open both challenges. To see this, assume she can compute an m_1 so that she knows a value for m_2 for both values of b. If we call the m_2-values α and β, we get:

$$m_1 = h^\alpha = g^\beta$$

so Peggy knows that

$$h = g^{\beta/\alpha \pmod{p-1}},$$

and this means that Peggy knew x from the beginning*, since there is only one x with

* The case that division by α modulo $p-1$ is not possible is ignored here for simplicity. This can be prevented by choosing p of a special form.

the property $h = g^x$. Thus, we know that Peggy cannot compute α and β.

A protocol is *zero-knowledge* if Vic does not obtain any information from the protocol that he cannot compute himself. Normally, we prove this by showing that Vic can compute all messages involved in the protocol, with the same probability distribution, without help from Peggy. This is called *simulation* of the protocol. From a simulation we can conclude that Vic will obtain no information running the protocol with Peggy.

Privacy and Efficiency

This dissertation consists of five chapters that contain different research topics in cryptology. The connection between the chapters is that they all involve the practical implementation of privacy related cryptographic protocols. The two aspects "privacy" and "efficiency" play a role throughout the thesis.

Privacy

Nowadays, most organizations have large databases containing private information of their clients. The clients have no access to this information, nor can they control what it is used for. Linking of this information gives an accurate idea of what individuals are doing. This is not a new idea; it already happens. Companies sell databases to each other for consumer statistics, credit-worthiness and other information.

Furthermore, organizations suffer from abuse by individuals. They react with more aggressive security measures, like identity cards and television cameras. This way, people lose more and more of their privacy. Another problem for the organizations is that they cannot prove to individuals that they protect their privacy even if they do: the clients have to trust them.

David Chaum addresses these problems in [Cha85], and gives solutions based on modern cryptography. Much more work on the subject has been done since. This thesis elaborates on several of the ideas, bringing the protection of privacy closer to reality.

Efficiency

The main argument against modern cryptography has always been that implementing public key systems like RSA was "too expensive". While present day technology makes the use of these systems more and more possible, the need for cheap and thus simple systems remains. This thesis includes a chapter that increases the speed of these computations, and shows methods to efficiently use public key cryptography without the need for high-performance devices.

Overview

The five chapters with research topics address privacy and efficiency in several ways. Here we discuss the contents of the five chapters briefly.

Disruption and synchronization in untraceable sending

In protocols where privacy is an issue, like voting, it is convenient to be able to send messages while the sender remains anonymous. The *DC protocol* solves this problem. A problem with the DC protocol is that disrupters can delay or block transmissions of other users, while they are protected by the anonymity of the scheme. There are solutions to this problem in the literature, but they cost a lot of transmission. Two new solutions are shown with much less transmission cost than the solutions in the literature.

An efficient voting scheme

A voting scheme much faster than existing in the literature is introduced. This scheme is based on the DC protocol. All of the transmission is done simultaneously by all participants, so that only a small number of transmission rounds is needed to perform the election. The privacy of the users is unconditionally protected, so that the value of the votes cannot be determined, even a long time after the election.

Addition chain heuristics

The RSA scheme is criticized because, though computationally simple, encryption and decryption are relatively expensive. Methods are proposed to improve the speed of the calculations. The calculation consists of many multiplications. Most research is spent on improving the speed of the multiplications involved using advanced calculation and special hardware. This chapter shows a method to reduce the number of multiplications, so that the other improvements still apply.

Provably unforgeable signatures

Digital signatures are one of the more useful applications of cryptography. For some applications, a form of provable security is needed. The scheme that has security in the strongest sense (signatures cannot be obtained for a given message, even if other signatures of the forger's choice are available), is rather inefficient. Here, a much more efficient scheme is shown, that allows for many applications, including the fast signing of short messages.

Verification of RSA computations on a small computer

Applications using smart cards are restricted because of the limited computation power of a smart card. It is possible to make a smart card check the computation of a larger computer using modulo reductions. A protocol is shown (of which a simpler version already exists in the literature), and a security analysis is given. The protocol is used in a commercial electronic cash system.

Notation

In the formulae in this thesis, the following notations are used:

blob(x,y)	blob of y with key x (see Chapter 3).
ceiling(x), $\lceil x \rceil$	the smallest integer $\geq x$.
DC	Dining Cryptographers scheme (see Chapter 2).
DES	Data Encryption Standard private key scheme.
exp x	shorthand for e^x.
floor(x), $\lfloor x \rfloor$	the largest integer $\leq x$.
gcd S	greatest common divisor of the elements of the set S.
GF(p), GF(p^k)	Galois field of order p or p^k.
$l(n)$	length of shortest addition chain of n (see Chapter 4).
lcm S	least common multiple of the elements of the set S.
log(x)	natural logarithm of x.
$\log_2(x)$	base-2 logarithm of x.
RSA	Rivest-Shamir-Adleman public key scheme.
$\xrightarrow{x}$	transmission of the value x from one place to another.
$x \bmod n$	the rest of x after division by n.
$x \operatorname{div} n$	Shorthand for $\lfloor \frac{x}{n} \rfloor$.
$x \equiv y \pmod{n}$	$x \bmod n = y \bmod n$, or equivalently, $n \mid (x-y)$.
$E \pmod{n}$	value of expression E computed modulo n. Symbolic notation meaning "interpret all computations in E modulo n".
$x \stackrel{?}{\equiv} y \pmod{n}$	check if $x \equiv y \pmod{n}$.
$\sqrt[x]{y} \pmod{n}$, $y^{1/x}$	RSA decryption: shorthand for $y^d \pmod{n}$, where d is defined by $d \cdot x \equiv 1 \pmod{\lambda(n)}$. It can only be computed if one knows the factorization of n (under the RSA assumption).
$a \mid b$	a divides b, or, equivalently, b is a multiple of a.
$p^k \| n$	p^k is the highest power of p that divides n.
$\varphi(n)$	Euler's φ function: number of $x \in \{1,\ldots,n\}$ satisfying $(x,n) = 1$.
$\lambda(n)$	Carmichael function: highest order of elements in multiplication group modulo n.
$\sum_{i \in S} f(i), \prod_{i \in S} f(i)$	Sum, respectively product, over all values in a set.
$n!$	Factorial function: equal to $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.
$\binom{x}{y}$	Binomial coefficient: the number of possibilities to choose y elements from a set of x elements. Equal to $\frac{x!}{y!(x-y)!}$.
$\{x \in S \mid E\}$	the set of elements x of the set S that fulfill the conditional expression E.
$\#X$	Number of elements of list or vector X.

References

[BCC88] **G. Brassard**, **D. Chaum**, and **C. Crépeau**: *Minimum Disclosure Proofs of Knowledge*, Journal of Computer and System Sciences **37** (No. 2, October 1988), pp. 156-189.

[Blu82] **M. Blum**: *Coin Flipping by Telephone*, Proc. IEEE Compcon (1982), pp. 133-137.

[CDG88] **D. Chaum**, **I. B. Damgård**, and **J. van de Graaf**, *Multiparty computation ensuring privacy of each party's input and correctness of the result*, Advances in Cryptology: Proc. Crypto '87 (Santa Barbara, CA, August 1987), pp. 87-119.

[Cha85] **D. Chaum**: *Security without Identification: Transaction Systems to make Big Brother Obsolete*, Comm. of the ACM **28** (number 10, October 1985), pp. 1030-1044.

[DH76] **W. Diffie** and **M. E. Hellman**: *New Directions in Cryptography*, IEEE Trans. Information Theory **IT-22** (No. 6, November 1976), pp. 644-654.

[LM89] **A. K. Lenstra** and **M. S. Manasse**: *Factoring by Electronic Mail*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, April 1989), pp. 355-371.

[McC89] **K. S. McCurley**: *The Discrete Logarithm Problem*, Technical Report RJ-6877, IBM Research division, Almaden Research Center, San Jose, CA.

[McC90] **K. S. McCurley**: *The Discrete Logarithm Problem*, Proc. Symposium of Applied Mathematics, American Mathematical Society (1990), to appear.

[MO91] **B. A. LaMacchia** and **A. M. Odlyzko**: *Computation of Discrete Logarithms in Prime Fields*, Designs, Codes and Cryptography **1** (Number 1, may 1991), pp. 47-62.

[QGB89] **J. J. Quisquater**, **L. Guillou**, and **T. Berson**: *How to explain Zero-knowledge to your children*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 628-631.

[RSA78] **R. L. Rivest**, **A. Shamir**, and **M. Adleman**: *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Comm. ACM **21** (No 2, February 1978), pp. 120-126.

2

Disruption and Synchronization in Untraceable Sending

Introduction

One of the aspects of privacy is secrecy of communication. With this, we do not only mean hiding the contents of the messages (which is easily done by encryption), but also hiding the identity of the sender and receiver, and the number of messages sent and received by each individual.

Hiding the receiver's identity is easily achieved by broadcasting the message. If the message is targeted at one receiver, the message can be encrypted so that only the proper receiver can decrypt it. Hiding the sender's identity is more difficult. In current transmission systems, the location of the sender of a message is always easily found. The transmission of a message that hides the sender's identity is called *untraceable sending*.

David Chaum [Cha88] introduced an untraceable sending system, the *Dining Cryptographers* (DC) system, that hides the sender of a message unconditionally, and only reveals an upper bound of the total number of messages sent. The name "dining cryptographers" comes from one of the examples used in his article.

The main disadvantage of the system is that messages may not be sent simultaneously. If two messages are sent simultaneously, their sum (the messages being interpreted as elements of a group) becomes public instead, so that the information is lost. This brings the following problems:

- Some kind of *synchronization* is necessary to prevent collisions.
- Every sender can *disrupt* the total by sending random data. A special protocol is needed to catch such a disrupter without reducing the privacy of the other users.

In the rest of this chapter, we first discuss the implementation of a DC network in practice. Then, we discuss the disruption and synchronization problems in detail. Finally, two new solutions to these problems are shown and compared to the other solutions that exist in the literature.

The Dining Cryptographers (DC) System

The Dining Cryptographers system is best introduced by an example. Assume that two people want to send a message anonymously to a third party. They want the receiver to be unable to find out which one of them sends this message. To do this, they first make a random binary number, called the *key*. If one of them wants to send a message, he outputs the bit-wise exclusive-or of that message and the key*. At the same moment, the other of the two outputs the key itself. The receiver, or any other interested party, can compute the exclusive-or of the two outputs to retrieve the message, but the two random outputs cannot be distinguished to see which one of them contained the message.

This idea can be extended to any number of senders. Let every pair of senders share a key. The sender of a message outputs the exclusive-or of all keys he shares and the message to be sent, while all other users output the exclusive-or of only their shared keys. The exclusive-or of all these outputs gives the message, because every key occurs twice in the total. Again, all outputs are random numbers, so all users are equally probable to be the sender.

In the applications we are interested in, the senders and receivers are the same group of participants.

The example clearly shows two important properties of the DC system:

- Protection of the sender is unconditional; there is no way to find the sender of a message, even with unlimited computing power.
- All participants transmit once for every message, even if they do not send a message themselves.

On top of the DC protocol, a *transmission rule* is needed that defines when participants are allowed to send. Transmission rules must synchronize messages and detect disrupters so that the network is used efficiently.

Literature

The problem of untraceable sending is first addressed in [Cha81]. This system uses a network of trusted mail-relaying machines, the *mixes*. A mix receives RSA-encrypted data, decrypts it, and forwards it in batches to the appropriate addresses. The privacy protection is not unconditional, and the system in its simplest form has been broken [PP89]. As far as we know, the mix system is the only alternative for the DC system.

[Cha88] introduces the DC system. The problems of synchronization and disruption are addressed, but the proposed transmission rule is rather inefficient.

[BB89] gives a more efficient solution to the synchronization problem, it was the first publication on a collision resolve system. [Wai89] discusses the verification problem (see below) and describes a synchronization protocol by Andreas Pfitzmann. The ideas of that paper are further elaborated in [WP89]. The latter paper also elaborates the placement of traps (see later on).

A discussion of practical considerations for untraceable sending occurs in [PW87].

* This is the same hiding of information as in the one-time pad.

Implementation

[Cha88] states that the exclusive-or operation of the example may be extended to other fields, but that it "seems to offer little practical advantage". It turns out however, as Birgit Pfitzmann discovered [Pfi85], that the exclusive-or operation may even be replaced by the operation of any commutative group. Such an extension can be used in many ways; for example see the voting scheme of Chapter 3. The group operations that will be proposed in this chapter and in Chapter 3 are mainly addition or multiplication modulo an integer, computations that are easy to perform in hardware. We write the group operation that is used as a sum.

In the general case of a commutative group, the two owners of a key stream must use inverse key values. If participants i and j use keys k_{ij} for their keys, we write $k_{ij} = -k_{ji}$ to show that they use opposite values of the keys. The message sent by user i is m_i; we write $m_i = 0$ if user i does not send a message. Using this notation, user i outputs the value

$$m_i + \sum_{j \in U} k_{ij} .$$

U is the set of all participants (the "universe"). It is easy to verify that the total of the transmitted outputs is the total of the messages:

$$\sum_{i \in U} (m_i + \sum_{j \in U} k_{ij}) = \sum_{i \in U} m_i + \sum_{i,j \in U} k_{ij} = \sum_{i \in U} m_i .$$

Key sharing

If two participants share a key stream, they must both store the keys. To guarantee unconditional privacy protection, the keys should not be generated by an algorithm, but have to be truly random bits, just as with the one-time pad. This can make the list of keys quite big; storage on an optical disk may be a practical solution.

Not all pairs of users need to share keys. Using less key pairs saves on three important points:

- computation of the user's output;
- storage of keys;
- key distribution.

The choice of which pairs of users share keys is important for privacy protection.

If not every pair of users shares a key, the untraceability of a message depends on the reliability of certain participants, which we will explain using the *key graph*. The nodes of the key graph are the participants; two nodes are connected by an edge if the participants share a key. Figure 5 gives an example of a key graph.

The set of users that share keys with a given user is called the set of *neighbours* of that user. In graph terminology, the set of neighbours of a node is the set of nodes directly connected to that node. The participant marked ★ in Figure 5 has 4 neighbours.

The properties of the key graph model the privacy protection of the participants. Take two trustworthy participants. If they are connected by an edge, it is impossible to distinguish them as sender of a message. Also, if there is a path in the graph connecting

them, it is still impossible to distinguish them. If we now consider the entire key graph, we see that if the key graph is connected, the privacy of all members is ensured. If the graph is disconnected, messages can be traced to one of the parts of the graph. Within the connected parts of the graph, all members are untraceable with respect to each other.

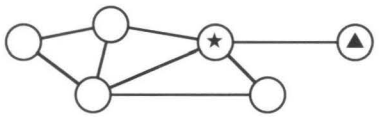


Figure 5: A key graph.

The structure of the key graph determines untraceability of users with respect to other users. A simple example is shown in Figure 5: the user marked ★ knows the only key used by ▲, so that all messages of ▲ are clearly visible to him. User ▲ has no untraceability with respect to ★, but unconditional untraceability towards all other users. This can be generalized to knowledge of arbitrary sets of keys. A user who knows a set of shared keys can subtract those keys from the corresponding outputs, and remove the corresponding edges from the key graph. This way, he can construct a graph of the keys unknown to him. The structure of this reduced graph determines the anonymity of all other users for him.

Users sharing their keys, called *colluders*, may know a large number of keys. This can remove many edges from the key graph. If the colluders know enough keys to partition the graph, they can see which part of the graph messages come from. The number of colluders that can partition the graph depends on the structure of the graph. For example, the "polygon" (ring-shaped) graph gives untraceability from all users towards all others, but every collusion splits the graph.

Practical key graphs have a structure that depends on the needs of the users in the graph. The goal of a good key graph is to contain as few keys as possible, while having enough keys to give all users sufficient privacy protection in their opinion. If the participants are allowed to choose their neighbours, they can choose those they trust. Another consideration is the physical key distribution: it is preferable to let the participants have neighbours that are physically close to them.

Addition networks

It is obvious that an efficient implementation of the DC scheme must quickly compute the output total. This can be done with what we call an *addition network*. An addition network connects a set of participants for a DC scheme. The network takes the output from each participant, adds them (using the proper group operation) and makes the result available to every participant. Some of the protocols use a mixture of group operations, and the addition network must make sure that the outputs are combined properly.

The addition network does not need to be protected against wire tapping, because all the users' messages are encrypted (by adding the shared keys). If the signals are publicly accessible, the computations of the addition network can be verified by any

interested party.

The distribution of the total sum among the participants can be performed in computation time logarithmic in $\#U$ (the number of users). This is done by a network in the shape of a tree, where every branch adds the result of its subbranches, and sends the sum to the next higher level. The highest level distributes the results back to the leaves (see Figure 6). The time it takes to compute and distribute the total is called the *turnaround time*. Because we assume a fixed transmission rate, this time is more conveniently expressed in the number of bits that can be transmitted in that time. In practical implementations, the turnaround time will be a few thousand bits.

If the turnaround time is long compared to the length of the messages, messages that depend on previous messages are delayed considerably until the previous output is distributed. Most protocols consist of *rounds* of messages whose contents do only depend on previous rounds. To make such protocols efficient, as few rounds as possible should be used to reduce the effect of the turnaround time.

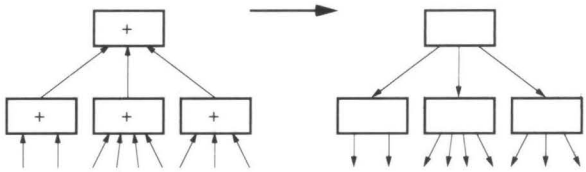


Figure 6: Distributing the sum in a hierarchical network.

Addition networks are not hard to implement in practice. Most standard networks can in principle be reprogrammed to implement an addition network. The worldwide Internet network and the local token ring networks are good examples, because they do some processing on the packets anyway.

Once an addition network is set up, it can also be used for other purposes, such as broadcasting of messages, or quick addition of traceable messages of all users. These last possibilities will be used for the construction of a voting scheme in Chapter 3.

An example

We found a very elegant addition network that can be built if the addition network only has to perform additions modulo an integer. Unfortunately, this makes it unusable for the voting protocol of Chapter 3.

The network has the form of a binary tree, where each node adds two numbers and sends the sum to the next node. The nodes consist of only a stream adder (see Figure 7). A stream adder adds two numbers with the least significant bit first. It uses only one bit of memory (for the carry). Numbers must be separated by one or more zeros, to give room for the carry. The addition network consists of a stream adder at every node, computing the unreduced sum of the output values. The modulo reduction is done only on the total. The last node computes this modulo reduction and distributes it back along the other nodes. If there are p participants, the unreduced total sum is $\log_2 p$ longer than the user outputs. Thus, after every number, $\log_2 p$ zeros must be fed into the

stream adders for the extra bits.

This system uses a very small amount of hardware per node; essentially, only a stream adder is necessary. Because only the last node has to take care about the modulus, this system is particularly efficient if the modulus changes often. The turnaround time is very low, because the nodes do not have to store the intermediate results.

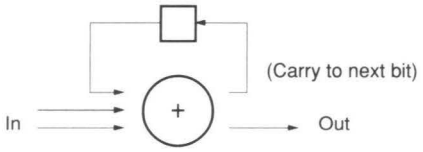


Figure 7: A stream adder.

Verification

The distribution of the output among the participants must give each participant the same value (that we call *input* for the discussion). If the participants can be given different inputs, traffic analysis is possible by observing reactions of participants. Making sure that all inputs are equal is called the *verification problem*.

The traffic analysis attack, and a solution to the verification problem, are described in [Wai89]. His solution is to make the keys dependent on the input of the previous round. If each participant receives the same input during a round (as it should be), the key pairs are distributed uniformly, so that there is normal untraceability. If two participants receive different inputs, their keys of the next round will not match, so that the total output of that round is uniformly random. Although the users can detect that two users received different inputs, there is no way to prevent it. Also, the turnaround time is not taken into consideration in this protocol. Every transmission depends on the previous total, so that the turnaround time effects every message.

A more efficient, but less secure solution to the verification problem is to distribute the total over several routes simultaneously. The nodes compare their inputs from other nodes to detect modification of the total along the way. An extra advantage of this method is that it improves the reliability of the network. The turnaround time need not be affected by this method.

Detection of Disrupters

Disruption is the transmission of data that are not allowed according to the protocol, either maliciously or because of a defect. A malicious disrupter will try to hide his identity using the untraceability of the network. A good transmission rule reduces the amount of harm disrupters can cause, without reducing the untraceability of the messages.

Opening

The only weapon in the battle against disrupters is *opening*. An output message is opened if every user publishes all keys used to compute his output for that message. This will also reveal the message m_i of every user. The public can verify whether the outputs were computed correctly, that participants sharing a key used matching keys (we assume the key graph is public), and that the transmission rule was obeyed. The anonymity of all user messages is lost, of course.

A disrupter can admit having sent the disruptive message, or try to hide it. If he wants to hide his message by lying about his keys, a conflict results between the disrupter and one of his neighbours. The only solution is to stop using this key (removing an edge from the key graph). A transmission error that is the result of wrong key sharing will also be removed. A disrupter who keeps disrupting and lying about his keys will quickly become suspect, and eventually lose all his keys.

Traps

Opening a message containing private information is impossible, because users do not want their messages to be traced. To catch a disrupter by opening a message, special messages must be made that contain no private information. A good solution proposed in [Cha88] is to send *trap messages*, messages that contain no private information. The sender of a trap message publishes a commitment of the trap in advance so that he can prove afterwards that the corresponding message can safely be opened. If a trap message gets disrupted, the corresponding message can be opened and the disrupter can be caught. [WP89] gives an explicit example of such a method.

Synchronization

The important problem in a practical implementation of the DC scheme is the coordination of the different senders. If two or more participants decide to send at the same time, the result may not be understandable for the public. (In the case of two participants, the participants in question can determine each other's message by subtracting their own, but the other users only get the sum of the messages.) The transmission rule determines when participants may send their message. This coordination problem is as old as computer networks, and there is a lot of literature on this subject (see for example [DBP79], [Tan88] for an overview). We do not go into details here, but the literature shows that a good transmission rule provides the following properties:

- every user must be able to send a certain amount of data;
- delay times must be as short as possible.

In untraceable sending, we need an extra property:

- the protocol may not reduce sender anonymity.

This last rule limits the number of protocols from the literature that can be applied, because anonymity is not an issue in these standard protocols.

From the first and second properties it is clear that a transmission rule should limit the harm caused by disrupters as much as possible. Finding disrupters is complicated by

the third property. Ideally, all the disruptions should be traced, while the rest of the messages stay untraceable.

Slot reservation

There are two kinds of transmission rules for untraceable sending. The simplest transmission rules use *slot reservation*. Transmission is divided into two rounds: the *reservation round* and the *communication round*. In the reservation round, all users send a message (possibly simultaneously), determining an ordering for the users. In the communication round, the users have to send in this order. All users send one (possibly empty) message in that order.

In slot reservation, detection of disrupters is rather simple: a disruption of the reservation round forces opening of the messages. The reservation round is designed so that only random data is involved, so that the reservations can be opened without compromising privacy. Disruptions in the communication round are caught using trap messages.

Collision detect

The other idea is to use *collision detect*. In a collision detect system, users send whenever they have something to send. The other users send zeros (output the sum of their keys only). This is a special case of the *ALOHA protocol*.

The original ALOHA protocol [DBP79] was designed for satellite oriented radio communication between islands. It is currently used on networks where all users have access to the same *bus*. This bus need not be a radio frequency; it can also be a cable (as is done on the Ethernet). The transmission rule is very simple: every user who wants to send, sends immediately. If the transmission happens to be simultaneous with that of another participant, the result will be garbled (a *collision*). In this case, both participants wait a random time* and try again. This protocol is very efficient, especially because senders do not have to wait for their turn. A disadvantage is the unpredictable behaviour if there is a lot of traffic.

In untraceable sending, this system is applicable if transmission occurs at regular intervals (this is called *slotted ALOHA* in [Tan88, section 3.2]). At each step, all users send a zero message, except for users who want to say something, who start sending a *header block*, followed by the actual message.

Since disrupters can send fake header blocks, these blocks must occasionally be opened. To protect the privacy, header blocks must not contain message information†. Disruption of header blocks can be detected by opening header blocks at random times or if the number of collisions is higher than statistically expected. Disruption of messages can be prevented as before using traps.

The efficiency of collision detect comes from not having to waste time on users who have nothing to send. The price paid for this is privacy: if a header block has to be opened, the users who intended to send at that time are revealed. The privacy loss can

* In practice, the delay is a uniform random number from an interval. If collision happens again, the delay is chosen from longer and longer intervals.

† This problem is overlooked in [Wai89].

be reduced at the cost of efficiency by letting users send random header blocks for empty messages at irregular intervals.

Collision resolve

The additive properties of the DC system allow to resolve collisions in a much more efficient way than waiting a random time. If the header blocks are chosen appropriately, the sum of the header blocks can determine an order for the group of messages that are involved in the collision. The messages can then be sent in the specified order, like the slot reservation system. This is called *collision resolve*. This notion was discovered independently by Andreas Pfitzmann [Wai89]. From the user's viewpoint, slotted ALOHA with collision resolve works as follows:

- If there is no message to send, send a 0.
- If there is a message to send, wait until the current group of messages ends.
- Start sending a header block.
- If there is a collision, determine when to send and wait for your turn in the message group.
- At the proper moment, send the message contents without header block.

All collision resolve protocols use this structure. The protocols determine the form of the header block, and the order of messages in a message group.

Sending long messages

The transmission of long messages (whether using collision detect or slot reservation) can be done in several ways.

The simplest solution is to give all messages a fixed length, and break long messages in pieces. This gives full untraceability of the messages, but it is rather inefficient.

It is more efficient to send messages in one piece. If all messages are sent in one piece, the messages must contain some information about their length. The sender of the next message must know this information to determine the moment he can start sending his message. Thus, the time between two messages is at least as large as the turnaround time. If the turnaround time is large, the protocol is again inefficient.

A more efficient solution is to encode the length of the messages in the header block or reservation round. This way, the locations of all messages are known beforehand, and the messages can be sent without delay. As opposed to the previous solutions, this solution does compromise privacy: if the header block or reservation round is opened, all the message lengths become public.

All protocols presented here can be adapted to use any of these three ways to handle long messages. For the rest of the discussion, we assume that all messages have the same length.

Comparison of Transmission Rules

We compare five transmission rules: two slot reservation protocols and three collision detect protocols. Two collision detect protocols are new, the others are known from the literature.

A slot reservation protocol and a collision detect protocol are very similar. Both protocols work in two steps: first an ordering is determined, then a list of messages is sent in that order. Slot reservation protocols use a reservation block that contains messages of every participant; collision detect protocols send a header block that contains messages of all participants who happen to send at that time.

It is always possible to use a collision resolve protocol as a slot reservation protocol: let all participants "collide" at a certain moment, and use the rules of the collision detect protocol to determine an order. Conversion in the other direction, from a slot reservation protocol to a collision detect protocol, is also possible. In fact, the two new collision detect protocols are adapted slot reservation protocols.

An advantage of slot reservation protocols over collision detect protocols is that they give more intrinsic privacy: opening the reservation round does not reveal who wants to send something, because everybody reserves something in the round. An advantage of the collision detect protocols, on the other hand, is that they allow more efficient channel usage.

The criteria we use for comparison of the transmission rules are: transmission time, sensitivity for turnaround time, and computation efficiency.

Slot reservation by Chaum

The slot reservation system as described in [Cha88] used a reservation round consisting of a number of messages of weight one (zero bits on all but one position). Every participant chooses one of the bit positions at random and sends a 1 on only this position. The total of these messages is computed bitwise modulo 2 (exclusive or). If two participants accidentally choose the same bit position, a *conflict* occurs. Such a conflict is detected if the number of ones in the reservation round is less than $\#U$. In this case, the round is repeated. Otherwise, the order of the ones in the round determines the order in which the participants may use the following rounds for their message (see Figure 8). This way, all participants know when it is their turn to send.

The transmission time is quadratic in $\#U$. This is because the length of the reservation round must be large enough to keep the probability of a collision low. (This is known as the *birthday paradox*. If there are more than 23 people in a room, the probability is more than 50% that there are two people among them with the same birthday.) If there is a collision, the reservation round has to be repeated.

The protocol is insensitive to a long turnaround time, and uses almost no computation. After the reservation round, the turnaround time does not cause any more delays. All users know when it is their turn to send, even before the total of the previous message is computed. This allows the transmission of all messages in one round, without breaks.

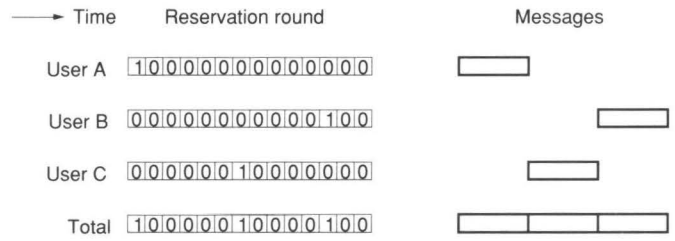


Figure 8: Slot reservation of [Cha88].

The transmission time for this system can be estimated statistically. The probability that there is a collision in the reservation round is

$$C = 1 - \frac{r!}{(r-n)!\,r^n},$$

where n is the number of participants, and r is the number of bits in the reservation round.

The turnaround time influences the optimal length of the reservation round. If the turnaround time is equivalent to the transmission of t bits, a reservation round takes $r+t$ bits. Because of collisions, a reservation is repeated $\frac{1}{1-C}$ times in average, so the expected total transmission time for the reservation rounds is $\frac{r+t}{1-C}$ bits. To compute the optimal choice of r, we differentiate with respect to r:

$$\frac{\partial}{\partial r}\frac{r+t}{1-C} = \frac{\partial}{\partial r}\left(\frac{r^n}{r+t}\cdot\frac{(r-n)!}{r!}\right) =$$

$$\frac{r^n}{r+t}\cdot\frac{(r-n)!}{r!}\cdot\left(\frac{1}{r+t}+\frac{n}{r}+\Psi(r+1-n)-\Psi(r+1)\right).$$

Using asymptotic approximation for small t, we get the optimal value for r:

$$r \approx \tfrac{1}{2}n^2 + \tfrac{1}{6}n + t - \tfrac{2}{9} - \tfrac{4}{3n}(t+\tfrac{1}{45}) + O(n^{-2}).$$

In some possible implementations of an addition network, t grows with r. If we set $t = \lambda r$, we get

$$r \approx \tfrac{1}{2}(1+\lambda)\cdot n^2 + (\tfrac{1}{6}-\tfrac{\lambda}{2})\cdot n + \frac{1}{9(1+\lambda)} - \tfrac{1}{3} - \frac{4}{135(1+\lambda)^2}\cdot\frac{1}{n} + O(n^{-2}).$$

From our computation we can see that the optimal value of r is quadratic in n. The probability of a collision is rather high (if $t = 0$, we get $r = \frac{1}{2}n^2$ and $C \approx 1 - \frac{1}{e}$), so that up to e repetitions of the round are to be expected. If $t = 0$, the expected total transmission time for the reservation round is $\frac{e}{2}n^2$ bits. (Note that the value for r of $100 \cdot n^2$ suggested in [BB89] is not only much higher than the optimum, but also significantly more than the expected total time.)

The influence of a long turnaround time on the total time for a reservation round is relatively small. For small values of t, the total time grows like $e \cdot t$, which is rather slowly. For larger values of t, the total time grows even more slowly, since the probability of collision drops.

A problem is that the participants can influence the position of their message during the next round by sending the 1 bit in the reservation round earlier or later. There is a

simple solution for this, that is not mentioned in the original article [Cha88]. We make the assigned order unpredictable with a cyclic shift over a random distance. If the original rule states that the users send in order $a_1, a_2, \ldots, a_n$, the modified rule defines the order to be $a_x, a_{x+1}, \ldots, a_n, a_1, a_2, \ldots, a_{x-1}$. The shift distance x is a random number from the interval $\{1, \ldots, n\}$. To make a random number known to all participants, we can for example compute x as a one-way function value of the reservation round. Another way is to let all participants send a random number for one round, and use the total of that round. Such optimizations can be applied to all systems mentioned here to make sure that the participants cannot predict their position in the messages.

Slot reservation by den Boer

Bert den Boer [Boe88] proposed a slot reservation protocol that almost never needs repetition of the reservation round. In it all participants send a special block containing a random number. The values of all these blocks can be computed from their sum using a rather complicated algorithm. The algorithm determines a transmission order that cannot be predicted by the participants.

The transmission time of the algorithm is rather low, because almost always only one block has to be sent. This makes the algorithm insensitive to a large turnaround time, because this delay only influences the time between the reservation round and the communication round. The computation, on the other hand, is much more involved than the previous protocol.

If the turnaround time is very high, or the computation is too much, the rounds can be skewed one or more rounds so that the reservation round determines a later message round; see Figure 9. This allows the channel to be used efficiently. This idea is also applicable for other slot reservation protocols, but not for collision detect protocols.

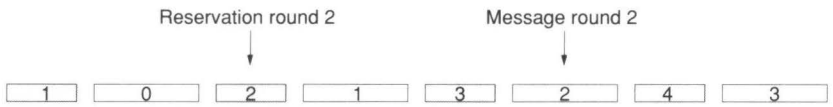


Figure 9: Dealing with long delays in slot reservation.

In this slot reservation scheme, user i sends a reservation block of the form

$$A_i, A_i^2, A_i^3, \ldots, A_i^n,$$

where the numbers A_i are nonzero random numbers in a finite field with q elements. For simplicity, we take q to be prime. The blocks are added up coordinatewise in the field, giving the sums

$$S_1, S_2, S_3, \ldots, S_n.$$

The set of A_i values can be determined from the sums alone. Later on we explain an algorithm to do this in detail. The order in which the participants may send in the message round is determined by this algorithm. The position of an A_i in the communi-

cation round is not determined by the value of A_i, so an algorithm to "shuffle" the communication round is not needed.

We now estimate the expected transmission time for the protocol. Call the number of field elements q. A field element can be encoded using $\log_2 q$ bits (ignoring roundoff). The length of the reservation phase is

$$n \log_2 q .$$

The probability of a conflict can be approximated in the same way as with the slot reservation algorithm of Chaum. The probability is about

$$C = 1 - e^{-n^2/2q} .$$

Taking account of the turnaround time, the total time of the reservation round with repetitions becomes $\frac{n \log_2 q + t}{1-C}$, which is minimal if q is the solution of the equation

$$q = \frac{\log 2}{2} \cdot n(n \log_2 q + t);$$

we choose for q a prime number close to this value, which is roughly $n^2 \log_2 n$. For example, if there are a million participants, the optimal value of q is about $1.52 \cdot 10^{13}$ and C is about 0.032.

We now explain the algorithm in detail. Every participant i chooses a random nonzero value A_i from the field. The reservation block is

$$A_i, A_i^2, A_i^3, \ldots, A_i^n .$$

The blocks are added up in the field, and we call the sums

$$S_j = \sum_{i \in U} A_i^j, \text{ where } j \in \{1, \ldots, n\}.$$

The A_i are the reciprocals of the roots of the polynomial over the field

$$P(x) = \prod_{i \in U} (1 - xA_i).$$

The coefficients of this polynomial can be computed from the sums S_j. To see this, first write out the product as

$$P(x) = \prod_{i \in U} (1 - xA_i) = \sum_{i=0}^{n} \sigma_i x^i .$$

From this product, the values of σ_i can be computed:

$$\begin{aligned} \sigma_0 &= 1 \\ \sigma_1 &= -\sum_{i \in U} A_i \\ \sigma_2 &= \sum_{i \neq i'} A_i A_{i'} \\ &\vdots \\ \sigma_n &= (-1)^n \prod_{i \in U} A_i \end{aligned}$$

These equations can be rewritten as *Newton's identities*:

$$\begin{aligned} S_1 - \sigma_1 &= 0 \\ S_2 - \sigma_1 S_1 + 2\sigma_2 &= 0 \\ S_3 - \sigma_1 S_2 + \sigma_2 S_1 - 3\sigma_3 &= 0 \\ &\vdots \\ S_n - \sigma_1 S_{n-1} + \cdots + (-1)^{n-1}\sigma_{n-1} S_1 + (-1)^n n\sigma_n &= 0 \end{aligned}$$

This is a triangular linear system of equations, so the σ_i can be computed from the S_j by simply solving the equations in order. Since q is a prime number, we know that no redundant equations occur in this system. (Later on, we will discuss the case where q is a power of 2.)

The σ_i are the coefficients of the polynomial P whose roots are the reciprocals of the values A_i. To find the values A_i, we find the roots of P by factoring. Since q is a large number with respect to n, we cannot try out values to find the roots. An easy way to factor a polynomial in a prime order field is using the polynomial

$$Q(x) = x^q - x,$$

that has all field elements as roots. If P has no multiple roots (all A_i are different), then Q is a multiple of P. In other words, if $Q \bmod P \neq 0$, there is a conflict, and the participants must send a new reservation block. From now on, we assume that Q is a multiple of P.

Q can be split into two parts

$$Q(x) = Q_1(x) \cdot Q_2(x) = (x^{(p-1)/2} + 1)(x^{(p+1)/2} - x).$$

Because Q is a multiple of P, P can be split in two parts using the greatest common divisor:

$$P = P_1 \cdot P_2 = \gcd(P, Q_1) \cdot \gcd(P, Q_2).$$

The ordering of the roots of P can now be defined: first the roots of P_1, then the roots of P_2. The splitting and ordering of the roots of P_1 and P_2 is performed recursively, splitting up the polynomials P_1 and P_2 using $Q(x+1)$. This process continues using $Q(x+2)$, and so on, until P is completely split up in polynomials of degree 1. (These polynomials of degree 1 are multiples of the factors $1 - xA_i$ in the original definition of P.)

A participant does not need to compute all the roots to find his position in the ordering. Consider user Alice, who wants to know the position of her value A in the reciprocals of the roots of P.

- First, she computes the coefficients of P using the method described above, and verifies $Q \bmod P = 0$ to make sure there is no conflict.
- To know if the root corresponding to her value of A is in the first or second half of P, she computes $Q_1(A^{-1})$. This value is 0 or 1.
- If it is 0, she is in the first group of participants, and she computes $P_1 = \gcd(P, Q_1)$, and proceeds recursively with P_1, computing $Q_1(A^{-1} + 1)$.
- If it is 1, she is in the second group of participants. She continues the recursion with $P_2 = \gcd(P, Q_2)$. Her position in the whole group is the sum of her

position in the second group and the size of the first group. Her position in the second group is computed by recursion on P_2, and the size of the first group is the degree of P_1. The degree of P_1 is the degree of P minus the degree of P_2, so P_1 does not have to be computed to find its degree.

- Alice repeats until she has a first-degree polynomial with root A^{-1}.

The expected number of steps of this algorithm is μ_n, where the numbers μ_k are defined as:

$$\begin{cases} \mu_1 = 0 \\ \mu_k = \dfrac{1}{2^k}\left(\mu_k + \displaystyle\sum_{i=1}^{k}\binom{k}{i}\mu_i\right) \end{cases}$$

It is possible to prove that $\mu_k = \log_2 k + O(1)$ [OD91], so that the expected number of steps is about $\log_2 n$. The proof of this approximation is surprisingly hard, and lies outside the scope of this book.

In this version of the protocol, the participants can influence their position by choosing certain values for the A_i. A simple extra requirement prevents this without using extra transmission or computation. As stated above, Alice computes in the i^{th} recursion the value of $Q_1(A^{-1}+i)$. If it is zero, she may go in the first group, and if it is one, she has to wait for the second group. Instead, we define that Alice may go first if this value is equal to the i^{th} bit of a random number. A good choice for this random number is the sum S_1, because it is a random number influenced by all the participants.

The actual protocol as proposed by Bert den Boer uses a field of characteristic 2. In this case the number of elements q is a power of two*. Computations in this field are easier, but there are some differences. In a field of characteristic 2, the equality

$$x^2 + y^2 = (x+y)^2$$

holds (the "freshmen's dream"). This makes that the sums on the even powers of the A_i do not give new information, since they can be computed from the sums of other powers. The reservation block consists of the odd powers

$$A_i, A_i^3, A_i^5, \ldots, A_i^{2n-1}.$$

The S_j can be computed from the totals:

$$S_1 = \sum_{i\in U} A_i;\ S_2 = S_1^2;\ S_3 = \sum_{i\in U} A_i^3;\ S_4 = S_2^2, \text{ and so on.}$$

The computation of the polynomial coefficients from these sums is different from the previous case, because the even lines of Newton's identities are useless. The linear system that yields the σ_i is not triangular anymore, so the equations cannot be solved the easy way. Solving this equation is related to decoding BCH codes; two algorithms that compute the σ_i from the S_j are described in [Bur71]. Once these coefficients σ_i are found, the factorization of P is performed using the polynomial

$$Q(x) = x^q + x.$$

$Q(x)$ can be split up as

$$\text{trace}(x)\cdot(\text{trace}(x)+1),$$

* The case where q is a power of another small prime is not considered here.

where the trace function is defined as

$$\text{trace}(x) = (x + x^2 + x^4 + \cdots + x^{q/2}).$$

As before, the polynomial P is split up in two halves P_1 and P_2 using the splitting of Q. For the recursion, the splittings

$$Q(x) = \text{trace}(\alpha x) \cdot (\text{trace}(\alpha x) + 1),$$

$$Q(x) = \text{trace}(\alpha^2 x) \cdot (\text{trace}(\alpha^2 x) + 1), \ldots$$

are used, where α is a generator of the field GF(q).

These computations can be performed quickly using special hardware, especially when a so called *(optimal) normal base* is used [AMOV91]. This might speed up the computations considerably over the previous case where q is a prime.

Collision detect by Pfitzmann

The first collision resolve algorithm we consider is an algorithm that is developed by Andreas Pfitzmann [Wai89], at the same time that we made the collision detect algorithm after the system of Bert den Boer. Pfitzmann's protocol uses a trick that divides the colliding participants in two groups, that use the algorithm recursively until the collision is completely resolved.

If the turnaround time is equivalent to the transmission of t bits, and the maximum collision size the algorithm can handle is $s_{\max}$, the collision resolve takes a total time of approximately

$$(s-1) \cdot (\log_2 n + 3\log_2 s_{\max} + t) \text{ bits},$$

where s is the actual size of the collision. The derivation is shown below. The participants only have to perform very simple computations, so that no special hardware is needed. The main problem of this protocol is the dependency on the turnaround time. Because the protocol relies on repetitive resolving of collisions, the turnaround time counts in every step.

The protocol is based on averaging. The header blocks are pairs $(1, M_i)$, where M_i is a random number chosen from $\{1,\ldots, M_{\max}\}$. The elements of the pairs are added up separately using a modulus* that is so large that the total is the actual sum of the headers. Each participant can now compute the average

$$\frac{\sum_{i \in C} M_i}{s}.$$

The participants divide themselves in two groups depending on their value of M_i. The participants with values of M_i that are lower than average may go first. They start immediately resending the header blocks, resulting in a new collision of about half the size. They recursively resolve the collision. When they all sent their messages, the group with values higher than average may send. They don't have to send the header block, because they can compute the total in advance by subtracting the total of the first group from M_i.

* The DC algorithm requires the use of a finite abelain group, so the addition must be modular.

The protocol is detailed in Figure 10.

1	Put a random element of $\{1,\ldots,M_{\max}\}$ in M. Send the message $(1,M)$, and wait for the total. Put the total in C,S.
2	If $C = 1$, the collision is resolved; send the message, and stop. Put the average S/C in A. If $M \leq A$: Send the header $(1,M)$, and wait for the total. Put the total in C',S'. If $C' = C$, there is a conflict; go to step 1. Otherwise, put C',S' in C,S and go to step 2. Alice is in the second group: Wait for the next total, and put it in C',S'. Wait for the C' messages in the first group. Put $C-C',S-S'$ in C,S, and go to step 2.

Figure 10: Pfitzmann's collision resolve protocol for user Alice.

The transmission time of this scheme is easy to calculate. The headers are added up using two moduli. The first modulus must be $n+1$, and the second modulus can be $s_{\max}M_{\max}+1$, so that all collisions of size at most $s_{\max}$ yield a correct sum. This makes the length of a header block equal to

$$\lceil \log_2(n+1) \rceil + \lceil \log_2(s_{\max}M_{\max}+1) \rceil.$$

Resolving a collision of s participants takes $s-1$ steps, so the total time to resolve a collision is

$$(s-1)\cdot\left(\lceil \log_2(n+1) \rceil + \lceil \log_2(s_{\max}M_{\max}+1) \rceil + t\right).$$

The optimal choice for the number $M_{\max}$ is about $s_{\max}^2 \log_2 s$.

If more than $s_{\max}$ participants collide, a special protocol must resolve the collision again. For example, all participants can send a second header with a higher value of $s_{\max}$.

Collision resolve after Chaum

We extend the slot reservation algorithm of [Cha88] to a collision resolve algorithm. The header block consists of r bits. It contains a 1 in one position and 0 in the other positions. If a collision is detected, the rules of Chaum's slot reservation algorithm determine the order in which to send.

The problem with this protocol is the detection of conflicts. Conflicts can be detected by sending an extra message containing a 1, just as with the previous protocol of Pfitzmann. The sum of these will be the number of participants in the collision. A conflict is detected if the number of ones does not match the number of colliding participants.

The transmission time of this scheme is quadratic in the number of colliding participants. It can be computed in a similar fashion as the previous protocol. The length of a header block is

$$r + \lceil \log_2(n+1) \rceil \text{ bits.}$$

The optimal choice for r is hard to calculate, because the probability of a conflict of header messages depends on the size of a collision. If we assume that the collision size s is equal to $s_{\max}$, the optimal choice for r would be

$$r = \frac{s_{\max}}{4}\left(s_{\max} + \sqrt{s_{\max}^2 + 8t}\right).$$

The probability of a collision grows fast with s, so that the optimal choice of r is probably slightly higher*. If all collisions are the same size, and the turnaround time is zero, r must be $s_{\max}^2/2$, and the expected total time for the headers is

$$\frac{e}{2}s^2 + e\lceil \log_2(n+1) \rceil \text{ bits.}$$

The dependency on the turnaround time is rather small. The expected extra time for a turnaround time of t is $t \cdot e$, because the expected number of retries for conflicts is e.

The computation needed for this slot reservation scheme is trivial.

Collision resolve after den Boer

The slot reservation scheme by Bert den Boer [Boe88] can also be extended to a collision resolve protocol. The effect of this protocol is that after one header block the collision is immediately resolved, and the parties can start sending in order. This makes the turnaround time count only once per collision.

The header block is the message

$$1, A_i, A_i^2, \ldots, A_i^{k-1}$$

where A_i are random numbers. The parameter k determines the maximal size of a collision. For best results, the value k can be chosen a small number, and if the collision turns out to involve more than k participants (by examination of the first sum), more powers of the same number A can be sent to resolve the collision completely.

The protocol works just like the slot reservation version. The characteristic 2 version can also be used. As a collision detect protocol, the computation done by the participants is a lot less because the expected number of participants involved in the collision is much less than $\#U$.

This new protocol is probably the most efficient protocol in practice, because it uses little data for the header blocks, and is relatively insensitive to the turnaround time. The only problem is that is uses a large amount of computation.

Overview

Table 2 shows an simplified overview of the discussed transmission rules.

The systems marked with a * are collision detect systems that we adapted here from slot reservation systems. The original author of the slot reservation system is given in those cases.

The column "Reference" lists the first publication of the protocol.

The column "Transmission" lists the approximate amount of bits it takes to determine the order between the participants involved. In the case of a slot reservation system, it is a function of the number of participants n. For a collision detect system,

* To compute this, we need a probabilistic model of the collision size.

we assume that all collisions have the same size s, and the turnaround time is zero.

The column "Turnaround" gives an impression of the sensitivity of the system for long turnaround times. It shows the expected number of repetitions of the scheme. During every repetition, the participants have to wait for the outcome. To compute the total delay for one round, the turnaround time has to be multiplied by the entry in the column.

The column "Computation" gives the amount of computation. A "–" sign means that the protocol needs negligible amount of computation, while "pol. factor(k)" denotes the amount of computation to factor a polynomial of degree k in a field.

The column "Type" designates whether the system is a Slot Reservation (S.R.) or Collision Detect (C.D.) system.

System	Reference	Transmission	Turnaround	Computation	Type
Chaum	[Cha88]	$\frac{e}{2}n^2$	$e \approx 2.72$	–	S.R.
den Boer	[Boe88]	$2n\log_2 n$	1	pol. factor(n)	S.R.
Pfitzmann	[Wai89]	$s\cdot(\log_2 n + 3\log_2 s)$	s	–	C.D.
Bos / Chaum *	–	$e\cdot(\log_2 n + \frac{1}{2}s^2)$	e	–	C.D.
Bos / den Boer *	[BB89]	$2s\log_2 s$	1	pol. factor(s)	C.D.

Table 2: Comparison of transmission rules.

Acknowledgement

I would like to thank Adri Olde Daalhuis for his work on the μ_k and on the asymptotic approximation of the r.

References

[AMOV91] **G. B. Agnew**, **R. C. Mullin**, **I. M. Onyszchuk**, and **S. A. Vanstone**: *An Implementation for a Fast Public-Key Cryptosystem*, Journal of Cryptology **3** (No. 2, 1991), pp. 63-79.

[BB89] **J. N. E. Bos** and **H. den Boer**: *Detecting Disrupters in the DC protocol*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, May 1989), pp. 320-327.

[Boe88] **H. den Boer**, personal communication.

[Bur71] **H. O. Burton**: *Inversionless Decoding of Binary BCH Codes*, IEEE Trans. Information Theory **IT-17** (No. 4, July 1971), pp. 464-466.

[Cha81] **D. Chaum**: *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms*, Comm. ACM **24** (No. 2, February 1981), pp. 84-88.

[Cha88] **D. Chaum**: *The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability*, Journal of Cryptology **1** (No. 1, 1988) pp. 65-75.

[DBP79] **D. W. Davies**, **D. L. A. Barber**, **W. L. Price**, and **C. M. Solomides**: *Computer Networks and their Protocols*, John Wiley and Sons, 1979.

[OD91] **A. Olde Daalhuis**: personal communication.

[Pfi85] **B. Pfitzmann**: personal communication.

[PP89] **B. Pfitzmann** and **A. Pfitzmann**: *How to break the Direct RSA-Implementation of mixes*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, May 1989), pp. 373-381.

[PW87] **A. Pfitzmann** and **M. Waidner**: *Networks without User Observability*, Computers & Security **6** (No. 2, April 1987), pp. 158-166.

[Tan88] **A. S. Tanenbaum**: *Computer Networks*, Prentice-Hall, 1988.

[Wai89] **M. Waidner**: *Unconditional Sender and Recipient Untraceability in spite of Active Attacks*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, May 1989), pp. 302-319.

[WP89] **M. Waidner** and **B. Pfitzmann**: *The Dining Cryptographers in the Disco*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, May 1989), p. 690.

3

An Efficient Voting Scheme

Introduction

The network for untraceable sending of Chapter 2 can be used for several purposes. The present chapter describes the implementation of an efficient voting scheme using this network. A *voting scheme* is a protocol that performs a secret-ballot election with the following properties:

- *Privacy*: the voting is anonymous*, so that the votes are kept secret;
- *Security*: the participants are unable to influence the outcome more than by just their vote;
- *Robustness*: it is hard for participants to disrupt the execution of the protocol;
- *Verifiability*: every voter is convinced of the correctness of the election outcome;
- *Efficiency*: the protocol can be applied in practice over an electronic network.

We discuss the features listed above in analogy of traditional statuary democratic decision procedures as in carrying or rejection of motions or in the election of representatives. In the rest of the chapter, we discuss the voting schemes that exist in the literature compared to the new scheme. Then, after introduction of the terminology used, we describe the protocol in detail. Proofs of the necessary properties are given, and extensions are shown that make the voting scheme more flexible and efficient, while preserving the necessary properties.

Privacy

In a voting scheme, privacy protection (that is, secrecy of the ballots) is never perfect. Any set of colluding participants can compute the total of all other users by comparing their total to the total of all users. Sometimes, this information is enough to determine a vote completely. For example, if the voting is unanimous, or if all but one user cooperate, the identity of that user can be determined with certainty.

The privacy protection of voting schemes must be compared to the information that

* If the voting were *not* anonymous, we can just make every voter publish her vote. This yields a protocol that is easily seen to have all other features, but cryptographically not very interesting.

can be obtained by collusion. In the ideal case, this is all the information that a colluder gets. Some voting schemes give cryptographic protection, meaning that getting extra information involves breaking a certain cryptographic assumption. The voting system introduced here gives the intrinsic privacy protection of the DC scheme of Chapter 2. That means that the privacy of the users is unconditionally protected, depending on the reliability of the set of neighbours in the key graphs. The details are explained in Chapter 2. The privacy does not depend on any computational assumption; even if the underlying cryptosystem is broken, the ballots are still secret.

In a practical election, the privacy of people using the voting machines* is not very high. The most apparent problem is that a voting machine cannot prove to the user that it does keep the votes secret. The identical paper ballots are better; the votes are collected in a container that will only be opened after the election is over. The user can wait until a few more votes are entered to make sure his vote is mixed in. The largest privacy restriction is that the total of the group of about 1000 voters to which they belong is made public.

Security

A voting scheme is secure if all a user can do to influence the total is cast a single valid vote. This assures that the result of the voting is correct.

In our voting scheme, breaking the security is as hard as finding a discrete logarithm.

In practice, security is maintained by checking a register of all legitimate voters.

Robustness

A protocol is robust if the participants are unable to disrupt the protocol by sending messages (whether valid or invalid). It does not matter if the protocol identifies the disrupter, but it must come to a proper ending.

Disruption of the practical voting involves the collusion of many parties. Computation of the tally is performed over many polling stations involving different people. Formal reports of all tasks in the polling station are collected and verified.

Verifiability

Verifiability of a protocol denotes that all participants can verify the outcome of the protocol. In principle, one could let each participant verify the tally by counting the votes himself, but this is very impractical. A nice example of a relatively efficient way to do this is shown in [Cha81]. The DC protocol does not have a high degree of verifiability. The voting scheme described here uses a new local verification method for the protocols using DC scheme.

The verifiability of the real election is high in theory, since every step in the tally is open to public scrutiny.

* A voting machine is a device for automatic voting. It is used in voting stations where a few hundred people vote. Voting is done by pressing a few keys. It does not get the identity of the voters. In principle however, it could produce a list of all votes in order, so that all votes can be reconstructed. In practice, most machines are mechanical, so that this is much harder.

Efficiency

Efficiency is hard to define, especially if one wants to make claims about applicability in the real world. There are a lot of protocols that are very practical in principle, but are not implementable for other reasons (insufficient infrastructure, for example). The protocol shown here involves as little transmission as possible. The protocol requires only five rounds of the DC protocol (as shown at the end of this chapter). Since the rounds themselves take an amount of time that is logarithmic in the number of participants, the entire protocol also takes time logarithmic in the number of participants. All other known protocols take time at least quadratic in the number of participants. The infrastructure that is used for the DC protocol is thought to be practically implementable. Needed networks like the addition network are already in existence today.

One aspect of the scheme is that all messages must be synchronized, but this is probably handled automatically by the devices that perform the transmissions.

The current election scheme requires only messages per polling station. In this sense, the scheme is very efficient.

Related Work

In [Cha81], the first voting scheme is described. A *mix network* is introduced of trusted mail relaying machines (called *mixes*), that provides an untraceable sending protocol. The mixes decrypt RSA-encrypted messages and forward them per batch to hide the identity of the senders. This makes the privacy depend both on the size of the mix batches and on the security of RSA. The voting scheme proposed in the article uses the mix network for sending encrypted ballots. The election outcome is easy to verify by scanning two public lists, one with the ballots and one with voter pseudonyms. It is one of the few protocols that really give verifiability to all participants. The protocol involves sending only four messages per voter, but the voters receive a large amount (linear in the number of participants) of data. Although the direct implementation is broken [PP89], the system can be adjusted to prevent this attack.

The issue of *multiparty computations* has generated a number of articles. A multiparty computation is the computation of a function value without revealing the input values. An election is a special case of a multiparty computation. The easiest way to implement such a protocol is to use a publically trusted computer with secret input channels, that computes the function value and publishes it. There is a proof in [BCC88, CDG88] that any computation using such a publically trusted computer can also be done without one, by using a multiparty computation. This method, however, is inefficient in general.

One of the first articles about multiparty computations is [DLM82]. It describes a protocol for multiparty computations, using a voting scheme as an example. This voting protocol is similar to that of [Cha81], but it involves more messages, it can be disrupted by every participant, and it requires that the participants know each other's identities. It

is not efficient enough to be practical.

[Yao82] also describes a general protocol, with a voting scheme as example. The protocol is the first to give unconditional privacy protection. On the other hand, it has a fixed number of participants, needs a lot of messages, and is not robust. It is not a practical protocol in the sense that it does not consider problems like transmission and computation overhead.

The protocol described in [GMW86] is again a general purpose protocol that can be used for voting applications. It is robust against any number of saboteurs up to half the participants. The result is also purely theoretical, so there has not given been much concern to efficiency.

[Ben87] is the final version of the voting protocols described in [CF85], [Coh86], [BY86]. (Josh Benaloh was earlier called Cohen). The protocols described in the earlier articles are much more practical, but have as main drawback that there is no privacy towards a certain participant called the *government*. [Coh86] suggests splitting the government up in *tellers*, so that any one part can protect privacy independently of the others. In [Ben87], both incriminating privacy and disrupting the election needs collusion between a group of tellers. The size of these groups varies with a security parameter—the larger one group, the smaller the other. The protocol depends on broadcasting by all parties.

The scheme in [Cha88] gives unconditional privacy using the DC scheme. The protection against cheating and disruption is based on the RSA assumption. Verifiability is excellent, because an official list is made public that allows anyone interested to count the votes. It uses a lot of messages, so that it is rather inefficient.

The scheme described in [BCC88, CDG88] is a very powerful, very general, and very unpractical scheme for multiparty computations. This article gives an overview of the exact power of multiparty protocols, and it contains an overview of blob implementations.

The protocol in [HT88] is also a multiparty computation scheme, based on [BCC88]. It is a theoretical result, with not much concern for practical applicability. To give unconditional privacy protection, it needs a secure channel between every pair of participants, an unrealistic assumption.

[MP89] is a protocol that allows computation of the sum of a set of private numbers. The privacy protection is obtained by using a DC-like network. Although the protocol has many similarities with the voting scheme presented here, it is not a voting scheme, because the participants are not convinced of the correctness of the outcome.

There are more articles on multiparty computations, but they only address the theoretical possibility of such a computation without considering a practical implementation.

The new voting scheme explained in this chapter was originally described in [BP88], but it was thoroughly extended since then. The protocol combines unconditional privacy with high efficiency.

An overview of the discussed protocols is shown in Table 3. The columns in the table describe the given properties for each protocol. The properties "verifiability" and

"efficiency" are not listed in the table as they are hard to compare fairly.

The entry "RSA" in the table means that the given property is as hard as breaking RSA (computing a root modulo n). "Pub. key" means it is as hard as breaking some particular public key cryptographic function. "Priv. key" means the breaking of a private key (classical) cryptosystem. "Residue" means the recognizing of a number mod n as being a certain power. This is believed to be about as hard as factoring [Ben87, p. 31]. The entry "No" in the "Robustness" column means that the protocols can be stopped by any participant who stops acting according to the protocol.

Reference	Privacy	Security	Robustness	Remarks
Cha81	RSA	RSA	RSA	Privacy depends on batch size
DLM82	Pub. key	Pub. key	No	Difficult to comprehend
Yao82	Uncond.	Pub. key	No	
CF85	Residue	RSA	RSA	No privacy from government
Cha88	Uncond.	RSA	DC	DC system
Ben87	Residue	Residue	Uncond.	privacy $\leftrightarrow$ robustness tradeoff
HT88	Uncond.	Priv. key	Priv. key	Needs secure private channels
Present	Uncond.	D. log.	DC	DC system

Table 3: Comparison of voting schemes.

Explanation of the Protocol

The Dining Cryptographers untraceable sending system that we described in Chapter 2 had the important feature that messages that were sent simultaneously summed together. This is just what is needed to count ballots. One can make a simple voting scheme by letting each voter send a 1 for "yes" and a 0 for "no" and use the DC protocol for the tally.

The ballot of user i is called b_i, and the key shared between user i and j is called t_{ij}, so that the values actually published over the addition network are

$$b_i + \sum_{j \in U} t_{ij} .$$

This gives us unconditional privacy protection for the voters, and some verifiability. The robustness can be violated if there are disrupters during this round. The round cannot be opened, because the ballots are private information. The security can only be guaranteed if users prove that their ballot was indeed a 0 or 1.

The problems of robustness and security are solved using blobs. In the introduction we explained that a *blob on a value* c is used to prove that c was chosen before a certain time; it commits a user to that value. This is done by sending a special encryption of c, called the *blob value*. This blob is *opened* at a later time by sending c together with the encryption key.

In the voting protocol, a variant on this idea is used. The blobs on the ballots cannot be opened as such, because the ballots are secret. The validity of the votes is proved

without revealing their value using a special protocol explained later on.

The voting protocol uses several rounds. In the first round, the users commit to their ballots with a blob. The second round of the protocol perform the actual tally in the *voting round*. This is the transmission of the ballots using the DC scheme, exactly as described earlier. The third and later rounds are used to prove that the ballots are valid.

Local verification

The outputs given by the users must be verifiable. In the ideal situation, all user's outputs are published, so that any interested party can check them. [Ben87] proposes the use of "bulletin boards" where interested parties can get the information that users put in it. A more efficient variation is to let the user's outputs be locally verified by a group of users called the *observers*. The observers can be chosen from the group of voters. These users check the outputs of the users that are physically close to them (for example, all incoming users of a certain node of the network). The observers get a special (traceable) round in which they send a 0 if there is no problem, and a 1 if there is. This way, the public can quickly verify the results without large overhead. Remember that anybody is assumed to have access to the network, so that this does not incriminate privacy; it only increases efficiency. A disrupting observer will quickly be found because his transmissions are traceable.

Blobs

The blobs used for this voting protocol are much like the blobs described in [BKK87; CDG88]. Other blobs are also possible; see the extensions at the end of this chapter. The blob of value c with key k is

$$\text{blob}(k,c) = \alpha^k \beta^{2c} \pmod{p}.$$

In this formula:

- p is a "safe prime", that is a prime number so that $\frac{p-1}{2}$ is also a prime number.
- p must be large enough to make the discrete log problem modulo p unfeasible [DH76; PH78; Odl84, McC90, LO90].
- α and β are different generators of the multiplicative group modulo p.
- c is the *contents* of the blob, chosen from the set $\{0,\ldots,\frac{p-3}{2}\}$.
- k is the *key* of the blob. It is a uniformly distributed random element from the set $\{0,\ldots,p-2\}$.

The validity of α, β, and p can easily be verified by the public. The size of the number p determines the level of security of the protocol. It also influences the amount of computation to generate a blob. In practice, p must have a size of about 150 decimal digits. This gives a high level of security, while still allowing the computation of blobs on a small computer.

Computations on blobs are in the field GF(p), so from now on all equations are implicitly modulo p.

Vic* *breaks* the blob by guessing the value of c from the blob value alone. This is just as hard as guessing c without knowing the blob value. The blinding factor α^k

* As in the introduction, Vic stands for the verifier and Peggy for the prover.

makes all values for the blob equally probable, independent of the value of c. This gives Peggy unconditional privacy protection, because a given value of the blob does not give any information about the value of c.

Peggy *cheats* the blob by opening it with another value than c. This is unfeasible by the discrete log assumption. To see this, assume that Peggy can open a blob in two different ways: $\text{blob}(c,k) = \text{blob}(c',k')$. Since she has a solution to the equation

$$\alpha^k \beta^{2c} \equiv \alpha^{k'} \beta^{2c'},$$

she can compute the discrete log of β^2 with respect to α as follows: she does this by taking the inverse* d of $c - c' \pmod{\frac{p-1}{2}}$, and computing

$$\alpha^{(k-k')d} \equiv \beta^{2d(c'-c)} \equiv \beta^2;$$

the last step follows from $d(c - c') \equiv 1 \pmod{\frac{p-1}{2}}$, so that $2d(c - c') \equiv 2 \pmod{(p-1)}$. From the discrete log assumption, Peggy is unable to find a discrete log, so we conclude that Peggy cannot cheat a blob.

Vic cannot find the value of Peggy's blob, even if he can break the discrete logarithm. In the voting system, this means that the ballots remain secret even if discrete log is broken during or after the election.

The blobs used have a nice and simple "additional" property, that allows a user who issues two blobs to open the sum or difference without having to open the blobs themselves. This turns out to be very convenient for our voting scheme. The additional property can be written as

$$\text{blob}(k,c) \cdot \text{blob}(m,d) = \text{blob}(k+m, c+d).$$

Note that there are three different moduli in this equation: the product of the blobs is taken modulo p; the sum of the keys is taken modulo $p-1$; and the sum of the values is taken modulo $\frac{p-1}{2}$. The *product* of two blobs is a blob on the *sum* of the values, that is opened by revealing the *sum* of the keys. We assume that the addition network we use can do both addition and multiplication computations, so that we can combine blobs, values and keys over the network.

Blob validation

To prove that a blob contains the value 0 or 1 without revealing its contents is a well known protocol [BCC88]. Assume that Peggy wants to prove that (previously sent) $\text{blob}(k,c)$ has a value c that is 0 or 1. To prove this, Peggy and Vic do the *blob validity protocol*:

- Peggy chooses random blob keys x and y and sends Vic the blobs: $\text{blob}(x,0)$; $\text{blob}(y,1)$, in random order.
- Vic sends a challenge "open" or "equal".
- On challenge "open", Peggy opens both blobs sent in the first round by revealing x and y.
- On challenge "equal", Peggy takes the blob of the first message that contains the value that is equal to c. She then opens the blob on the difference between her blob and this blob. This difference blob contains a 0 as value. In other words, Peggy sends $v = k - x$ if $c = 0$, and $v = k - y$ if $c = 1$.

* This inverse exists, because the absolute value of c–c' is smaller than the prime number $(p-1)/2$.

Vic verifies the received value v by checking if the product of the designated blob of the first message and blob(v,0) equals Peggy's blob.

We now prove that this is a zero-knowledge proof (see the introduction):

The protocol is complete, because if Peggy does not cheat, Vic computes, if c = 0:

$$\text{blob}(x,0) \cdot \text{blob}(v,0) = \text{blob}(x + k - x,0) = \text{blob}(k,c)$$

or, if $c = 1$:

$$\text{blob}(y,1) \cdot \text{blob}(v,0) = \text{blob}(y + k - y,1) = \text{blob}(k,c).$$

In any case, Vic will accept Peggy's answers.

The protocol is sound, because if Peggy does not have a proper value of c, she cannot answer both challenges of Vic at the same time. In this case, she must guess what challenge she will get:

- If she guesses that the challenge will be "open", she can follow the protocol as it stands and open the blobs of the first round.
- If she guesses "equal", she can take one of the blobs of the first round to contain the improper value c, so that she can open the difference blob.

If Peggy makes the wrong guess, she cannot answer the challenge, so that she will be caught with probability $\frac{1}{2}$ in both cases. This protocol can be repeated as many times as Vic needs to be convinced that Peggy has a valid blob. In practice, about 25 times is a reasonable value, giving a probability of about $2.98 \cdot 10^{-8}$.

To show that the protocol is zero-knowledge, we show a simulation of the protocol. Vic generates a simulated transcript as follows:

- First, Vic guesses the challenge he will receive in the second round.
- If the challenge is "open", the simulated first round consists of sending two random blobs with values 0 and 1.
- If the challenge is "equal", Vic takes a random blob key v and a random group element u. The first round consists of sending the pair blob(k,c)/blob(v,0);u in random order.
- The third round contains the blob values if the challenge was "open", and the value v if the challenge was "equal".

It is easy to verify that these values satisfy the same checks as Peggy's answers, and have the same probability distribution as the messages of a real transcript.

In the voting protocol, a modified version of the blob validity protocol will be used that allows all participants to prove simultaneously that their blobs are correct.

Description of the Protocol

We assume an underlying "addition" network that can add messages modulo $p - 1$ and $\frac{p-1}{2}$, add messages modulo 2 (exclusive-or), and multiply messages modulo p. This network can then perform the voting protocol. The shared keys used by the DC scheme are only used in those rounds where privacy protection is needed. This reduces the amount of keys needed in the scheme. Also, the scheme allows certain values to be checked locally so that disrupters can be found quickly.

The protocol consists of five rounds:

- Issuing of blobs on the keys used in the DC sending
- The actual vote counting round using the DC system
- Issuing of blobs for proving the validity of the ballots
- Construction of a challenge bit
- Response to the challenge

The last three rounds have to be repeated a number of times to provide the needed level of accuracy. (This can be done simultaneously, so that the protocol consists of five rounds again; see the end of this chapter.)

To aid understanding the formulae, all are formed using blobs. For example, we write blob(x_i, 0) even though this is just α^{x_i}. The former form is easier to verify, because all other formulae use blobs too. The proofs use only the blob properties mentioned in the previous section and simple arithmetic. Note that the addition property of blobs involves three different moduli; all three moduli are used by the addition network.

The values of the parameters p, α and β can best be chosen just before the voting protocol is initiated. This gives cryptanalists as little time as possible to solve the discrete log problem for α and $\beta^2 \pmod p$ to break the protocol. It does not matter if the discrete log is found after the protocol is over, but if it is broken during the protocol, the outcome will be unreliable.

The output of user i in round n is denoted as $[n]_i$, and the total is written $[n]$.

First round

Initially, the users share keys t_{ij} that are to be used for the second round. Similarly, the users share blob keys k_{ij} for blobs on those keys. In the DC scheme, it is required that $t_{ij} = -t_{ji}$ to make the keys cancel out, and $t_{ij} = 0$ means that there is no key used between user i and j. The numbers k_{ij} are constructed the same way, so that also $k_{ji} = -k_{ij}$ and $k_{ij} = 0$ if $t_{ij} = 0$. The only difference is that the k_{ij} are random numbers from the set $\{0,\ldots,\frac{p-3}{2}\}$, while the t_{ij} are from the set $\{0,\ldots,p-1\}$. Thus, the numbers k_{ij} can easily be generated and distributed in advance together with the t_{ij}.

In the first round, user i sends a blob on the keys t_{ij} she is going to use during the voting round.

The output of user i is

$$[1]_i : \text{blob}(\sum_{j\in U} k_{ij}, \sum_{j\in U} t_{ij}).$$

The network multiplies all messages modulo p, yielding (nobody cheating)

$$[1] = \prod_{i\in U} [1]_i = \text{blob}(\sum_{i\in U}\sum_{j\in U} k_{ij}, \sum_{i\in U}\sum_{j\in U} t_{ij}) = \text{blob}(0,0) = 1.$$

All numbers involved in this message are random numbers. If the result of this transmission is not equal to 1, the blobs can be opened to catch a disrupter. Then a new round, with new values of t_{ij} and k_{ij}, can be performed. The individual results of this round are saved; for efficiency, the users may do this themselves. The observers will also save the messages for later verification.

Second (voting) round

The actual tally is now obtained. The users send their ballots over the DC network in additive mode. This is sent untraceably using the keys t_{ij} from the first round:

$$[2]_i : b_i + \sum_{j \in U} t_{ij} .$$

The addition is done modulo $\frac{p-1}{2}$. Because p is very large, $\frac{p-1}{2}$ will be bigger than the number of voters. Thus, the total result of this DC transmission will be the number of 1 ("yes") votes:

$$[2] = \sum_{i \in U} [2]_i = \sum_{i \in U} b_i + \sum_{i \in U} \sum_{j \in U} t_{ij} = \sum_{i \in U} b_i .$$

Cheaters and disrupters will not be detected until the fifth round. If everything works well, the total of these transmissions will be the election result. Of course, this round cannot be opened because the private vote information is in it.

Third round

The third round is the first step of the blob validity protocol that takes the last three rounds of the protocol. To make things more efficient, it is executed simultaneously by all users, adding the results.

We want to prove the validity of a blob on the ballot of user i. This blob does not occur explicitly in the protocol, but it can be computed:

$$\frac{\text{blob}(0,[2]_i)}{[1]_i} = \frac{\text{blob}(0, b_i + \sum_{j \in U} t_{ij})}{\text{blob}(\sum_{j \in U} k_{ij}, \sum_{j \in U} t_{ij})} = \text{blob}(-\sum_{j \in U} k_{ij}, b_i).$$

This blob can be constructed by everyone who stored the values of the earlier messages sent by user i. The key of this blob is

$$K_i = -\sum_{i \in U} k_{ij} .$$

The blob keys and blob contents are only known to user i. Of course, opening this blob would reveal b_i.

In the first round of the blob validity protocol, every user sends a pair of blobs containing a 0 and a 1 as follows: user i chooses a random bit s_i and two blob keys x_i and y_i, and sends

$$[3\text{a}]_i : \text{blob}(x_i, s_i), \text{ and}$$
$$[3\text{b}]_i : \text{blob}(y_i, 1 - s_i) .$$

All users do this simultaneously, and the blobs will be multiplied modulo p by the addition network, giving the products

$$[3\text{a}]: \prod_{i \in U} \text{blob}(x_i, s_i) = \text{blob}(\sum_{i \in U} x_i, \sum_{i \in U} s_i), \text{ and}$$

$$[3\text{b}]: \prod_{i \in U} \text{blob}(y_i, 1 - s_i) = \text{blob}(\sum_{i \in U} y_i, \#U - \sum_{i \in U} s_i).$$

The results of this transmission cannot be verified as such, because the blob keys and contents are not public. To make it possible to verify the messages, all users send the values of x_i, y_i and s_i. For privacy protection, the values must be sent over the

network using untraceable sending. The DC keys used are not shown here.

$$[3c]_i{:}s_i,\ [3d]_i{:}x_i,\ \text{and}\ [3e]{:}y_i.$$

These messages are added over the untraceable sending network modulo $\frac{p-1}{2}$ for [3c] and modulo $p-1$ for [3d] and [3e], giving the sums

$$[3c]{:}\sum_{i\in U} s_i,\ [3d]{:}\sum_{i\in U} x_i,\ \text{and}\ [3e]{:}\sum_{i\in U} y_i.$$

Using these sums, the products [3a] and [3b] can be verified in public using the equations

$$[3a] = \text{blob}([3d],[3c]),\ \text{and}\ [3b] = \text{blob}([3e], n-[3c]).$$

Fourth (challenge) round

The fourth round of the voting protocol is the second step of the blob validity protocol. The challenge has to be formed. The challenge is a trusted random bit formed by all users. Every user sends a random bit

$$[4]_i{:}r_i.$$

These bits are added up modulo 2 (exclusive-or) by the network, giving the challenge bit r:

$$[4] = \sum_{i\in U} r_i = r.$$

This round cannot be cheated or disrupted at all, because the only thing that matters is that the bit is not known in advance by the participants.

Fifth (response) round

Now the response to the challenge has to be sent. The description of the blob validity protocol shows that the users can send two different responses, depending on the challenge: opening the blobs from message [3], and opening a difference blob between the ballot and either of the two blobs. All users respond in the same way, because they receive the same challenge.

If the challenge bit is 0, the blobs from message [3] are to be opened. The user just sends the values

$$[5a]_i{:}s_i,\ [5b]_i{:}x_i,\ \text{and}\ [5c]{:}y_i$$

over the network, adding up the results modulo $\frac{p-1}{2}$ and $p-1$. The sums must be the same as the sums [3c], [3d] and [3e]. Since the outputs are now unprotected by DC keys, the individual outputs of every user can be verified instead of only the total sum. The outputs of the users are verified locally by the observers:

$$[3a]_i = \text{blob}([5b]_i,[5a]_i),\ \text{and}\ [3b]_i = \text{blob}([5c]_i, 1-[5a]_i).$$

If the challenge bit is 1, the user must prove that her blob is the same as one of the blobs sent in round [3]. Which of the two blobs this is may vary between users, because it is dependent on the s_i. The responses are such that the values of b_i and s_i cannot be found from the responses.

If $b_i = s_i$, the user opens the difference blob between b_i and the first blob:

$$[5a]_i{:}K_i - x_i.$$

The messages will be added modulo $p - 1$. To simplify public verification, the user also sends

$$[5\mathrm{b}]_i{:}[3\mathrm{a}]_i = \mathrm{blob}(x_i, s_i),$$

that will be multiplied modulo p.

If $b_i \neq s_i$, the user opens the difference blob between b_i and the second blob:

$$[5\mathrm{a}]_i{:}K_i - y_i, \text{ and correspondingly}$$

$$[5\mathrm{b}]_i{:}[3\mathrm{b}]_i = \mathrm{blob}(y_i, 1 - s_i).$$

For both cases, the response can be verified using the formula

$$\frac{\mathrm{blob}(0,[2]_i)}{[1]_i \cdot \mathrm{blob}([5\mathrm{a}]_i,0)} = [5\mathrm{b}]_i.$$

This is because

$$\frac{\mathrm{blob}(0,[2]_i)}{[1]_i \cdot \mathrm{blob}([5\mathrm{a}]_i,0)} = \frac{\mathrm{blob}(K_i,b_i)}{\mathrm{blob}(K_i - x_i,0)} = \mathrm{blob}(K_i - K_i + x_i, b_i) = \mathrm{blob}(x_i, s_i) = [5\mathrm{b}]_i$$

or, if $b_i \neq s_i$ for this user,

$$\frac{\mathrm{blob}(0,[2]_i)}{[1]_i \cdot \mathrm{blob}([5\mathrm{a}]_i,0)} = \frac{\mathrm{blob}(K_i,b_i)}{\mathrm{blob}(K_i - y_i,0)} = \mathrm{blob}(K_i - K_i + y_i, b_i) = \mathrm{blob}(y_i, 1 - s_i) = [5\mathrm{b}]_i.$$

Apart from the local verification of the responses per user, the totals can also be verified:

$$\frac{\mathrm{blob}(0,[2])}{[1] \cdot \mathrm{blob}([5\mathrm{a}],0)} = [5\mathrm{b}].$$

Overview

Table 4 shows an overview of the voting scheme messages and verifications.

The column "Message" shows the message that user i sends in that round. The symbols are explained in the text. The column "Group" shows the group in which the computation of the total takes place.

The last column shows the verifications that are to be done. The verifications in round [5] that are indexed by i denote local verifications. As said before, this should preferably be done by local agencies. There could be a traceable sixth round for transmissions of the verification results. The local verifications are expected to be accompanied by corresponding verifications of the total result by all participants.

The two possibilities for round [5] are separated by a dashed line. Above the dashed line is the reply for a challenge (value of round [4]) of 0, the replies under the line are for challenge 1. In the latter case, the two different replies vary per user. Both replies are shown in the Message column; the network does not distinguish between them.

Round	Message	Group	Verification
[1]	$\text{blob}(\sum_{j\in U} k_{ij}, \sum_{j\in U} t_{ij})$	$\prod \bmod p$	$[1] = 1$
[2]	$b_i + \sum_{j\in U} t_{ij}$	$\sum \bmod \frac{p-1}{2}$	
[3a]	$\text{blob}(x_i, s_i)$	$\prod \bmod p$	
[3b]	$\text{blob}(y_i, 1-s_i)$	$\prod \bmod p$	
[3c]	s_i + DC keys	$\sum \bmod \frac{p-1}{2}$	
[3d]	x_i + DC keys	$\sum \bmod (p-1)$	$[3a] = \text{blob}([3d],[3c])$
[3e]	y_i + DC keys	$\sum \bmod (p-1)$	$[3b] = \text{blob}([3e],\lvert U\rvert-[3c])$
[4]	r_i	$\sum \bmod 2$	
[5a]	s_i	$\sum \bmod \frac{p-1}{2}$	$[5a] = [3c]$
[5b]	x_i	$\sum \bmod (p-1)$	$[3a]_i = \text{blob}([5b]_i,[5a]_i)$
[5c]	y_i	$\sum \bmod (p-1)$	$[3b]_i = \text{blob}([5c]_i,\lvert U\rvert-[5a]_i)$
[5a]	$\begin{cases} -\sum_{j\in U} k_{ij} - x_i \\ -\sum_{j\in U} k_{ij} - y_i \end{cases}$	$\sum \bmod (p-1)$	
[5b]	$\begin{cases} \text{blob}(x_i, s_i) \\ \text{blob}(y_i, 1-s_i) \end{cases}$	$\prod \bmod p$	$\frac{\text{blob}(0,[2]_i)}{[1]_i \cdot \text{blob}([5a]_i,0)} = [5b]_i$

Table 4: Overview of the transmissions.

Proofs

What remains to be done is a proof that the claimed properties of the voting scheme are fulfilled.

Privacy

In any voting system, electronic or otherwise, every group of participants can obtain information over the votes of the others by sharing votes. In our voting system, this is the only way to obtain information over a vote, under the assumptions of the underlying DC system. This means that if a collusion of participants does not split the *key graph* (see Chapter 2), all the votes are optimally protected; otherwise, privacy loss occurs.

If the DC system is not compromised, the privacy protection of the voting scheme follows from three facts:

- the unconditional protection of messages [2], [3c], [3d] and [3e] by the DC network;
- the unconditional protection of the contents of the blobs in [1], [3a], [3b];
- the zero-knowledge property of the blob validation protocol in rounds [3], [4] and [5].

Privacy loss occurs if the DC protocol is compromised. This happens if a group of participants share the keys used for the protection of their votes (of course losing their

own privacy). Using the terminology of Chapter 2, the keys of this group split the key graph in subgraphs. The users in the collusion can compute the total tally for each of these subgraphs, and no more. The amount of privacy loss for these participants depends on the size of the subgraphs; the smaller the subgraph, the bigger the privacy loss.

Robustness

The voting scheme is robust in the sense that any participant who tries to prevent the protocol from finishing successfully can be caught. The method to catch a disrupter varies per round.

If round [1] does not give a total of 1, the values of k_{ij} and t_{ij} can be published to catch disrupters (this may be called "opening", but it is not opening in the DC sense). Round [3] can just be opened if the public verification does not succeed. Round [4] cannot be disrupted at all. Round [5] uses a verification per user, that makes it very easy to catch a disrupter directly without opening the round.

The only problem is round [2], that cannot be opened immediately because the actual votes are in there. A disrupter in this round will be caught in round [5] with probability $\frac{1}{2}$ for every repetition.

Catching disrupters using opening in round [1] and [3] may cause a delay because opening a round does not reveal the disrupter immediately (see Chapter 2). If the disrupter is found, he can immediately be excluded from the tally, so that the protocol can proceed without further delay.

Security

Security of our voting system means that a single participant cannot influence the total otherwise than casting a single valid vote. Sending invalid messages causes the user to be treated as a disrupter, and to be caught by the previously mentioned methods. Users that try to use ballots with values other than 0 or 1 will be caught by the validation protocol.

If a user can open a blob in two ways (thus, break discrete log), he can cast any number of votes at once, because he can break the blob validation protocol.

Verifiability

Verifiability means that any participant can convince himself of the validity of the outcome of the protocol. In our scheme, this depends on two factors.

First, the tally can be checked simply, since the tally is equal to the total of round [2]. Every participant can perform the verifications of the other rounds, so that he knows the tally was correct.

Second, the participants must trust the total of the outputs as computed by the network. This is more a problem of the DC network than a problem of this voting scheme. In theory, every participant can check this for himself, since the totals are computed from publicly accessible numbers. In practice this might give problems; a good solution seems to have several independent agencies compute the total.

Efficiency

Our protocol can be implemented using an existing addition network for the DC system. This protocol makes very efficient use of the network, since it uses only five rounds of the network. We assume that the DC network is efficiently implementable.

Since the number of rounds is fixed, the protocol takes time proportional to the logarithm of the number of voters, since the DC network is logarithmic in the number of participants. This is a lot faster than all proposed systems published until now. All the discussed voting systems from the literature were more than linear in the number of participants. A disadvantage of the system is that a lot of keys have to be generated and stored for one instance of the protocol. On the other hand, compared to other uses of the DC system, this number is very small.

Extensions

There are several features that can be added to the protocol to make it more efficient in practice.

Parallel computations

The blob validity protocol of rounds [3] to [5] is executed several times to produce a high level of certainty. It is more efficient to execute these rounds in parallel, so that every round stands for a number of rounds in the original version.

The parallel blob validity protocol is executed as follows:

- Peggy sends Vic a list of pairs of blobs.
- Vic sends Peggy a list of random bits (a *challenge vector*).
- Peggy responds to the respective bits of the challenge vector.

If the parallel blob validity protocol is executed between two participants, it is not zero-knowledge [BCC88]. To see this, assume that Vic's challenge vector is a one-way function value of the first message. Now Vic cannot simulate the messages of the protocol. To simulate the protocol, Vic must start with the challenge vector. Because this is a one-way function value, the message from the first round cannot be found, so simulation is impossible.

In the voting scheme, several instances of the parallel blob validity protocol are executed together by all participants. Take a voter Peggy, who proves the validity of her votes; call the community of all other voters Vic. The difference with the two-party parallel blob validity protocol is that not only Vic, but also Peggy determines the challenge vector. It is no use for Vic to compute the challenge in a special way (for example, using the above mentioned one-way function), because Peggy can make the challenge a random number with her own input. This makes the protocol simulatable for Vic, and thus zero-knowledge. The actual proof is much the same as that at the beginning of this chapter.

This parallelization decreases the number of rounds needed to only five. Because the number of rounds is low, the protocol does not depend on a low turnaround time

(see Chapter 2).

Precomputation

Precomputation is the computation of values ahead of executing a protocol, so that during the protocol the values only have to be looked up in a table. The users can precompute all values they send, so that the protocol itself is not delayed by computations performed by the users. The computations performed by the addition network cannot be precomputed, but these are only simple additions and multiplications.

The blob values can be computed using a vector addition chain (see Chapter 4), saving over 40% of the computation time.

More options

The voting scheme allowed only two possible votes: "yes" and "no". This can easily be extended to more realistic elections with more options. The easiest way to extend this number is by doing multiple elections. They can be performed simultaneously, so that the number of rounds is the same as with one voting. This allows to make voting schemes with multiple nominees. Of course, it is also possible to let participants choose to abstain from voting. In those cases, the blob validity protocol must be adapted to reflect all possible votes instead of just 0 and 1. Instead of two blobs, the voter must send as many blobs in round 3 as there are legitimate votes.

Other blobs

The blobs used in this protocol can be replaced by other kinds, if the new blobs have the following features:

- Unconditional unbreakability, so that the ballots cannot be computed when the election is over.
- Some kind of addition property, to make the blob computations over the network possible.

A good alternative are the blobs used in [BD90]. The main advantage of these blobs is that the factorization of $p - 1$ can remain secret [BD90, footnote on page 2]. This allows to decrease the size of p while sustaining the level of security. Also, blobs based on elliptic curves or hyperelliptic curves can be used, possibly saving bits.

Verification at the nodes

The nodes of the tree that perform the additions are nice places to perform the local verification. The nodes that are closest to the participants do the local verification, and the top node does the public verification.

Less distribution of the results

In rounds [1], [3] and [5] the total result of the transmission is only needed for verification. Instead of the sum, the nodes could just distribute the result of the verification, consisting of only one bit (in principle). This reduces the time to distribute the results a little. Even more savings would occur in the "Parallel computations" and

"more options" versions proposed above. The transmission contains several rounds here, and the results will be a string of bits or even one bit, telling if something has gone wrong.

Conclusion

The DC protocol is not only applicable for untraceable sending, but also for an efficient voting protocol. Other multiparty protocols that use the additional and privacy protecting properties of the DC protocol are a challenging area for further research.

Acknowledgement

I thank George Purdy for the cooperation on the design of the protocol.

References

[BCC88] **G. Brassard**, **D. Chaum**, and **C. Crépeau**: *Minimum Disclosure Proofs of Knowledge*, Journal of Computer and System Sciences **37** (No. 2, October 1988), pp. 156-189.

[BD90] **J. F. Boyar** and **I. B. Damgård**: *A Discrete logarithm blob for Noninteractive XOR gates*, Technical report DAIMI **PB-327**, Århus University (August 1990).

[Ben87] **J. Benaloh**: *Verifiable Secret-Ballot Elections*, Ph.D thesis of Yale University (September 1987).

[BKK87] **J. F. Boyar**, **M. W. Krentel**, and **S. A. Kurtz**, *A Discrete Logarithm Implementation of Zero-Knowledge Blobs*, Technical Report **87-002**, University of Chicago (March '87).

[BP88] **J. N. E. Bos** and **G. Purdy**: *A voting scheme*, Rump session of Crypto '88. Does not appear in proceedings.

[BY86] **J. Benaloh** and **M. Yung**: *Distributing the Power of a Government to Enhance the Privacy of Voters*, Proc. **5**th ACM Symp. Principles of Distributed Computing (Calgary, AB, August 1986), pp. 52-62.

[CDG88] **D. Chaum**, **I. B. Damgård**, and **J. van de Graaf**, *Multiparty computation ensuring privacy of each party's input and correctness of the result*, Advances in Cryptology: Proc. Crypto '87 (Santa Barbara, CA, August 1987), pp. 87-119.

[Cha81] **D. Chaum**: *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms*, Comm. ACM **24**, 2 (February 1981), pp. 84-88.

[Cha88] **D. Chaum**: *Elections with Unconditionally Secret-Ballots and Disruption Equivalent to Breaking RSA*, Advances in Cryptology: Proc. Eurocrypt '88 (Davos, Switzerland, May 1988), pp. 177-182.

[CF85] **J. Cohen** and **M. Fisher**: *A Robust and Verifiable Cryptographically Secure Election Scheme*, Proc. **26**th IEEE Symp. Foundations of Computer Science (Portland, OR, October 1985), pp. 372-382.

[Coh86] **J. Cohen**: *Improving Privacy in Cryptographic Elections*, Technical Report **454**, Yale University, Department of Computer Science (New Haven, CT, February 1986).

[DH76] **W. Diffie** and **M. E. Hellman**: *New Directions in Cryptography*, IEEE Trans. Information Theory **IT-22 6** (November 1976), pp. 644-654.

[DLM82] **R. DeMillo**, **N. Lynch**, and **M. Merritt**: *Cryptographic Protocols*, Proc. **14**th ACM Symp. Theory of Computing (San Fransisco, CA, May 1982), pp. 383-400.

[GMT82] **S. Goldwasser**, **S. Micali**, and **P. Tong**: *Why and How to establish a Private Code On a Public Network*, Proc. **23**rd IEEE Symp. Foundations of Computer Science (Chicago, IL, November 1982), pp. 134-144.

[GMW86] **O. GoldReich**, **S. Micali**, and **A. Widgerson**: *Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design*, Proc. **27**th IEEE Symp. Foundations of Computer Science (Toronto, ON, October 1986), pp. 174-186.

[HT88] **M. A. Huang** and **S. Teng**: *Secure and verifiable Schemes for Election and General Distributed Computing Problems*, Proc. **7**th ACM Symp. Principles of Distributed Computing (Toronto, On, August 1988), pp. 182-196.

[LO90] **B. A. LaMacchia** and **A. M. Odlyzko**: *Computation of Discrete Logarithms in Prime Fields*, preprint.

[McC89] **K. S. McCurley**: *The Discrete Logarithm Problem*, Technical Report RJ-6877, IBM Research division, Almaden Research Center, San Jose, CA.

[McC90] **K. S. McCurley**: *The Discrete Logarithm Problem*, Proc. Symposium of Applied Mathematics, American Mathematical Society (1990), to appear.

[Mer83] **M. Merritt**: *Cryptographic Protocols*, Ph.D thesis of the Georgia Institute of Technology (February 1983).

[MP89] **C. A. Meadows** and **G. B. Purdy**: *Summing over a network without revealing summands*, BIT **29** (No. 1, 1989), pp. 110-125.

[Odl84] **A. M. Odlyzko**: *Discrete logarithms in finite fields and their cryptographic significance*, Advances in Cryptology: Proc. Eurocrypt '84 (Paris, France, April '84), pp. 224-314.

[PH78] **S. C. Pohlig** and **M. E. Hellman**: *An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance*, IEEE Trans. Information Theory **IT-24** (No. 1, January 1978), pp. 106-110.

[PP89] **B. Pfitzmann** and **A. Pfitzmann**: *How to break the Direct RSA-implementation of Mixes*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, April 1989), pp. 373-381.

[QGB89] **J. J. Quisquater**, **L. Guillou**, and **T. Berson**: *How to explain Zero-knowledge to your children*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 628-631.

[Yao82] **A. Yao**: *Protocols for Secure Computations*, Proc. **23**rd IEEE Symp. Foundations of Computer Science (Chicago, IL, November 1982), pp. 160-164.

4

Addition Chain Heuristics

Introduction

Modern cryptographic protocols allow a large variety of applications that were not possible with the classic methods. Almost all of these protocols are based on exponentiation modulo a large number. This is true for all protocols based on RSA, but also for systems using discrete log (such as the voting scheme described in Chapter 3).

The computation of such a power takes quite a lot of computation compared to classic cryptosystems (DES, for example). Several methods have been tried to improve the speed of the computation. The only known way to compute a power is by multiplications. While a lot of current research emphasizes improving the speed of those multiplications (see for example, [Mon85]), we try to decrease the total number of multiplications by rearranging the computation. In practice, this saves about 20% of the multiplications that are normally used to perform the exponentiation. This optimization can again be combined with improved multiplication methods to get maximal performance.

Although a lot of research is spent on addition chains, almost no work has been done in practically applying addition chains for doing RSA or other schemes. Here we show heuristics that perform well in constructing addition chains to compute powers in practical applications, both hardware and software.

A related problem is the computation of products of powers. Products of powers occur for example in digital signature systems and payment systems. A product of powers can be computed much more efficiently than the separate powers by combining the computation of the individual powers. We show a practical algorithm that computes a product of powers using these techniques.

The notation used in the literature for these algorithms is not suitable for determining memory usage; we introduce an new notation that closely resembles the actual computation of (products of) powers in a computer. This notation is also well suited for storage of addition chains.

The results in this chapter are also trivially applicable for computations in elliptic curves and other groups.

In the rest of this chapter, we first introduce the four different kinds of addition

processes. Then we discuss the literature on this subject. Then, for each of the four kinds, we discuss algorithms to heuristically compute them. In the appendix to the chapter, all algorithms used are shown in detail, so that the reader can try out for himself.

Addition chains

If a power is computed by repeated multiplication, the process can be described by all intermediate results. For example, a computation of x^{15} in five multiplications can be described by the sequence $x, x^2, x^3, x^6, x^{12}, x^{15}$. In fact, the only interesting aspect of this sequence are the exponents, so we could just as well have written 1, 2, 3, 6, 12, 15 to describe the process. Such a list of numbers is called an *addition chain*.

More specifically, an *addition chain of n* is a list of (integer) numbers satisfying

- the first number is 1;
- every number, except the first, is the sum of two previous numbers;
- the list is ascending*, and the last number is n.

The number n is called the *target* of the chain.

The *length* of an addition chain is the number of elements minus one (that is, the number of additions necessary to make the chain, or the number of multiplications to compute the corresponding power). For reasons of efficiency, short addition chains are preferred over long ones. Unfortunately, the computation of a minimal length addition chain is hard; this problem is even NP-complete [DLS81]. Our goal is to construct short (but not necessary optimal) addition chains at a reasonable cost.

To get an impression of the length of an addition chain, we define:

$l(n)$: The length of a minimal addition chain with target n.

It is easy to see that $l(n) \geq \lceil \log_2 n \rceil$. The binary method, the simplest non-linear method that is known, produces addition chains with expected length $\frac{3}{2}\log_2 n$ and maximum length $2\log_2 n$, so that large savings (more than 30%) using addition chains are not to be expected. Even so, small savings are sometimes worth the trouble. Asymptotically, $l(n)$ behaves like $\log_2 n$ [Bra39]:

$$\lim_{n\to\infty} \frac{l(n)}{\log_2 n} = 1.$$

Addition sequences

The notion of an addition chain can be generalized to make chains with more than one target number; we call those *addition sequences*. Formally, an *addition sequence of a set of numbers* (the numbers called again targets) is a list of numbers satisfying the following conditions:

- the first number is 1;
- every number, except the first, is the sum of two previous numbers;
- every number in the set occurs in the list;
- the list is ascending, and the last number is a target.

* Every number in the chain, except the first, is larger than the previous number.

The length of an addition sequence is one less that the number of elements. Addition sequences can be used to make addition chains for large numbers, as will be shown later on. In practice, addition sequences do not occur often. We use them mainly to construct addition chains and vector addition chains, because these are relatively easy to construct.

Vector addition chains

A product of powers can be computed more efficiently than computing the powers separately and multiplying the product. For example, to compute the product $x^9y^{13}z^{22}$, we could use the intermediates

$$x, y, z, xz, yz, xyz^2, xy^2z^3, x^2y^3z^5, x^4y^6z^{10}, x^8y^{12}z^{20}, x^9y^{13}z^{22}$$

to obtain the product in eight multiplications. Computing the product using these steps is much less work than computing the powers first; the power z^{22} alone already needs six multiplications. Like the addition chain, we only have to describe the exponents to describe the computation. We get a list of *vectors*:

[1 0 0], [0 1 0], [0 0 1], [1 0 1], [0 1 1], [1 1 2], [1 2 3], [2 3 5], [4 6 10], [8 12 20], [9 13 22].

Such a list is of vectors is called a *vector addition chain*. A vector addition chain is defined as a list of vectors satisfying:

- the first vectors are the unit vectors;
- every vector, except if it is a unit vector, is the sum of two previous vectors;
- the last vector is the target.

The length of a vector addition chain is the number of elements minus the unit vectors (which is again the number of additions necessary to make the chain). There are many practical applications for vector addition chains. Examples are the blobs of Chapter 3, and the signatures of Chapter 5.

Vector addition sequences

The obvious extension of the three problems defined here is the *vector addition sequence*. This is the most general problem; all other posed problems are special cases. The only mentioning of vector addition chains that we found is [Pip76]. In this paper, Nicholas Pippenger gives a lower bound for the length of vector addition sequences. This lower bound implies that in the worst case, the savings of vector addition sequences over vector addition chains or addition sequences will not be as large as the savings of vector addition sequences over vector addition chains. In other words, vector addition sequences are not very good for saving multiplications. Asymptotically, vector addition sequences do not perform better than a set of vector addition chains or addition sequences.

The general term we use for addition chains, addition sequences, vector addition chains and vector addition sequences is *addition processes*. Table 5 shows the four types of addition processes.

	One target	Multiple targets
Powers	Addition Chain	Addition Sequence
Products of powers	Vector Addition Chain	Vector Addition Sequence

Table 5: Addition Processes

An addition process is *evaluated* if the computation with corresponding intermediate results is made. Evaluation of an addition chain for an exponentiation would be the computation of the actual power. Evaluating an addition process takes as many steps as the length of the process. Except for the time it takes to evaluate an addition process (which depends on the length of the process), we also consider the amount of memory needed (which depends on the maximum number of intermediate results needed during evaluation).

Related Work

There is quite a lot of literature on addition chains. Most of the research is on the asymptotic behaviour of the minimum length of addition chains. Only a few articles address the actual computation of usable addition chains. For a long list of asymptotic results, see the references of [Cos90].

The original problem statement is by Arnold Scholz [Sch37]. He was mainly interested in the minimum length of addition chains, but he notes that constructing addition chains is useful for exponentiation with as few multiplications as possible (for manual computation; there were no computers at that time). The problem is elaborated by A. Brauer [Bra39] and, a long time later, by Paul Erdös [Erd60]. These two articles are the basis of a lot of research on the minimum length of addition chains.

The first statement of the vector addition chain problem is from Richard Bellman [Bel63]. He claims (incorrectly!) that "It is easy to determine the minimum number of multiplications required to generate a^N from a". The problem was worked out by E. G. Straus [Str64]. With this partial solution, the editors note that "The proposer agrees that the problems he has posed are not easy and that the minimum chain is not known [...]". Also, they note that Bellman was working on a computational algorithm. We could not find a reference to this algorithm.

Donald Knuth gives a good introduction to the addition chain problem in the second volume of his well-known series "The Art of Computer Programming" [Knu69]. He considers some theoretical bounds on the length of addition chains, but also discusses practical methods to construct chains for small numbers (up to about five digits). He also introduces the *m-ary method*, that produces practical addition chains for large numbers (hundred digits or more) that are more efficient than the standard binary method. He introduces the addition sequence problem as an open problem.

More research on the length of addition chains was done by Andrew Yao [Yao76] and Edward Thurber [Thu76].

Nicholas Pippenger [Pip76] poses addition processes in their full generality. He

was the first to note the relation between vector addition chains and addition sequences. We shall see this equivalence later in this chapter.

Knuth shows, in the second (thoroughly rewritten) edition of his book [Knu81], that an addition chain can be "reversed". The fact that this reversal yields a correspondence between addition sequences and vector addition chains, is mentioned in Exercise 39.

This correspondence, and a conversion from vector addition chains to addition sequences is shown by Jorge Olivos [Oli81]. Although he refers to [Pip76] in his paper, he does not say that Pippenger already mentions the conversion. He does not consider the practical application of his algorithm.

Peter Downey, Benton Leong, and Ravi Sethi [DLS81] prove that the construction of an optimal addition sequence is NP-complete. They do this using addition graphs.

It is surprising to see how little use is made of addition chains in practice. For example, H. R. Chivers [Chi84] describes a way to compute exponentiations on a small computer. He did not know of the existence of addition chains, although the m-ary method existed for fifteen years; he even claims that "An exponentiation with 100 bit integers would involve 100 repeated squarings and an average of 50 other multiplications."

The practical applicability of addition chain methods was only realized recently. Nowadays, there are several exponentiation devices that use addition chains (this is stated in [Bri90], but he does not say anything more that that).

Apparently, our work [BC89] inspired Y. Yacobi to make another algorithm for the practical construction of addition chains [Yac90]. His method is very elegant, has a simple description, but it uses some more multiplications and quite a lot more memory than the on-the-fly method presented in this chapter.

This chapter is based on material of [BC89] together with the vector addition chain work [BC89a] and [Cos90].

Addition Graphs and Addition Machines

We introduce two alternative representations of addition chains that are more useful for describing addition processes than the list of intermediate results. The first notation occurs in the literature, and allows the useful definition of "reversing" an addition process. The second notation is new; it allows us to see how efficient an addition chain is in both memory and operation efficiency.

Addition graphs

It is often convenient to see the computation of a power as a graph instead of a list. Addition graphs were introduced in [Pip76] and they turn out to be a useful tool for describing the vector addition chain algorithm.

We define an *addition graph for a set of values* as a directed loop-free graph* with a vector in each node, fulfilling the following properties:

- Every source (node with no incoming edges) contains a different unit vector. The dimension of these vectors is equal to the number of sources.
- Every non-source node contains the sum of the values of the nodes of the incoming edges.
- There is a sink (node with no outgoing edges) for every target value. This is for reasons of symmetry; most sinks will have one incoming edge from a node with the same contents.
- Every internal node (node that is neither a source nor sink) has two or more incoming edges.

(Integers are treated as vectors of length one.)

Figure 11 gives an example of an addition graph. If all internal nodes of an addition graph have two incoming edges, the nodes of the graph form a (vector) addition chain. A node in an addition graph that has more than two incoming edges specifies multiple additions; the order in which the additions take place is not specified, but the number of additions is fixed. Such nodes can be split into a set of nodes with two incoming edges, making the order of the additions explicit. We call the process of splitting nodes to make an addition graph with at most two incoming edges per node *specification*. Specification of the addition graph in Figure 12 produces the addition graphs of Figure 11. Note that specification always produces the same number of nodes.

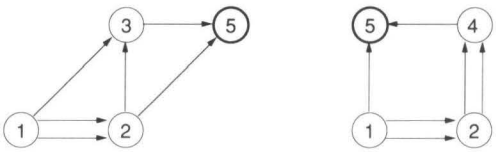


Figure 11: Two addition graphs for the number 5.

Generalization of an addition graph is the converse process of specification. All nodes with exactly one outgoing edge can be removed, unless it is a source. At removal of a node, the incoming edges become the incoming edges of the node it pointed to.

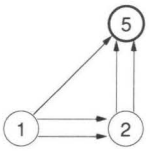


Figure 12: Generalized addition graph for the number 5.

An addition graph with one source and one target can be converted to an addition

* Directed loop-free graph: the edges of the graph are ordered pairs (denoted by arrows), and no node is connected to itself.

chain by specification; the nodes contain the values of an addition chain. This is easy to see from the definition of an addition chain. If we also consider addition graphs with more sources or more targets, we get vector addition chains and (vector) addition sequences. For an overview, see Table 6 (also see Table 5).

Addition chains with	Are convertible to and from:
One source, one sink	Addition chains
One source, more sinks	Addition sequences
More sources, one sink	Vector addition chains
More sources, more sinks	Vector addition sequences

Table 6: Conversion of addition graphs.

If we call addition processes that correspond to the same generalized addition graph *equivalent*, we get an equivalence relation. For example, the addition chains 1, 2, 3, 5 and 1, 2, 4, 5 of Figure 11 are equivalent. Although equivalent addition processes have the same length (because specification always produces the same number of nodes), the memory usage does not need to be the same. Investigating this equivalence relation might be subject for further research.

An interesting operation on addition graphs is the reversal of all edges. If the graph is generalized, it is easy to see that the resulting graph is also an addition graph. The reversal of the corresponding addition graph converts addition chains to other addition chains and vector addition sequences to other vector addition sequences. The interesting thing is that vector addition chains are converted to addition sequences, and vice versa. This relationship, first stated by Nicholas Pippenger [Pip76], is used by Jorge Olivos [Oli81]. The graph representation makes the relation between vector addition chains and addition sequences easy to see. We call this conversion of addition sequences to and from vector addition chains *reversal*.

The graph representation is used in the literature for several different purposes [DLS81, Oli81, Pip76], because it beautifully shows the structure of an addition process. Since we are also interested in the memory usage of addition processes, which it not shown by the addition graph, we introduce still another notation.

Addition machines

There are several properties of addition processes that cannot be expressed with either the addition chain or the addition graph notation. The first property is the order in which the additions take place. The second property, which is very important in practical implementation of addition chains, is the amount of memory needed during evaluation. The third property is the "doubling", the adding of a number to itself, which is a special case in most implementations of addition processes. We introduce a new notation, the "addition machine program", that can make these properties explicit. An addition machine program is a description of an addition process that specifies all details of the addition chain without being machine dependent.

An *addition machine* is a hypothetical device that evaluates addition chains (see Figure 13). The addition machine consist of an *adder*, the unit that does the actual addition of numbers. The result of the addition of two numbers is stored in the *accumulator register*. Intermediate results that are needed later on in the addition process are stored in a *register*. The adder adds the contents of the accumulator to one of the registers, or it adds the accumulator to itself. The latter case is a doubling. The number of registers that is needed to evaluate an addition process determines the memory usage of the addition process.

The addition machine executes a *program* that consists of a sequence of *instructions*. Initially, the accumulator contains 1. (In the discussion of vector addition chains, the unit vectors are initially in a given subset of the registers.) Then, the program is executed by executing all instructions in order. At the end of a program, a determined subset of the registers contains the *result* of the execution.

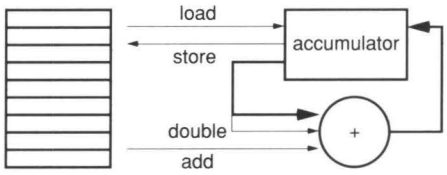


Figure 13: The Addition Machine

The instruction set of the addition machine is:

- **double**: double the contents of the accumulator (add it to itself).
- **add** *register*: add the contents of *register* to the accumulator.
- **store** *register*: store the contents of the accumulator into *register*.
- **load** *register*: store the contents of the register into the accumulator.

In Figure 13, thin lines show connections that are only active for certain instructions. The words next to these lines show for which instructions the lines are active.

The memory usage of an addition process can now simply be defined as the minimum number of registers that is needed to execute the program on the addition machine. This number accurately represents the memory usage in reality.

The instruction **double** is in principle not necessary in addition processes, because it could be replaced by the sequence **store** 1; **add** 1 (if 1 is a register that is used nowhere else in the program). We choose to use **double** explicitly, because it is often treated separately in the implementation. This can be for efficiency reasons (a squaring is less expensive than a multiplication) or out of necessity (in elliptic curve groups, addition of a point to itself is a different operation than adding different points). If the addition process is evaluated, some of the **add** instructions can accidentally involve two equal numbers, but this will seldomly happen; we ignore this possibility in our performance approximations.

Any addition process can be converted to an addition machine program. The length

of the process is equal to the number of **add** and **double** instructions in the program*.

Evaluation of the corresponding value of an addition process is trivial by emulation of the addition machine on a real computer, with corresponding interpretations of the instructions. For example, to compute a power of x using an addition chain, the instructions are interpreted as follows:

- the initial value of the accumulator is x, instead of 1
- **double** becomes a (modular) squaring
- **add** becomes a (modular) multiplication
- **store** and **load** retain their meaning

The same thing can be done for different addition processes (for example: vector addition chains to compute products of powers) of for different groups (for example: computing of elliptic curve elements). Using an addition machine program, an addition process can be stored efficiently for later evaluation.

Quick introduction to ABC

We describe all algorithms in the programming language ABC [Pem90]. This language is specially suitable for demonstrating algorithms. The notation is typographically modified to improve legibility.

A program definition starts with a header that consists of the words **how to** and a template showing how the program is to be called. The template contains the names of the program's arguments. This adds a new command to the language.

A simple example:

```
how to greet name:                          greeting program
    write "Hello ", name, "!" /             / produces "end of line"
```

If this program is entered, we could type "greet "Jurjen"", resulting in

```
greet "Jurjen"
Hello Jurjen!
```

There are the following data types in ABC:

- Numbers, e.g. 3215031751.
- Strings, e.g. "This is a string". Strings are delimited by " or '.
- Lists, e.g. {"a"; "a"; "c"}. Lists are always sorted and may contain doubles.
- Tables, e.g. {[1]: "a"; [3]: "b"; [9]: "c"}. They are generalizations of arrays.
- Tuples, e.g. (1, "a"). Used for grouping of data.

In this thesis, we use the following typographic conventions for ABC programs:

- **bold** font is used for keywords (originally, ABC uses capitals).
- *italic* font is used for variable names.
- roman font is used for functions. Functions can be defined using **how to return**.

* The notion of length can be generalized to give a different weight to **add** and **double**.

Making Addition Sequences

We make addition sequences with a very simple algorithm, that performs remarkably well. The algorithm can be adapted to produce sequences that have special properties.

The addition sequences produced by this algorithm are used in the addition chain and vector addition chain algorithms. The only application of addition sequences we know of is in these two algorithms; we have not seen any practical protocol that uses addition sequences.

The protosequence algorithm

The algorithm we use to make addition sequences is called the *protosequence algorithm*. It is a skeleton algorithm that produces addition sequences from higher to lower elements using an auxiliary function that determines the exact behaviour of the algorithm. This auxiliary function is called "new.numbers", and it produces a list of numbers that is to be included in the addition sequence.

To compute an addition sequence for the number in the list *a*, we execute the following algorithm.

```
how to make sequence a:                    make addition sequence for list a
    put a in p
    insert 1 in p
    insert 2 in p                          initialize protosequence p
    put {1; 2} in sequence                 initialize sequence
    while max p > 2:
        put max p in f                     f scans downwards
        insert f in sequence
        for n in new.numbers p:            generate new lower elements
            if n not.in p:
                insert n in p              put into protosequence
        remove f from p
    write sequence                         print the result
```

The function "new.numbers" in the above algorithm outline is the *protosequence function*. The function returns a list of numbers. An example definition is shown below. The protosequence function determines what addition sequence will be generated. The function is applied to the list *p*, the *protosequence*. The result of the function is a list of numbers with the following properties:

- The largest number *f* of the protosequence must be the sum of two numbers coming from the function result or the protosequence.
- All numbers in the function result must be between 3 and $f - 1$, inclusive.

The list produced by the protosequence function may be empty, if the number *f* happens to be the sum of two numbers from the protosequence. Numbers from the list that already occur in the protosequence are discarded by the skeleton program. Note that the skeleton program does not use any time-consuming operations; the management of the protosequence is rather simple, since the number of elements of the protosequence is small.

If the protosequence function fulfills the above conditions, the variable *sequence* contains at the end of the algorithm an addition sequence for the numbers in *a*. To see this, we check the four properties of addition sequences. Obviously, the first number is 1, since it is put directly in *sequence* by the skeleton program. To see that every number is the sum of two previous numbers, consider what happens if a new number *f* is inserted into *sequence*. The prototype function produces a list of numbers so that *f* is the sum of two of these numbers. These numbers are inserted in *p*, and they will be put into *sequence* at a later stage. The third property is that every number in *a* occurs in the list, which is trivially true from the algorithm. The last property is the result of the property of ABC to sort lists automatically. A practical algorithm in another language can easily keep the lists as a sorted array.

Note that the algorithm produces addition sequences from higher to lower numbers. Also, the algorithm treats the number 1 and 2 in a special way: they are always included in the output. This is done to simplify the protosequence function definition. The algorithm assumes that *a* contains an element larger than 1. It is easy to make it work for all cases, but this is omitted for simplicity.

The choice of the protosequence function depends on the application. For the discussion, we call the elements of the protosequence in order from largest to smallest *f*, $f_1, f_2, \ldots$. The simplest protosequence function just always returns the number $f - f_1$:

```
how to return simple.numbers p:                simplest protosequence function
    put max p in f
    put f max p in f1                          f max p: largest element <f of p
    return {f-f1}
```

To use this protosequence function in the program, we define "new.numbers" as:

```
how to return new.numbers p:                   define function new.numbers
    return simple.numbers p
```

This very simple solution does not produce excellent addition sequences, but it demonstrates the idea. The addition sequence algorithm would produce for the numbers

12, 17, 32

the (not optimal) addition sequence:

1, 2, 3, 6, 9, 12, 15, 17, 32.

A much more complicated prototype function is introduced in [BC89]. We will not

discuss it here.

The protosequence function can be adapted to produce addition machine programs instead of the values of the addition sequence. An ABC program that does this is shown in the appendix to this chapter.

On-the-fly Algorithms

Until now, we only considered precomputed addition processes, that are evaluated later on. In some applications, addition processes are used only once (for example, the computation of blob values used in Chapter 3). If an addition process is used only once, it is advantageous to spend less time generating the process, even if this results in a longer process, if the time to generate the process is reduced. An algorithm that uses an addition process only once can save memory by evaluating the process as it is generated. Such an algorithm is called an *on-the-fly algorithm.*

Unfortunately, an on-the-fly algorithm for addition sequences is not known. To evaluate an addition sequence, it must be generated first. This problem occurs for example in the on-the-fly addition chain program that is shown later on.

Making Addition Chains

There are many ways to make addition chains. [Knu69] gives precomputation methods to make addition chains for small numbers. We are only interested in large numbers (a hundred digits or more). We first give some examples of on-the-fly addition chain algorithms that occur in the literature, then we introduce a new on-the-fly algorithm for addition chains. Finally, we show how this algorithm can be adapted for precomputation purposes.

The binary method

The simplest example of an on-the-fly addition chain is the *left-to-right binary method.* Such an addition chain can easiest be constructed by repetition of the following: if n is even, insert $n/2$ into the chain; otherwise, insert $n-1$. The resulting chain consists of all numbers that form an initial segment of n in binary notation.

For example, the number 105 written in binary is 1101001. The resulting addition chain consists of the numbers 1, 2, 3, 6, 12, 13, 26, 52, 104, 105; in binary these are the numbers 1, 10, 11, 110, 1100, 1101, 110101, 110100, 1101000, 1101001.

In practice, the left-to-right addition chain algorithm is used on-the-fly like this:

```
how to left2right binary method x power n:        compute power x**n
    put (floor (2 log n)) + 1 in v
    put x in result                               addition chain element: 1
    for j in {2...v}:
        put result*result in result               double addition chain element
        if n bit (v – j)=1:                       n bit (v – j) is d_{v–j}
            put result*x in result                set last bit to 1
    write result
```

Where "bit" is an auxiliary function defined in ABC as

```
how to return n bit b: return (floor(n/2**b)) mod 2
```

Although this ABC program for "bit" takes an exponentiation, the function takes a negligible amount of computation on most (binary) computers.

This left-to-right binary method computes x^n without using auxiliary registers (except for two counters j and v). The variable *result* contains during the computations the powers of x corresponding to the elements of the addition chain. The number of multiplications needed to compute an n^{th} power depends on the number of digits v and on the number of nonzero digits w. As is easily seen from the program, the number of multiplications is

$$v + w - 2,$$

and if we approximate w by $1 + \frac{v-1}{2}$, we get the approximate number of multiplications

$$\tfrac{3}{2}(v-1).$$

There is also a right-to-left binary method; the algorithm is shown in the appendix.

The *m*-ary method

A generalization of this algorithm is the *m-ary method* [Knu81]. This method uses a different number base. More specifically, we write n as

$$n = \sum_{i=0}^{v-1} m^i d_i, \text{ where } 0 \le d_i < m,$$

with $d_{v-1} > 0$, and we put in the addition chain the following numbers:

- $1,\ldots, m-1$;
- for j in $1,\ldots,v$, the number consisting of the first j digits of n;
- all numbers from the preceding line, with the last digit replaced by 0.

The number base m may be any number, but the resulting addition chains are much more efficient if m is a power of two.

In our program notation, we would get:

```
how to left2right m ary method x power n:        compute power x**n
    put (floor (m log n)) + 1 in v
    put {[1]: x} in aux                          initialize auxiliary array
    for j in {2...m - 1}:
        put aux[j - 1]*x in aux[j]               [a] elements 2 ... m - 1
    put 1 in result                              for simplicity, start chain with 0
    for j in {1...v}:
        put result**m in result                  [b] multiply with m
                                                 (more than one step in the chain)
        if digit > 0:                            if d_{v-j} is not zero*
            put result*aux[digit] in result      [c] first j m-ary digits of n
    write result                                 print the result
digit:                                           definition of digit:
    return (floor(n/m**(v - j))) mod m           returns the j^th m-ary digit of n
```

This algorithm needs $m-2$ extra registers. The computation of "digit" is assumed to be negligible on the computer implementing the algorithm.

We estimate the number of multiplications that is necessary to compute a n^{th} power with the m-ary method. The addition chain consists of three different steps (see the corresponding locations in the program):

[a] The initial steps to generate the numbers 1 to $m-1$. This takes $m-2$ multiplications.

[b] The steps that multiply a number with m. This is executed v times, and each step takes $l(m)$ additions.

[c] The steps to compute the next prefix of n. The number of additions is equal to the number of digits of n that are not zero. Call this w.

During the first execution of the loop, the multiplications [b] and [c] are trivial multiplications by 1, so we do not count them. This makes the total number of multiplications to compute a n^{th} power equal to

$$m + (v-1)l(m) + w - 3.$$

We assume m is a power of two, so we have

$$m = 2^k$$

$$l(m) = k$$

$$v = \lfloor \log_m n \rfloor + 1 = \left\lfloor \frac{\log_2 n}{k} \right\rfloor + 1$$

$$\text{average } w = 1 + (v-1)\frac{m-1}{m}$$

Substituting this, we get the expected value of the number of multiplications:

$$2^k + \left\lfloor \frac{\log_2 n}{k} \right\rfloor (k + 1 - 2^{-k}) - 2.$$

For practical values of n of 150 to 200 decimal digits, the optimal value for k is 5,

* The word "digit" notifies a local function, a so-called *refinement*: the definition of "digit" appears at the end of the program.

giving expected 635 multiplications for a number n near 2^{512} (about 10^{154}). The binary method needs about 767 multiplications for those numbers, so the 2^5-ary method saves 17% of the work.

The window method

The m-ary method can be improved upon quite easily. The method puts the initial numbers $2,\ldots, m-1$ in the addition chain, while most of these numbers are not needed at all in the computation. The even numbers except 2 can also be removed. If j is such a number, it will always occur in the addition chain as $x, 2x, 2x+j$, which can be replaced by $x, x+\frac{j}{2}, 2x+j$. This saves both the time to compute these numbers, and the memory to store them.

This idea was the start of the method we used in the article [BC89]. Basically, the algorithm does the following:

- Split the binary description of n into "windows" consisting of odd binary numbers separated by strings of zeros. Make sure that the first window is the largest*.
- Produce an addition sequence for the numbers that occur in the windows.
- Produce the final chain, consisting of doublings starting from the value of the first window, adding in numbers of the addition sequence at the proper positions.

The total length of the addition chain is the sum of the length of the addition sequence, the number of doublings, and the number of windows in the chain minus one. A complete ABC program for the window method is shown in the appendix.

For example, to compute an addition chain for the number 496300971, we first write the number in binary and split it in windows:

1110111001010011111001110101011

The windows are 111011 (59), 101 (5), 1111 (15), 111 (7), and 101011 (43). The addition chain becomes:

1, 2, 4, 5, 7, 11, 15, 16, 27, 43, 59	made from window values with protosequence algorithm above
118, 236, 472, 944, 1888	doublings
1893	next window value (5) gets added in
3786, 7572, 15144, 30288, 60576, 121152	
121167	window value 15
242334, 484668, 969336, 1938672, 3877344	
3877351	window value 7
7754702, 15509404, 31018808, 62037616, 124075323, 248150464, 496300928	
496300971	the final result n

The length of this addition chain is 37. This chain is shorter than the addition chains

* This is without loss of generality; the first window can be chosen larger even without influencing the resulting addition chain.

for this number made with the other methods mentioned above. The binary method produces a chain of length 45, and the m-ary methods generates chains of length 42, if m is 4 or 8, and 49 or more for other values of m. This behaviour is typical: the m-ary method is better than the binary method, while the improved method is still better, using larger windows that the m-ary method. (Actually, a chain of length 36 exists; such a chain can be generated with the algorithm in the appendix, if the window size is chosen to be 7.)

There are two things left to consider: the window distribution and addition sequence algorithm we use.

Window distribution

In the optimal case, the windows must be chosen in such a way that the addition sequence that generates the window values has minimal length. Since this requires a lot of computation, we need a heuristic.

Our heuristic for generating the window distribution is as follows:

- Choose the size of the initial window. We take a fixed value, depending on the size of n.
- Find the minimal number of windows needed to split n. All window values must be smaller than or equal to the value of the initial window.
- Among the distributions with the minimal amount of windows, find the one with the smallest product of the window values.

Such a distribution can be computed in time linear in the number of bits of n. An algorithm for this is shown in the appendix. We tried other heuristics, but they took more time to calculate (more than linear in the number of bits of n), and they did not perform much better. Ideally, the heuristic should minimize the resulting addition sequence for the window values.

Addition sequence

When the window distribution is computed, an addition sequence must be constructed and generated for the window values. We use the protosequence algorithm for this. Since we do not know of an on-the-fly addition sequence algorithm, the generated addition sequence is stored as an addition machine program, and evaluated immediately after it is generated.

The addition sequences we generate for the window values must be generated quickly, so we want a very simple protosequence function, that can be computed in little time. The first example function "simple.numbers" is a bit too simple, because it gives bad results if the quotient of the highest two elements of the protosequence f/f_1 becomes large. We use the function "quick.values", that uses a simple trick to avoid this:

- If the number f/f_1 is smaller than 2, produce $f-f_1$.
- If f is even, produce $f/2$.
- If f is odd, produce the difference of f and the largest odd number in the protosequence that is smaller than f.

The last step of the window method is a series of doublings, starting from the initial window value, with the addition of the window values. This is very easy to implement.

The expected total number of steps for the window method is hard to compute. Random tests of a 512-bit number gave an average of 608 multiplications for random 512-bit numbers, if the optimal initial window of size 6 is chosen. This is not only 21% less than the binary method, but also still 5% less than the 2^5-ary method.

Precomputation

We precompute an addition chain if it is to be used for a large number of exponentiations with the same exponent. The precomputed addition chain can be stored as an addition machine program, so that evaluation is very efficient. The chain could even be stored as a computer program that computes the corresponding power directly, or encoded as an addition machine program.

To use the window method for precomputation, we suggest using the window method with several initial window sizes, and using the shortest resulting chain. Also, a more effective protosequence function can be used that recognizes several special cases. The appendix shows statistical results on the resulting addition chain lengths.

Making Vector Addition Chains

It is very advantageous to use vector addition chains if they are applicable, since they give a large savings for only little work. Also, it is possible to do a vector addition chain based computation without actually storing the chain, thus saving memory.

Vector addition chains can be generated from vector addition sequences by reversing the addition sequence. The addition sequence is made again by the protosequence algorithm shown above. Since the addition sequences are generated backwards, the vector addition chain is generated in the forward direction, so that it can be evaluated while it is generated. The resulting algorithm is surprisingly simple and gives large savings if the number of factors is large.

The memory usage of the algorithm depends on the addition sequences generated. The protosequence function has a special form the make the resulting vector addition chain use as little memory as possible.

The appendix shows the algorithm to compute vector addition chains, with statistical information about the performance.

Precomputation

Precomputed computation of vector addition chains can decrease the length because there is time for a more sophisticated protosequence function. Using this technique only is effective if the length of the vector is small (two or three). For details, see the appendix.

Conclusion

We discussed several ways to use addition processes in practice using heuristic techniques. An overview of the cases is shown in Table 7. "Yes" means that a new heuristic algorithm is introduced in this chapter, and "?" means that no efficient algorithm is known.

We constructed heuristic precomputing algorithms for addition chains, addition sequences and vector addition chains; the algorithm for vector addition sequences of [Pip76] cannot much be improved upon, as is proved in the same article. The addition processes can easily be stored in the form of an addition machine program.

Addition chains and vector addition chains can also be generated on-the-fly. This saves memory and time over precomputation algorithm if the process is only used once. The explicit algorithms are shown in the appendix.

	On - the - fly	Precomputation
Addition chains	Yes	Yes
Addition sequences	?	Yes
Vector chains	Yes	Yes
Vector sequences	?	[Pip76]

Table 7: Overview of the algorithms.

Several attempts by us to generate addition chains using other heuristics have failed. For example, we tried to apply simulated annealing (also known as statistical cooling), but this method seems not to apply to the addition process problem.

References

[BC89] **J. N. E. Bos** and **M. J. Coster**: *Addition Chain Heuristics*, Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 400-407.

[BC89a] **J. N. E. Bos** and **M. J. Coster**: *Heuristics for addition chains*, Report CS-R8945, Centrum voor Wiskunde en Informatica (Amsterdam, November 1989).

[Bel63] **R. Bellman**: *Problem 5125*, American Mathematical Monthly **70** (September 1963), p. 765.

[Bra39] **A. Brauer**: *On addition chains*, Bulletin American Mathematical Society **45** (October 1939), pp. 736-739.

[Bri90] **E. Brickell**: *A survey of Hardware Implementations of RSA*, Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 368-370.

[Chi84] **H. R. Chivers**: *A practical fast exponentiation algorithm for public key*, International conference on Secure Communication Systems (London, February 1984), pp. 54-58.

[Cos90] **M. J. Coster**: *Some algorithms on addition chains and their complexity*, Report CS-R9024, Centrum voor Wiskunde en Informatica (Amsterdam, June 1990).

[DLS81] **P. Downey**, **B. Leong**, and **R. Sethi**: *Computing sequences with addition chains*, SIAM Journal on Computing **10** (No. 3, August 1981), pp. 638-646.

[Erd60] **P. Erdös**: Remarks on number theory III: On addition chains, Acta Arithmetica **VI** (1960), pp. 77-81 (with erratum for p. 78).

[Knu69] **D. E. Knuth**: *Seminumerical algorithms*, The Art of Computer Programming **2** (1969), section 4.6.3.

[Knu81] **D. E. Knuth**: *Seminumerical algorithms*, The Art of Computer Programming **2** (second edition, 1981), section 4.6.3.

[Mon85] **P. L. Montgomery**: *Modular Multiplication Without Trial Division*, Mathematics of Computation **44** (No. 170, April 1985), pp. 519-521.

[Oli81] **J. Olivos**: *On Vectorial Addition Chains*, Journal of Algorithms **2** (June 1981), pp. 13-21.

[Pem90] **S. Pemberton**: *The ABC programming language*, Prentice-Hall, 1990.

[Pip76] **N. Pippenger**: *On the evaluation of powers and related problems*, Proc. **17**th IEEE Symp. Foundation of Computer Science (Houston, TX, 1976), pp. 258-263.

[Sch37] **A. Scholz**: *Aufgabe 253*, Jahresbericht der Deutschen Mathematiker-Vereinigung **47** (1937), pp. 41-42.

[Str64] **E. G. Straus**: *Solution to problem 5125*, American Mathematical Monthly **71** (September 1964), p. 806-808.

[Thu76] **E. G. Thurber**: *Addition chains and solutions of* $l(2n) = l(n)$ *and* $l(2^n - 1) = n + l(n) - 1$, Discrete Mathematics **16** (1976), pp. 279-289.

[Yac90] **Y. Yacobi**: *Exponentiating faster with addition chains*, Proc. Eurocrypt '90 (Århus, Denmark, May 1990), to appear.

[Yao76] **A. Yao**: *On the evaluation of powers*, SIAM Journal on Computing **5** (No. 1, March 1976), pp. 100-103.

Appendix to Chapter 4: The Programs

This appendix shows a set of ABC programs that can be used to try out the different algorithms discussed in the chapter. Some statistical information on the performance is also supplied.

ABC is a programming language that is designed as a truly simple programming language for beginning programmers, but it turns out to be very useful for trying out algorithms. In a book, it is important that algorithms are denoted clear and precise, and we believe that ABC is the best way to do this. The ABC code is only typographically modified, so the programs are executable as they stand.

The right-to-left binary method

The right-to-left binary method is, like the left-to-right binary method, based on the binary notation of the target number. It uses the same amount of multiplications. In addition graph terminology, the right-to-left binary method is the reversal of the left-to-right binary method. It is an on-the-fly algorithm, because the elements of the addition chain are generated in order.

The algorithm bears its name from the order in which the bits of the exponent are processed. This right-to-left order is often easier to program; for example, the ABC program below is much shorter than the corresponding program for the left-to-right method. On the other hand, the method does use an auxiliary register that the left-to-right method does not need.

The addition chain of the right-to-left binary method for the number n consists of the binary suffixes of n, combined with all powers of two that are smaller than n. For example, the right-to-left addition chain for 13 contains the numbers 1, 2, 4, 5, 8, 13.

An on-the-fly algorithm to compute a power using the right-to-left binary method is

```
how to right2left binary method x power n:        compute power x**n
    put 1, x, n in result, b, e
    while e > 0:                                  result*b**e = x**n
        while e mod 2 = 0:
            put b*b, e / 2 in b, e
        put result*b, e - 1 in result, e
    write result
```

In this algorithm, b is repeatedly squared, corresponding to the powers of two in the addition chain, while e follows the binary suffixes of n, with corresponding powers *result*. We see that one auxiliary register (called b in the program) is used. On most binary computers, the bits of n can be accessed directly without using the auxiliary variable e.

The right-to-left *m*-ary method

The right-to-left binary method can be generalized to other number bases. In principle, the right-to-left *m*-ary method produces the reversed chains of the left-to-right *m*-ary method, so the length of the chain is the same. The method uses $m - 1$ auxiliary registers; this is one more than the left-to-right *m*-ary method. The algorithm is shown below. The computations on the number *e* can be omitted if the computer has instructions to access the *m*-ary digits of a number directly.

```
how to right2left m ary method x power n:        compute power x**n
   put x, n, { } in b, e, mem
   for i in {1...m–1}: put 1 in mem[i]            initialize auxiliary registers
   while e ≥ m:
      if e mod m ≠ 0:
         put mem[e mod m]*b in mem[e mod m]
      put b**m, floor(e / m) in b,e               This takes l(m) multiplications
   put mem[e]*b in mem[e]
   for i in {–m+1...–2}:                          Compute result from intermediates
      put mem[–i–1]*mem[–i] in mem[–i–1]
   for i in {2...m–1}:
      put mem[i]*mem[i–1] in mem[i]
   write mem[m–1]
```

The window method

The algorithm to compute an addition chain with the window method consists of three parts. First, the target is split into windows. Then, an addition sequence has to be generated for the window values, and evaluated to produce the elements corresponding to the window values. When these values are computed and stored, the actual chain can be computed by repeated doubling and adding with the stored values.

We will show algorithms for the three parts separately. The heuristic window distribution is found by the routine "window.split". The window distribution that is computed is based on the heuristic explained before: it is the window distribution with the fewest possible windows that has the smallest product of window values. We admit that this is a very naive heuristic, but it produces reasonable results. The other heuristics we tried were too complex to compute a window distribution in short time.

The algorithm to compute the window distribution is straightforward; the number *n* is scanned from left to right, and the optimal window distribution is computed for every prefix of *n*. In each step, the window distribution is found from the previously computed distributions. In fact, only distributions with exactly one less window are needed to compute an optimal distribution.

```
how to return k window.split n:                    split a number in windows
    put value(n|k) in iw                           n|k = the first k characters of n
    put {[k]: (iw, {[k]: iw})}, {} in prev, opt
    put k in pos                                   "item k" means the k^th element
    while some pos in {pos+1...#n} has n item pos="1":
        if value(n|pos@(max keys prev)+1)>iw:
            put opt, {} in prev, opt               we need another window
        put 0 in m
        for p in keys prev:                        all solutions with one less window
            put prev[p] in val, sol
            put value(n|pos@p+1) in nwv
            if nwv≤iw and (m=0 or val*nwv<m):
                put val*nwv in m                   we found a better solution
                put nwv in sol[pos]
                put m, sol in opt[pos]
    put opt[pos] in val, result                    take optimal solution
    return result
```

The function of the most important variables in "window.split" is:

iw	The value of the initial window.
k	Parameter of the algorithm: size of the first (and largest) window.
opt	Description of window distribution with same amount of windows.
prev	Description of window distributions with one less window. *opt* and *prev* are arrays indexed by the value of *pos*. The contents are pairs of values: the product of the window values, and an array mapping positions to window values.
pos	The position of the bit currently inspected. *opt*[*pos*] is the optimal distribution of bits 1 through *pos* of *n*.
m	The maximum product of window value found so far. If *m*=0, no window has been found yet.
n	Parameter of the function: binary string representing the target.
nwv	Value of the new window.
result	The optimal window distribution: output of the function.

The function "window.split" uses a simple auxiliary function "value", which returns the value of a binary number that is expressed as a string:

```
how to return value s:                             value of a binary string
    put 0 in result
    for d in s:
        put result*2+{["0"]: 0; ["1"]: 1}[d] in result
    return result
```

Computing the addition sequence for the window values is done with the addition sequence algorithm described in the chapter. For "new.values", we use the function "quick.values". The algorithm is repeated here in a slightly different form, so it fits in the main program:

```
how to return make.sequence p:                    make addition sequence
    insert 1 in p
    insert 2 in p                                 initialize protosequence
    put {1; 2} in sequence                        initialize output sequence
    while max p > 2:
        put max p in f                            get next element f
        remove f from p
        insert f in sequence
        put new.number in n                       new.number is a refinement
        if n not.in p: insert n in p              update protosequence
    return sequence                               return the result
new.number:                                       the protosequence function
    put max p in f1
    select:
        f / f1 < 2: return f – f1                 subtract
        f mod 2=0: return f / 2                   double
        some i in {–#p...–1} has (p item (–i))mod 2=1:
            return f – (p item (–i))              f – largest odd element of p
```

The complete computation is performed by the main program, that uses the other two programs:

```
how to window method n size k:                    chain length of window method
    put k window.split n in windows               window values in array
    put { } in values
    for i in windows:
        if i not.in values: insert i in values    convert array to list
    put make.sequence values in part1             initial addition sequence
    put windows[min keys windows] in element      value of first window
    for d in {1+min keys windows...#n}:
        put element*2 in element                  squaring
        if d in keys windows:                     at a window?
            put element+windows[d] in element     add window value
    write "Length of the chain:" /                / gives a new line
    write "Sequence:",#part1–1 /
    write "Doublings:", #n–min keys windows /
    write "Memory:", #windows–1 /
```

In this program, the variable *part1* contains the addition sequence for the window values, and *element* successively gets all the values of the rest of the addition chain. Note that the program does not evaluate the addition chain; it only computes the length.

We investigated the length of the addition chains produced by the window method as a function of the number of ones in the binary representation of the number. The binary method has a simple linear dependency, but the window method gives a curved graph as shown in Figure 14. To make the figure, we generated strings of random bits with different probabilities for zeros and ones, and applied the above algorithm to the resulting numbers. The strings had a length of 512 bits; the first bit was always one. The black dots denote the results of the experiment, while the line gives the length of the addition chain of the binary method.

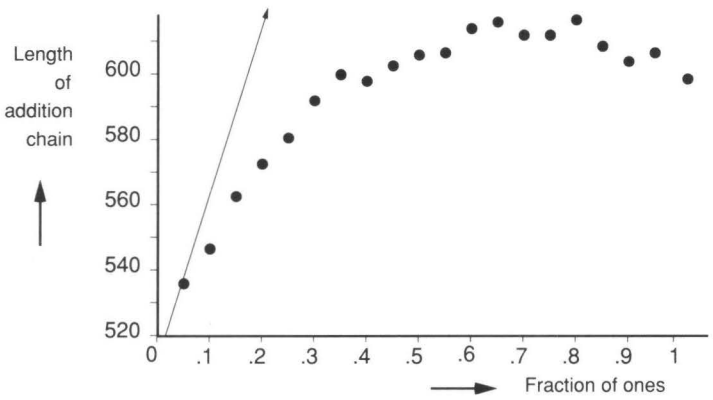


Figure 14: Window method for different numbers of ones.

Addition machine programs for addition sequences

If the addition chain program above is to be used in practice, the addition chain needs to be evaluated. The most interesting part of this is the evaluation of the addition sequence for the window values. If those values are computed, the rest of the computation is simple.

We need a special version of the addition sequence algorithm that can evaluate the addition sequence. To do this, we construct an addition machine program. This program can later be used to evaluate the chain. The addition machine program can be produced in reverse order directly with a modification of the protosequence algorithm.

An example program that produces this addition machine program, using the same protosequence function as before, is shown below.

```
how to addition sequence a:                          produce addition machine program
   init
   while p ≠ {1}:
      put max p in f
      remove f from p                                take f off protosequence
      store                                          produce store if needed
      add                                            produce add if needed
   select:
   keys mem={}: pass
   #keys mem=1:
      write "store", min keys mem                    produce last store
init:                                                initialization
   write "Reverse addition program:" /
   put {} in mem
   for i in {1...#a}:                                initialize registers
      put a item i in mem[i]
   put a in p
   if 1 not.in p: insert 1 in p
   if 2 not.in p: insert 2 in p                      initialize protosequence
store:
   if some i in keys mem has mem[i] = f:             produce store if necessary
      write "store", i /
      delete mem[i]
add:                                                 equivalent to new.number
   put max p in f1
   select:
      f/f1<2:
         put f-f1 in f
         memory
      f mod 2=0:
         put f/2 in f
         write "double" /
      some i in {-#p...-1} has (p item (-i))mod 2=1:
         put (p item (-i)) in n
         memory
         put f - n in n
   if f not.in p: insert f in p
```

```
memory:                                    find memory location for term f
  select:
    some i in keys mem has mem[i]=f:
      write "add", i /
    some i in {1...#p} has i not.in keys mem:
      put f in mem[i]
      write "add", i /
```

This program takes a list of values, and produces an addition machine program that evaluates the corresponding addition sequences. After this program is finished, the values are stored in order in the registers of the addition machine. The addition machine program is produced in reverse order. The program keeps the contents of the addition machine registers in the variable *mem*.

Example:

```
>>>addition sequence {2; 5; 14}
Reverse addition program:
store 3
double
add 1
store 2
add 2
double
store 1
double
store 2
```

In a practical application, the addition program would be stored, and evaluated at a later moment.

Precomputed addition chains

If an exponent is about 512 bits, the minimal number of multiplications produced by the window method depends on the parameter k: the *window size*. The minimum number of multiplications occurs for a window size near 6. Figure 15 shows the minimum, maximum and average lengths of the vector addition chains for 19 random numbers. A window size of 6 does not always generate the minimum addition chain, so if the time is available, it is worth to try several window sizes to see which one gives the best result. On average, this saves about two multiplications.

The addition sequences generated for the window method can also be improved upon. The heuristic techniques of [BC89] produce slightly better addition chains at a relatively high cost. In the graph of Figure 15, this would reduce the number of multiplications for k in the range from 6 to 15, so that the range of values for k that give a good chain is broader. The amount of computation needed for these addition

chains is quite high, while the number of multiplications that can be saved is another one or two. This method is rather expensive, so that it is only worth the trouble if the chain is going to be used very often.

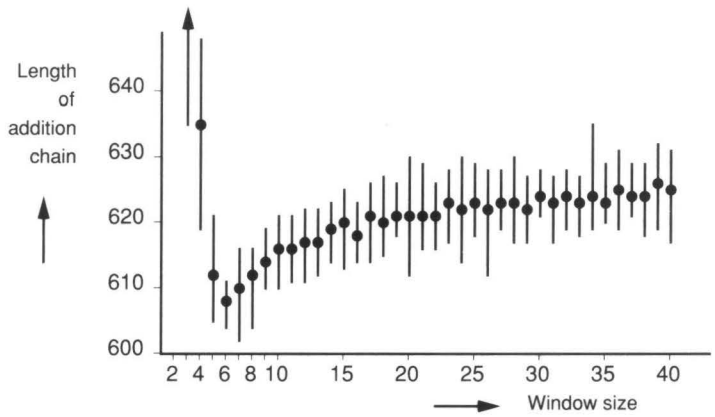


Figure 15: The window method for 512-bit numbers.

The window size that gives a minimum length chain is almost always 6 for numbers of 512 bits. If we take a smaller exponent, we get a different picture; Figure 16 shows the results for random 128- and 256-bits numbers. It is clear that the position of the minimum is much more variable. The reason is that the addition sequence algorithm gets fewer numbers, so that its performance is less predictable.

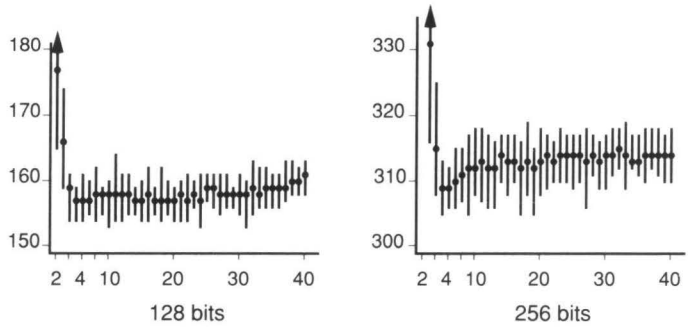


Figure 16: The window method for smaller numbers.

On-the-fly vector addition chains

Vector addition chains can be computed with a simple on-the-fly algorithm. The algorithm consists of three computations:

- generation of the addition sequence;
- reversal of the addition sequence to a vector addition chain;
- evaluation of the vector addition chain.

These computations are performed in parallel, so that almost no extra memory for the computation is needed. We discuss the three processes separately, although the program code contains all three intermingled.

The addition sequence is generated by a special protosequence function that generates sequences that yield efficient memory behaviour when reversed. The protosequence function works like this:

- Compute the quotient of the largest two elements of the protosequence; call them f_2 and f_1.
- If the quotient f_1/f_2 is smaller than 2, produce the difference $f_1 - f_2$.
- If the quotient is larger than 2, use the left-to-right binary method to generate a chain from f_2 to $\lfloor f_2/f_1 \rfloor \cdot f_2$.

The method used in the last case, when the quotient is larger than 2, is designed to make memory usage of the resulting vector addition chain as low as possible. The reversal of the left-to-right binary method used is the right-to-left binary method. This latter method uses minimal memory. Because of this, the algorithm never uses more memory than was needed for the inputs, and the computation can be performed in place. We use a special version of the protosequence function that generates the elements of the chain one by one, to simplify the rest of the algorithm.

There may be addition sequence algorithms that use slightly more memory and give better performance when reversed. Since the quotient f_1/f_2 is almost always small, an algorithm that makes good chains for small numbers should be used.

The distribution of the quotients f_1/f_2 depends on the length of the vectors. The case that the vectors have length one can be considered as a special case (the algorithm performs poorly in this case). If the vectors have length 2, the quotient is smaller than 14 in 90% of the steps [Knu69, section 4.5.3]. If there are more factors, the quotient is even smaller. To make good addition chains for such small numbers, a table of optimal addition chains can be used, together with a simple addition chain method for numbers not in the table. This saves approximately 2% of the multiplications. We computed this figure comparing the binary method and the optimal method for number up to 100. If the vector addition chains are precomputed, the optimal addition chains for those small numbers can be easily computed using the techniques from [Knu69].

The reversal of an addition sequence is a tricky process, although it requires little computation. The produced output of the protosequence algorithm is used to determine the addition graph, and the reverse is produced in the form of a sequence of instructions for an addition machine. The graph is stored in a special data structure. Some optimizations are performed to eliminate unnecessary **store**s and to make sure **double** is used whenever possible.

The final step is the execution of the produced addition machine instructions. As explained before, the instructions are "interpreted multiplicatively" so that the actual product of powers is computed.

The program below computes the product

$$\prod_i b_i^{e_i},$$

where the b_i are stored in *bases*, and the e_i are in *exponents*. The index set for i is the key list of both *bases* and *exponents*.

```
how to vector chain bases power exponents mod n:
   check keys bases=keys exponents          index sets must be equals
   init                                     initialize algorithm
   while #ref > 1:
      generate                              first step: addition sequence
      reverse                               second step: reversing
      store                                 third step: evaluation
   put max keys ref in f
   reverse                                  final multiplications
   write "Result:", acc                     result of the computation
init:                                       initialize ref, memory, mem.list
   put bases, {[1]: {}}, {} in memory, ref, mem.list
   for i in keys bases:
      if exponents[i] not.in keys ref: put {} in ref[exponents[i]]
      insert i in ref[exponents[i]]
      insert i in mem.list
   put max keys exponents in m
   write "load", m /                        load initial number
   put memory[m], m in acc, stored
   remove m from ref[exponents[max keys exponents]]
   remove m from mem.list                   initial multiplication not necessary
generate:                                   make the addition sequence
   put (keys ref) item #ref, (keys ref) item (#ref–1) in f, f2
   put floor(f / f2),f2 in q, d             (most of the time, q is 1)
   while q mod 2=0: put q/2, d*2 in q, d
   if d = f: put d/2 in d
   if d not.in keys ref: put {} in ref[d]
   if f–d not.in keys ref: put {} in ref[f–d]   update ref for d and f
   if d > f–d: put f–d in d                 make sure d is smallest
```

```
reverse:                                        ref does the reversing
  if f in keys ref and stored in ref[f]:        produce double instead of add
    remove stored from ref[f]
    remove stored from mem.list
    write "double" /
    put (acc*acc)mod n, 0 in acc, stored
  for m in ref[f]:                              produce all multiplications
    select:
      m>0:
        write "add", m /
        put (acc*memory[m])mod n, 0 in acc, stored
        remove m from mem.list
  delete ref[f]
store:                                          store number
  select:
    f = 2*d:                                    use double instead of store and
      add
      write "double" /
      put (acc*acc)mod n, 0 in acc, stored
    stored>0:                                   value is already stored
      insert stored in ref[d]
      insert stored in mem.list
    else:                                       find empty memory location
      select:
        some m in {1...1+#mem.list} has m not.in mem.list:
          put m in stored
          write "store", m /
          put acc in memory[m]
      insert stored in ref[d]
      insert stored in mem.list
```

For example, to compute $2^5 \cdot 3^7$ mod 101, one would type:

```
>>>vector chain {[1]: 2; [2]:3} power {[1]: 5; [2]: 7} mod 101
```

to get the addition machine program and result:

```
load 2
add 1
store 1
double
add 2
double
add 1
Result: 92
```

The variables used in the program have the following meaning:

acc	accumulator of the addition machine
bases	table of the numbers b_i (input parameter)
d	is the place where a * instruction is generated
exponents	table of the numbers e_i (input parameter)
mem.list	contains used memory locations (with multiplicity)
memory	stores intermediate results
n	modulus for the multiplications (input parameter)
ref	contains the edges of the addition graph
ref[*l*]	is the list of numbers that have to be multiplied in when the sequence reaches *l*
keys *ref*	is the protosequence (partial addition sequence)
stored	keeps track of the contents of the accumulator: number: memory location; 0:a product not in memory

The on-the-fly vector addition chain algorithm gives no savings if there is only one factor in the product; in fact, it is not more efficient than the binary method. The savings for 2 or more factors are probably always worth the trouble of the algorithm. Table 8 gives the average lengths of vector addition chains. The top row gives the number of factors in the addition chain, and the first column gives the number of bits of the factors. Each entry is the average of fifteen chains for random numbers. The bottom row gives the number of multiplications that a product of 500-bit numbers would take if the binary method was used together with multiplication of the factors. As can be seen, the savings of using a vector addition chain are enormous, even for only two factors.

factors	1	2	5	10	20	50
20 bits	30	32	47	69	114	227
50 bits	73	81	110	169	263	468
100 bits	148	161	221	330	471	737
200 bits	299	321	440	650	988	1615
500 bits	743	803	1093	1627	2573	4778
1000 bits	1493	1608	2183	3245	5192	9973
multiply	748	1498	3746	7494	14989	37474

Table 8: On-the-fly vector addition chains.

The percentage of doublings in a generated vector addition chain decreases if the number of factors increases. Table 9 gives the percentage of doublings with respect to the length of the vector addition chain for the experiments that yielded Table 8. We see that the percentage of multiplications decreases to a very small portion of the multiplications if the number of factors is large. This is not a surprise, but it means that

for a large number of factors, substitution of **double** whenever possible is not effective anymore; it is probably cheaper to always use a multiplication or detect squarings at multiplication time.

factors	1	2	5	10	20	50
20 bits	64.2%	44.0%	18.5%	7.7%	3.0%	1.8%
50 bits	66.8%	44.6%	17.4%	4.8%	2.0%	1.9%
100 bits	66.8%	44.6%	15.7%	4.4%	2.9%	2.4%
200 bits	66.5%	45.8%	15.7%	4.5%	1.8%	1.3%
500 bits	67.1%	45.3%	16.3%	4.0%	0.8%	0.4%
1000 bits	66.9%	45.5%	15.9%	4.0%	0.6%	0.2%

Table 9: Doublings in vector addition chains.

5

Provably Unforgeable Signatures

Introduction

One of the greatest achievements of modern cryptography is the digital signature. A *digital signature on a message* is a special encryption of the message that can easily be verified by third parties. Signatures cannot be denied by the signer nor falsified by other parties.

There are several attempts in the literature to make an efficient provably secure signature scheme. With "secure", we mean that it is hard for unauthorized parties to make a false signature. The strongest sense of security is defined in [GMR88], and a scheme is described that provides this level of security under the factoring assumption. Our new signature scheme gives the same level of security much more efficiently—in fact, it is about as efficient as the RSA scheme. Parameter values can be chosen to suit special needs. In the most efficient case, a signature on a short message (64 bits) can be signed in 33 modular multiplications (not counting precomputation) and verified in 35 multiplications. The scheme is based on the modular root (RSA) assumption.

After the introduction, we discuss other signature schemes relevant to this work. We discuss the Lamport signature scheme, on which this signature scheme is based, in detail. Then, the new scheme is explained, and the possible choices for parameter values are shown.

Signature scheme

An overview of signature schemes, comparing securities, can be found in the paper mentioned earlier [GMR88]. We use their notation. They define a signature scheme as consisting of the following components:

- A *security parameter* k, that defines the security of the system, and that may also influence performance figures such as the length of signatures, running times and so on.
- A *message space* $\mathbf{M}$, that defines on which messages the signature algorithm may be applied.

- A *signature bound b*, that defines the maximal number of signatures that can be generated without reinitialization. Typically, this value depends on k, but it can be infinite.
- A *key generation algorithm* **G**, that allows a user to generate a pair of corresponding public and secret keys for signing. The secret key S is used for generating a signature, while the public key P is used to verify the signature.
- A *signature algorithm* σ, that produces a signature, given the secret key and the message to be signed.
- Finally, a *verification algorithm*, that produces **true** or **false** on input of a signature and a public key. It ouputs **true** if and only if the signature is valid for the particular public key.

Some of these algorithms may be *randomized*, which means that they may use random numbers. Of course, **G** must be randomized, because different users must produce different signatures. The signing algorithm σ is sometimes randomized, but this tends to produce larger signatures. The verification algorithm is usually not randomized.

A simple example of a signature scheme is a trapdoor one-way function f. The function f is used for verification by comparing the function value of the signature with the message to be signed, and σ is the trapdoor of f. The main problem with such a scheme is that random messages $f(x)$ can be signed by taking a random signature value x. A simple solution is to let **M** be a sparse subset of a larger space, so that the probability that $f(x)$ is a valid message for random x is low. An example of a sparse subset is the set of "meaningful" messages.

Related work

The notion "digital signature" was introduced in [DH76]. This paper, which can be considered the foundation of modern cryptography, discusses the possibility of digital signatures and the use of a trapdoor one-way function to make them.

[RSA78] is the original article on the RSA scheme. It introduces the famous RSA trapdoor one-way function. This function is still widely in use and is applied frequently. A well-known weakness of RSA is that it is multiplicative: the product of two signatures is the signature of the product. This potential problem can be prevented as above by choosing an appropriate sparse message space.

Since then, an enormous number of signature schemes have been proposed [Rab77, MH78, Sha78, Rab79, Lie81, DLM82, GMY83, Den84, GMR84, OSS84, ElG85, OS85, FS86, BM88, GMR88, CA89, EGL89, EGM89, Mer89, Sch89, SQV89, BCDP90, Cha90, CR90, Hay90, CHP91], applied [Wil80, Cha82, Gol86, Bet88], and broken [Yuv79, Sha82, Tu84, BD85, EAKMM85, Roo91]. We will not discuss all these schemes here; we only discuss the ones that are interesting to compare with the new scheme.

The schemes [Rab79, GMY83, GMR84, GMR88] are steps towards a provably secure signature scheme. The scheme described in the last article is secure in a very

strong way: it is "existentially unforgeable under an adaptive chosen-message attack" with probability smaller than $1/Q(k)$ for every polynomial Q. This means that generating a new signature is polynomially hard if signatures on old messages are known, even if the old signatures are on messages chosen by the attacker.

The scheme in [GMR88] is based on factoring. While our scheme is based on the slightly stronger RSA assumption, it is much more efficient. The signature scheme of [GMR88] uses a large amount of memory for the signer, and quite a lot of computation. Our scheme uses no memory at all, except for a counter and the public values, and signing and verifying takes about as much computation as RSA does, depending on the parameters.

The Lamport Scheme

To explain the new system, we compare it to the earlier *Lamport scheme* (explained already in [DH76, page 650]). To make a signature in this scheme, the signer makes a secret list of $2k$ random numbers

$$\mathbf{A} = a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots, a_{k,0}, a_{k,1},$$

applies a one-way function f to all elements, and publishes the result **B**:

$$\mathbf{B} = \begin{cases} f(a_{1,0}), f(a_{2,0}), \ldots, f(a_{k,0}) \\ f(a_{1,1}), f(a_{2,1}), \ldots, f(a_{k,1}) \end{cases}$$

The signature consists of the numbers $a_{1,m_1}, a_{2,m_2}, \ldots, a_{k,m_k}$ from the list **A** (one from each "column"),where m_1, m_2,..., m_k are the bits of the message to be signed. The lists **A** and **B** cannot be used again.

The properties of Lamport's scheme are easy to verify:

- Signing a message is only the publication of the proper elements of **A**.
- To forge a signature, one needs to find certain values from the list **A**. How hard this is, depends on the security of the one-way function f.
- If the values **A** are only used for one signature, new signatures cannot be made from old ones.
- Verification of a signature consists of applying the one-way function to the signature values, and comparing them to the public values determined by the signed message.

The new system uses the same idea, with three important differences. First, the list **B** is replaced by another list that can be used for all signatures. Second, the list **A** is constructed from two lists so that less memory is needed to define it. Third, the elements of **A** in the signature can be combined into a single number.

A small optimization

There is a trivial optimization of Lamport's scheme that reduces the number of public function values to almost half, that we could not find in the literature. This optimization is independent of the signature scheme as such. Basically, the signer signs by publishing a k-element subset of the $2k$ secret numbers. Lamport's scheme

chooses a particular set of subsets of the set of $2k$ elements, as shown above. The necessary property of this set of subsets is that no subset includes another.

There are other sets of subsets with the property that no subsets includes another. A largest set of subsets with this property is the set of all k-element subsets (a well-known result from lattice theory). For these sets, it is easy to see that no subset includes another.

For example, in Lamport's scheme, the list of 6 elements

$$\mathbf{A} = a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, a_{3,0}, a_{3,1}$$

allows us to sign messages of 3 bits. If we renumber **A** as $a_1, a_2, a_3, a_4, a_5, a_6$, we get the set of 20 three-element subsets of **A**:

$$\{a_1,a_2,a_3\}, \{a_1,a_2,a_4\}, \{a_1,a_2,a_5\}, \{a_1,a_2,a_6\}, \{a_1,a_3,a_4\},$$
$$\{a_1,a_3,a_5\}, \{a_1,a_3,a_6\}, \{a_1,a_4,a_5\}, \{a_1,a_4,a_6\}, \{a_1,a_5,a_6\},$$
$$\{a_2,a_3,a_4\}, \{a_2,a_3,a_5\}, \{a_2,a_3,a_6\}, \{a_2,a_4,a_5\}, \{a_2,a_4,a_6\},$$
$$\{a_2,a_5,a_6\}, \{a_3,a_4,a_5\}, \{a_3,a_4,a_6\}, \{a_3,a_5,a_6\}, \{a_4,a_5,a_6\};$$

this allows us to sign one of 20 messages, which is equivalent to more than 4 bits.

In general, there are

$$\binom{2k}{k}, \text{ or about } \frac{2^{2k}}{\sqrt{k\pi}},$$

k-element subsets, so that we can sign messages of about $2k - \frac{1}{2}\log_2(k\pi)$ bits. The original Lamport scheme allowed messages of only k bits, so that we get almost a doubling of the message size for the same size of the list **B**. This simple improvement can also be used in our new signature scheme.

To encode a signature, a mapping needs to be defined between messages and these subsets:

$$s(message) = subset.$$

The simplest mapping just enumerates messages (interpreted as numbers from 0 onwards) to sets (seen as binary strings that denote 1 for presence and 0 for absence) in order. Such a mapping is easily and efficiently computed by the algorithm shown in Figure 17. The binomial coefficients do not need to be computed by repeated multiplication and division. The first binomial coefficient is always the same, so it can be precomputed, and the others can be computed by one multiplication and one division by small numbers using the properties:

$$\binom{t-1}{e} = \binom{t}{e} \cdot \frac{t-e}{t} \text{ and } \binom{t-1}{e-1} = \binom{t}{e} \cdot \frac{e}{t}.$$

The algorithm outputs ones and zeros corresponding to the elements in the resulting set.

Note that the Lamport scheme uses another mapping that maps numbers onto k-element subsets, but that only a small number of these sets are used.

Let n, the message, be a number in the range $0 \ldots \binom{2k}{k} - 1$.

Put $2k$ in t and k in e.

While $t > 0$:

Put $t - 1$ in t.

If $n \geq \binom{t}{e}$, put $n - \binom{t}{e}$ in n, $e - 1$ in e, and output a 1 (this t is in the set).

Else, output a 0 (this t is not in the set).

Figure 17: Algorithm for the mapping s.

The New Signature Scheme

The new signature scheme replaces the list **A** of the Lamport scheme by a list of numbers that can be organized in a matrix. Instead of using a new list **B** for every signature, a fixed list called **R** is used for all signatures and all participants. The one-way function f is replaced by a set of trapdoor one-way functions, that changes per signature. For the trapdoor one-way functions, we use the modular root function of [RSA78].

The construction allows us to sign long messages using only a few numbers to define the set **A**. In the example of Figure 18, the set **A** of 12 elements is constructed from three primes p_1, p_2, p_3 (used only for this signature) and four public values r_1, r_2, r_3, r_4 (that can be used again). This set allows us to sign messages of 9 bits, since there are $924 > 2^9$ possible 6-element subsets of **A**. Signing messages of 9 bits in the original Lamport scheme takes 18 public values that can be used only once.

$$\begin{array}{cccc} \sqrt[p_1]{r_1} & \sqrt[p_1]{r_2} & \sqrt[p_1]{r_3} & \sqrt[p_1]{r_4} \\ \sqrt[p_2]{r_1} & \sqrt[p_2]{r_2} & \sqrt[p_2]{r_3} & \sqrt[p_2]{r_4} \\ \sqrt[p_3]{r_1} & \sqrt[p_3]{r_2} & \sqrt[p_3]{r_3} & \sqrt[p_3]{r_4} \end{array}$$

Figure 18: Example list **A** of the new scheme.

The numbers a_i of **A** are secret encryptions of the numbers r_i of **R**, and the corresponding decryption exponents are public. The multiplicative property of RSA allows us to multiply the values of the signature to form one number. Verification of a signature can be done using a simple computation, without having to compute the separate factors.

The public values of the new system are:

- One modulus per signer;
- The system-wide list **R**. This list is used by all users, and that it does not change often, so that distribution does not require much traffic. The numbers in **R** are smaller than the smallest modulus used by the signers.
- A list of sets of primes that may be used for signing. For security reasons, the sets may not overlap each other, and the signers may only use these sets of primes.

A signature consists of the original message signed, the signature proper (an integer smaller than the modulus of the signer), and a description of the prime set.

In the language of [GMR88]:

- The security parameter determines the size of the RSA modulus. This modulus can vary per user.
- The message space **M** is (equivalent to) the set of subsets of **A** that include half the elements.
- The size of the public list of sets of primes determines the signature bound b.
- Key generation is a matter of generating an RSA modulus, and computing exponents for the modular root extractions.
- Signing and verification are defined below.

Signing

For the list **A** of a signature, the set of RSA encryptions

$$\mathbf{A} = \left\{ \sqrt[p]{r} \bmod n \,\middle|\, p \in \mathbf{P}; r \in \mathbf{R} \right\}$$

is used, where:

- **P** a set of primes from the public list;
- **R** is the public list of verification values;
- n is the RSA modulus of the signer.

As explained above, a signature is constructed from a subset determined by $s(m)$ of half these numbers. The constant k used in the algorithm that maps s is equal to $\left\lfloor \frac{\#\mathbf{P} \cdot \#\mathbf{R}}{2} \right\rfloor$. This allows us to sign a message of almost $\#\mathbf{A} = \#\mathbf{P} \cdot \#\mathbf{R}$ bits. The product of the elements of **A** in this subset is the signature. Since this is a single number, the signature is much more compact than in Lamport's scheme.

Thus, signing a message consist of the following steps:

- Choose the set **P** of primes that is to be used for this signing from the public list. This determines **A**:
$$\mathbf{A} = \left\{ \sqrt[p_i]{r_j} \bmod n \,\middle|\, i,j \in \{1,\ldots,\#\mathbf{P}\} \times \{1,\ldots\#\mathbf{R}\} \right\}.$$
Like the sets **A** and **B** in Lamport's scheme, the set **P** can be used only once. The list **A** need not be computed.
- Determine the message m to sign. This could be a message, or a public hash function value of that message, for example.
- Compute the subset **M** of index pairs from $\{1,\ldots,\#\mathbf{P}\}\times\{1,\ldots,\#\mathbf{R}\}$ from the message m with the algorithm described above:
$$\mathbf{M} = s(m)$$
- Compute the signature proper:
$$S = \prod_{i,j\in\mathbf{M}} \sqrt[p_i]{r_j} \pmod n,$$
and send m, **P**, and S to the recipient.

There are two ways to increase the efficiency of signing. If there is time to do a precomputation, the entire set **A** can be computed before the value of m is known. Although this takes quite a while, signing becomes much faster, since signing consists only of multiplying the proper values of **A** together. If precomputation is not possible, the computation of S can be speeded up with a *vector addition chain* (see the previous chapter).

Verification

Instead of trying to compute individual factors of the signature, the number S can be verified in a single computation. To see this, we note that the power of the signature
$$S^{\prod_{k\in\mathbf{P}} p_k}.$$
should be equal to the following product that can be computed from public values:
$$\prod_{i,j\in\mathbf{M}} r_i^{\prod_{k\in\mathbf{P}} p_k / p_j}.$$
The lower product can be computed with a vector addition chain. Verification of a signature consists of checking that these two values are the same. The verification can be performed with a single vector addition chain, if the inverse of the signature is computed first:
$$(S^{-1})^{\prod_{k\in\mathbf{P}} p_k} \cdot \prod_{i,j\in\mathbf{M}} r_i^{\prod_{k\in\mathbf{P}} p_k / p_j},$$
which must evaluate to 1 (mod n). To increase the efficiency of the verification, the signer could send $1/S$ instead of S, so that the inversion is performed only once by the signer, and not by every verifier.

If not all prime numbers from **P** occur as exponents in the set **M**, it is possible to verify a signature using slightly fewer multiplications by raising S to only the occurring

primes. Unfortunately, this optimization is only applicable in the less interesting cases where verification requires a lot of multiplications.

The verifier must also check whether **P** occurs in the public list. If **P** is described as an index number in this list, this is of course unnecessary.

Parameters

In practice, the following parameter values could be used:

- A modulus size big enough to make factorization hard (200 digits, or 668 bits).
- **R** a list of 50 numbers.
- The sets **P** consisting of the $(5n+1)^{\text{th}}$ to the $(5n+5)^{\text{th}}$ odd prime number, where $n \in \{0,\ldots,16404\}$ is the sequence number of the signature. This uses the primes of up to 20 bits.

With these parameters, we have sets **A** of 250 elements, so that a message of 245 bits (30 bytes) can be signed. A signature consists of the message, the signature product (668 bits, or 84 bytes), and the index number of the prime set (15 bits, or 2 bytes). Computing a signature takes about 1512 modular multiplications, and verification about 272; both these numbers are obtained using vector addition chains.

The list of the odd primes up to 20 bits (the highest being 1048557) can easily be stored; it would need only 64 K bytes of storage (using a bit table of the odd numbers) and contain 82025 primes. Such a list can easily be stored in a ROM chip. When all primes are used up, the user can choose a new modulus and start again. Another solution is to change the list **R** often enough so that users do not run out of primes. To make it possible to verify old signatures, old values of **R** and the user moduli must be saved.

The list **R** can be computed from a seed number using a public hash function. This way, only one seed number is needed to define **R**. This allows us one to use a long list **R** while using small amounts of data to distribute it. Also, less data is needed to save old lists.

Table 10 shows the performance of the algorithm for several sizes of **R** and **P**. For each of the entries in the table, the modulus is 668 bits (200 decimal digits), and the size of the primes in **P** is 20 bits. The entries are computed by averaging random number approximations. The entries marked by * have an estimated standard deviation higher than 10, so that the last digits are likely to be inaccurate.

Powers and products were computed using addition chains and sequences; see the previous chapter. The products were computed collecting the base numbers; for example, the product

$$b_2^{e_1} \cdot b_3^{e_1} \cdot b_1^{e_2} \cdot b_3^{e_2} \cdot b_4^{e_2} \cdot b_2^{e_3}$$

would be computed as

$$b_1^{e_2} \cdot b_2^{e_1+e_3} \cdot b_3^{e_1+e_2} \cdot b_4^{e_2}$$

using a vector addition chain algorithm. In the cases were a single power was to be computed, the "window method" of Chapter 4 was applied.

The table shows that in the general case, where verification is done more often than signing, it is advantageous to use a small **P**, possibly of only one element. The length of the list **R** is not a problem if it is generated from a seed, as suggested above. Another advantage of using a small set **P** is that the list **R** has to change less often.

#**R**	#**P**	message	sign	verify
250	1	245	910	152
50	5	245	1512	272
5	50	245	1451	2048*
1	250	245	796	7123*
500	1	495	1035	278
50	10	495	2964*	1372*
68	1	64	819	61
17	4	64	1317	162
4	17	64	1301	659*

Table 10: Performance for different size of **R** and **P**.

The influence of the modulus size and prime size on the performance is shown is Table 11. In this table, the size of **R** is set to 50 elements, while the sets **P** contain 5 elements each. The number of multiplications for signing depends on the size of the modulus only, while the number of multiplications for verifying depends on the size of the prime numbers only. Although it saves a little time during the signing to use a shorter modulus, we suggest using a modulus of 668 bits, since the current technology already allows factoring numbers of up to 351 bits.

The size of the primes in the sets **P** determines the verification time. Choosing smaller primes increases the speed of verification, but allows fewer signatures before a new list **R** is needed.

modulus size	signing
512	1172
668	1512

prime size	verifying
10	171
20	272
30	381

Table 11: Performance for different sizes of modulus and primes.

If the elements of **A** are precomputed, signing takes #**A**/2–1 multiplications. The precomputation takes about 796·#**A** multiplications, so precomputation is only effective if there is plenty of time for doing it.

For extremely fast verification of signatures, we choose a list **R** of 68 elements, generated from a seed number that is part of the signature, and **P** ={3}. For these parameters, the message to be signed is 64 bits (8 bytes). This allows verification of a signature in only 35 modular multiplications, plus the time to generate the elements of

R. Signing takes about 819 multiplications. Using precomputation, signing takes 33 multiplications, but about 55000 multiplications for the precomputation.

Proof of unforgeability

We prove that the signature scheme is "existentially unforgeable under an adaptive chosen-message attack". This means that, under the RSA assumption, if an attacker can influence the signer to sign any number of messages of his liking, he cannot forge new signatures in polynomial time, even if the messages depend on the signatures on earlier messages.

The main theorem used to prove unforgeability of the signature system is proved by Jan-Hendrik Evertse and Eugène van Heijst in [EH90], and is a generalization of a theorem by Adi Shamir [Sha83]. The theorem is about computing a product of RSA roots with a given modulus if a set of products of signatures is known. Under the RSA assumption, the theorem states that if a set of products of roots is known, the only new products of roots that can be constructed in polynomial time are those that can be computed using multiplication and division.

One assumption we make is that the attacker cannot combine the signatures of different participants, because they have different moduli. This is still an open problem. This assumption allows us to use the results of [EH90].

In our situation, we assume an attacker who knows many signature products S from a participant. These products can be written as products of roots of elements of **R**:

$$r_1^{x_1} r_2^{x_2} r_3^{x_3} \cdots r_{\#\mathbf{R}}^{x_{\#\mathbf{R}}},$$

where the numbers x_i are rational numbers. The theorem of [EH90] states that if we interpret the x as vectors, the only new products that can be computed by the attacker correspond to linear combinations of these vectors. What remains to be proved is that linear combinations of these vectors do not give products that the attacker can use for new signatures.

The denominators of the rational numbers x_i are products of primes from the set **P** of the corresponding signature, since the x_i are sums of the form $\frac{1}{p_1} + \frac{1}{p_2} + \cdots$, where $p_i \in \mathbf{P}$. This means that we can speak of "the set of primes in a vector", meaning both the set of primes that occur in the denominators of the elements, and the set **P** used for generating the signature. Every signature uses another **P**, and the sets **P** do not overlap, so the sets of primes in the vectors also do not overlap. A linear combination of vectors will contain only primes that occurred in the original vectors. From this we see that combining signatures with multiplication and division will not produce a signature with a set **P** that is not used before.

For a set **P** that has already been used, the only linear combination of vectors that contains the primes of **P** is a multiple of the corresponding vector, because any other linear combination of vectors contains primes not in **P** . This means that other signature products do not help compute a new signature product with a given set **P**. From the definition of the signature product, we see that a power of a product cannot be a

signature on another message, so this method also yields no new signatures for the attacker.

Note that if m is a one-way hash function of a message, signatures on other messages can be forged if the hash function is broken. This is of course a separate problem from the security of the signature scheme.

From the above we conclude that an attacker cannot, under the RSA assumption, produce a signature product that is not already computed by the signer. This finishes the proof that the signature scheme is secure.

Conclusion

It was already known that a signature with provable unforgeability existed under the factoring assumption. Our scheme, based on the modular root assumption, improves on the scheme in the literature on several points: signatures are smaller, while signing and verification use much less memory and computation. The new scheme has a large degree of flexibility, allowing the signing of both long and short messages by varying the parameters.

References

[BCDP90] **J. F. Boyar**, **D. Chaum**, **I. B. Damgård** and **T. Pedersen**: *Convertible Undeniable Signatures*, Advances in Cryptology: Proc. Crypto '90 (Santa Barbara, CA, August 1990), to be published.

[BD85] **E. F. Brickell** and **J. M. DeLaurentis**: *An Attack on a Signature Scheme proposed by Okamoto and Shiraishi*, Advances in Cryptology: Proc. Crypto '85 (Santa Barbara, CA, August 1985), pp. 28-32.

[Bet88] **T. Beth**: *A Fiat-Shamir-like Authentication Protocol for the ElGamal Scheme*, Advances in Cryptology: Proc. Eurocrypt '88 (Davos, Switzerland, May 1988), pp. 77-86.

[BM88] **M. Bellare** and **S. Micali**: *How to Sign Given any Trapdoor Function*, Advances in Cryptology: Proc. Crypto '88 (Santa Barbara, CA, August 1988), pp. 200-215.

[CA89] **D. Chaum** and **H. van Antwerpen**: *Undeniable Signatures*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 212-216.

[Cha82] **D. Chaum**: *Blind Signatures for Untraceable Payments*, Advances in Cryptology: Proc. Crypto '82 (Santa Barbara, CA, August 1982), pp. 199-203.

[Cha90] **D. Chaum**: *Zero-knowledge Undeniable Signatures*, Advances in Cryptology: Proc. Eurocrypt '90 (Århus, Denmark, May 1990), pp. 458-464.

[CHP91] **D. Chaum**, **E. van Heijst**, and **B. Pfitzmann**: *Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer*, Advances of Cryptology: Proc. Crypto '91 (Santa Barbara, August 1991), to be published.

[CR90] **D. Chaum** and **S. Roijakkers**: *Unconditionally Secure Digital Signatures*, Advances in Cryptology: Proc. Crypto '90 (Santa Barbara, CA, August 1990), pp. 209-217.

[Den84] **D. E. R. Denning**: *Digital Signatures with RSA and Other Public-Key Cryptosystems*, Comm. ACM **27** (No. 4, April 1984), pp. 388-392.

[DH76] **W. Diffie** and **M. E. Hellman**: *New Directions in Cryptography*, IEEE Trans. Information Theory **IT-22** (No. 6, November 1976), pp. 644-654.

[DLM82] **R. DeMillo**, **N. Lynch**, and **M. Merritt**: *Cryptographic Protocols*, Proc. **14**th ACM Symp. Theory of Computing (San Fransisco, CA, May 1982), pp. 383-400.

[EAKMM85] **D. Estes**, **L. M. Adleman**, **K. Kompella**, **K. McCurley**, and **G. L. Miller**: *Breaking the Ong-Schnorr-Shamir Signature Scheme for Quadratic Number Fields*, Advances in Cryptology: Proc. Crypto '85 (Santa Barbara, CA, August 1985), pp. 3-13.

[EGL89] **S. Even**, **O. Goldreich**, and **A. Lempel**: *A Randomized Protocol for Signing Contracts*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 205-210.

[EGM89] **S. Even**, **O. Goldreich**, and **S. Micali**: *On-line/Off-line Digital Signatures*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 263-275

[EH90] **J-H. Evertse** and **E. van Heyst**: *Which RSA Signatures can be Computed from Some Given Signatures?*, Advances in Cryptology: Proc. Eurocrypt '90 (Århus, Denmark, May 1990), pp. 83-97.

[EH91] **J-H. Evertse** and **E. van Heyst**: *Which RSA Signatures can be Computed from Certain Given Signatures?*, Report W 91-06, February 1991, Mathematical Institute, University of Leiden.

[ElG85] **T. ElGamal**: *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithm*, IEEE Trans. Information Theory **IT-31** (No. 4, July 1985), pp. 469-472.

[FS86] **A. Fiat** and **A. Shamir**: *How to Prove Yourself: Practical Solutions of Identification and Signature Problems*, Advances in Cryptology: Proc. Crypto '86, (Santa Barbara, CA, August 1986), pp. 186-194.

[GMR84] **S. Goldwasser, S. Micali**, and **R. L. Rivest**: *A "Paradoxical" Solution to the Signature Problem*, Proc. **25**th IEEE Symp. Foundations of Computer Science (Singer Island, 1984), pp. 441-448.

[GMR88] **S. Goldwasser, S. Micali**, and **R. L. Rivest**: *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks*, SIAM Journal on Computing **17** (No 2, April 1988), pp. 281-308.

[GMY83] **S. Goldwasser, S. Micali**, and **A. Yao**: *Strong Signature Schemes*, Proc. **15**th ACM Symp. Theory of Computing (Boston, MA, April 1983), pp. 431-439.

[Gol86] **O. Goldreich**: *Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme*, Advances in Cryptology: Proc. Crypto '86 (Santa Barbara, CA, August 1986), pp. 104-110.

[Gol86a] **O. Goldreich**: *Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme*, Report MIT/LCS/TM-315, Massachusetts Institute of Technology.

[Hay90] **B. Hayes**: *Anonymous One-Time Signatures and Flexible Untraceable Electronic Cash*, Advances in Cryptology: Proc. Auscrypt '90 (Sydney, Australia, January 1990), pp. 294-305.

[Lie81] **K. Lieberherr**: *Uniform Complexity and Digital Signatures*, Theoretical Computer Science **16** (1981), pp. 99-110.

[Mau91] **U. Maurer**: *Non-interactive Public Key Cryptography*, Advances in Cryptology: Proc. Eurocrypt '91 (Brighton, United Kingdom, April 1991), to be published.

[Mer89] **R. C. Merkle**: *A Certified Digital Signature*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 218-238.

[MH78] **R. C. Merkle** and **M. E. Hellman**: *Hiding Information and Signatures in Trapdoor Knapsacks*, IEEE Trans. Information Theory **IT-24** (No. 5, September 1987), pp. 525-530.

[Oka88] **T. Okamoto**: *A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems*, ACM Trans. Computer Systems **6** (No. 8, November 1988), pp. 342-441.

[OS85] **T. Okamoto** and **A. Shiraishi**: *A Fast Signature Scheme Based on Quadratic Inequalities*, Proc. 1985 Symp. Security and Privacy (Oakland, CA, April 1985), pp. 123-132.

[OSS84] **H. Ong**, **C. P. Schnorr**, and **A. Shamir**: *Efficient Signature Schemes based on Polynomial Equations*, Advances in Cryptology: Proc. Crypto '84 (Santa Barbara, August 1984), pp. 37-46.

[Rab77] **M. O. Rabin**: *Digitalized Signatures*, Foundations of Secure Computations 1977 (Atlanta, GA, October 1977), pp. 155-168.

[Rab79] **M. O. Rabin**: *Digitalized Signatures and Public-key Function as Intractable as Factorization*, Report MIT/LCS/TR-212, Massachusetts Institute of Technology.

[Roo91] **P. J. N. de Rooij**: *On the security of the Schnorr Scheme using Preprocessing*, Proc. Eurocrypt '91 (Brighton, United Kingdom), to be published.

[RSA78] **R. L. Rivest**, **A. Shamir**, and **M. Adleman**: *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Comm. ACM **21** (No 2, February 1978), pp. 120-126.

[Sch89] **C. P. Schnorr**: *Efficient Identification and Signatures for Smart Cards*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 239-251.

[Sha78] **A. Shamir**: *A Fast Signature Scheme*, Report MIT/LCS/TR-107, Massachusetts Institute of Technology.

[Sha82] **A. Shamir**: *A polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem*, Proc. **23**rd IEEE Symp. Foundations of Computer Science (Chicago, IL, 1982), pp. 145-152.

[Sha83] **A. Shamir**: *On the Generation of Cryptographically Strong Pseudorandom Sequences*, ACM Trans. Computer Systems **1** (No. 1, February 1983), pp. 38-44.

[Sha84] **A. Shamir**: *Identity-based Cryptosystems and Signature Schemes*, Advances in Cryptology: Proc. Crypto '84 (Santa Barbara, CA, August 1984), pp. 47-53.

[SQV89] **M. de Soete**, **J.-J. Quisquater**, and **K. Vledder**: *A Signature with Shared Verification Scheme*, Advances in Cryptology: Proc. Crypto '89 (Santa Barbara, CA, August 1989), pp. 253-262.

[Tu84] **Y. Tulpan**: *Fast Cryptoanalysis of a Fast Signature System*, Master's thesis in Applied Mathematics, Weizmann Institute, Israel, 1984.

[Wil80] **H. C. Williams**, *A Modification of the RSA Public-Key Encryption Procedure*, IEEE Trans. Information Theory **IT-26**, (No. 6, November 1980), pp. 726-729.

[Yuv79] **G. Yuval**: *How to Swindle Rabin*, Cryptologia **3** (No. 3, July 1979), pp. 187-189.

6

Verification of RSA Computations on a Small Computer

Introduction

In many cryptographic applications, a protocol has to be performed between two parties where one of the parties is a cheap, small and handy computer, for example a smart card. In particular, we assume that this small computer is not capable of performing RSA calculations. In this chapter a protocol is presented that allows such a small computer to verify an RSA computation with high certainty, using only operations that require little memory and computing power. When the verification succeeds, the smart card can sign the result produced by the larger computer for later use.

For example, although smart cards that perform RSA are on the drawing tables nowadays, current cheap smart cards have about 128 bytes of RAM memory, 3 kilobytes of EEPROM memory (slow non-volatile memory), and an 8 bits processor. This is not enough to perform RSA in reasonable time (it would take several minutes), since the intermediate results do not fit in RAM memory. For the rest of this chapter, we will speak of the small computer as "the smart card" or SC and of the larger party as "the computer" or LC.

The smart card verifies the computation of the LC by performing the same computations as the LC on smaller numbers. The SC replaces the number by the residue (remainder after division) modulo the *verification modulus*. The verification modulus is a secret number only known to this smart card, and small enough that the remainders are manageable for it. The verification modulus can be fixed for a particular smart card or vary per instance of the protocol. This method allows the smart card to verify additions, subtractions, and multiplications. Division or modulo reduction can be verified if the LC sends both the quotient and the remainder to the SC, so that it can check if the numbers match. Repeated multiplication and modulo reduction can be used to perform RSA encryptions.

The SC never stores the numbers involved in the computation of the LC, because it always deals with residues. If the LC sends a number to the SC, the residue is

calculated during reception of this number. It is also possible for the SC to send a number that it does not store, by generating the number during transmission. In this case, the residue can also be computed during that transmission. This allows the smart card to deal with several large numbers without running out of RAM space.

The communication between a small and a large computer was addressed earlier in [MKI88, QS90]. [QS90] shows several protocols, one of which is similar to the protocol shown here, but it turns out to be insufficiently secure. Secure versions will be presented here, and the level of security is investigated for practical parameter values.

This chapter is organized as follows: first, the protocol is explained in general. Then we show an application of the protocol for a privacy-protecting payment system, and algorithms to compute the residues. Finally, we show a possible attack on the system, and compute the probability of success of this attack for different parameter values.

The Protocol

In general, the protocol consists of three phases, as depicted in Figure 19:

- agreement on initial values known to both parties;
- computation of the actual result, verified by the SC;
- issuing of a signature by the SC.

Initially, the SC and LC must agree on the values that are going to be used in the computation. How these values are obtained, is not important for the protocol; it depends on the application. The SC does not have to store these numbers; it will only store the residue modulo w. Residues modulo w are depicted by a tilde ($\tilde{}$).

In the second phase, the LC computes new values from the values that are agreed upon. The smart card performs the same computations on the residues modulo w. If the LC does a division or a modulo reduction, both the quotient and remainder are sent to the smart card for verification. If the LC wants to reduce the number p_i modulo n, the remainder r_i and quotient q_i of the division by n are sent to the SC as depicted in Figure 19. The smart card verifies the modular equation $\tilde{q}_i \cdot \tilde{n} + \tilde{r}_i \stackrel{?}{\equiv} \tilde{p}_i \pmod{w}$ and checks the range and validity of the numbers sent to it. The numbers r_i and q_i can be used in later calculations.

For RSA calculations, the exponent is computed using repeated multiplication and modulo reductions. Although it is faster than performing the calculation on the smart card, this can take some time, especially if slow communication is used (as with most smart cards). For this reason, it is advisable to use a short exponent, if possible.

For security reasons, the size of p_i is restricted; normally, p_i may be up to n^2; that is, p_i may be a product of two numbers modulo n. Increasing the size of p_i decreases the total number of bits sent (especially for RSA calculations) at the cost of security. The relation between the size of p_i and the security is shown later on.

In the last phase, the LC sends the result r of the complete computation to the SC. During reception of this number, the smart card computes both the signature on r and the residue $r \bmod w$. The smart card must do this simultaneously with the reception,

because r does not fit in its memory. The signature is a simple secret-key hash function that can be computed from r during reception without storing it. Hash functions of this form can be constructed from a conventional encryption function like DES [GPV91]. If the modular verification succeeds, the smart card sends the signature. The signature can be any number that the LC can use to prove validity of its computation in later protocols.

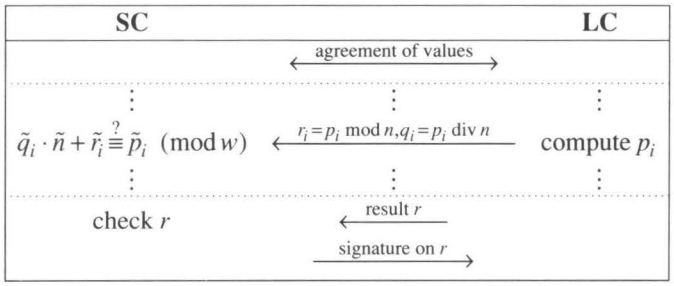


Figure 19: Sketch of the general protocol.

Example: SmartCash

To demonstrate how the protocol is used in a practical application, we show an example. This example demonstrates the use of the protocol in the SmartCash privacy protecting payment system suggested and developed by David Chaum [BC90; earlier systems appeared in BCHMS89, CFN88, Cha89, Cha90].

In the SmartCash payment system, users have a smart card owned and controlled by the organizing organization called *the bank*. Also, every user has a *transponder*, a pocket computer capable of RSA calculations and available freely on the market, that they own and control themselves. The smart card ensures the security of the system, while the transponder acts as an intermediary for the user, ensuring his privacy. The transponder is the only way in which the smart card communicates with the outside world.

We will not explain the working of the complete SmartCash system here, but only discuss those parts of the system that are relevant for the discussion. The only subprotocol of the SmartCash system that we are interested in, is the so called *blind check* protocol depicted in Figure 20. This protocol is a good example of our verification scheme. It is performed between the smartcard and the transponder (that takes the place of the LC in this protocol).

The blind check protocol prepares for a "blind signature" [Cha82, Cha85] by a signature authority called *the bank* . The bank performs the signing of the blinded number c in a later protocol with an RSA signature with public key 3. This number c is later used for doing a payment.

The transponder makes sure that the value c is perfectly blinded, ensuring the user's privacy for the payment involving this number. The bank, and the smart card as

its representative to the transponder, make sure that the transponder can only use the protocol for obtaining a signature on the number c, and not for other things. This prevents users (via their transponders) attacking the system by letting the bank sign numbers of a special form.

Smart Card		**Transponder**
compute x	$\xrightarrow{f(x)}$	
compute $\tilde{y}$	$\xleftarrow{y}$	choose y
recompute x	$\xrightarrow{x}$	verify x
compute $\tilde{b} = \tilde{x} + \tilde{y}$		compute $b = x + y$
$\tilde{q}_1 \cdot \tilde{n} + \tilde{r}_1 \stackrel{?}{\equiv} \tilde{c} \cdot \tilde{b} \pmod{w}$	$\xleftarrow{r_1 = c \cdot b \bmod n, q_1 = c \cdot b \operatorname{div} n}$	
$\tilde{q}_2 \cdot \tilde{n} + \tilde{r}_2 \stackrel{?}{\equiv} \tilde{r}_1 \cdot \tilde{b} \pmod{w}$	$\xleftarrow{r_2 = c \cdot b^2 \bmod n, q_2 = r_1 \cdot b \operatorname{div} n}$	
$\tilde{q}_3 \cdot \tilde{n} + \tilde{r}_3 \stackrel{?}{\equiv} \tilde{r}_2 \cdot \tilde{b} \pmod{w}$	$\xleftarrow{r_3 = c \cdot b^3 \bmod n, q_3 = r_2 \cdot b \operatorname{div} n}$	
compute s	$\xrightarrow{s}$	

Figure 20: The blind check protocol of the SmartCash system.

In the SmartCash version of the verification protocol, the verification modulus w is fixed. The residue $\tilde{n} = n \bmod w$ of the RSA modulus n of the bank is stored in advance in the smart card ROM. In the protocols preceding the blind check protocol, the check number c is computed by the smart card and the transponder, and the smart card keeps the residue $\tilde{c}$.

In the first phase of the blind check protocol, the smart card and the transponder agree on a blinding factor to be used for blinding the number c. They make this number together using a three step protocol: first the smart card commits to its term x using a one-way function f, then the transponder sends its term y, and finally the smart card sends x, opening the commitment. The blinding factor, the value that is going to be used for the blinding, is the sum

$$b = x + y.$$

In the second phase, the transponder computes the blinded number $r_3 = c \cdot b^3 \bmod n$, sending intermediate results allowing the smart card to verify the computation. The verifications performed by the smart card are modular versions of the defining equality of div and mod:

$$(a \operatorname{div} n) \cdot n + (a \bmod n) = a.$$

In the SmartCash protocol, the smart card does not check whether $r_i < n$ and $q_i < n$, but instead the smart card receives only as many bits as the length of the modulus. (Actually, since the number b may be larger than n (up to $2 \cdot n$), the quotients may be one bit longer than n.) This extra freedom does not allow significant extra room for cheating.

In the third phase, the smart card sends a signature s to the transponder. The final result of the protocol, r_3, does not have to be sent, since it is already sent in the second phase. Also at that time, the signature was computed during reception of the number, so

no computation is necessary here. The signature is a secret key encryption of r_3 based on DES; it is a number of 512 bits. The key used for this encryption is also known to the bank.

In the next protocol, the transponder will send $r_3 = c \cdot b^3 \bmod n$ to the bank, and get from the bank the cube root, added to the signature s:

$$\sqrt[3]{c \cdot b^3} + s = \sqrt[3]{c} \cdot b + s \pmod{n}.$$

After reception of the number, the transponder can compute the RSA signature $\sqrt[3]{c}$ by subtraction of s and modular division by b. This signature on c is used in the system at a later point.

Computation of the residues

The residue modulo w that is computed by the smart card can be computed by straightforward long division of the number sent to it by the verification modulus. To be explicit, we will show two algorithms to perform this computation. We assume that the message is processed in blocks d of b bits*; furthermore, we assume that $w > 2^b$. The first algorithm computes the residue if the number is sent with the most significant block first:

```
put 0 in r                          initialize
for i in {1...blocks}:              blocks is the number of blocks
    receive d                       receive next block
    put (r * 2**b) mod w in r
    put (r + d) mod w in r
```

The number is sent to the smart card expressed in base 2^b. At the end of execution, r contains the remainder mod r of the number. The algorithm takes two modulo reductions per received block, but the second reduction is trivial: the corresponding quotient is 0 or 1, because both r and d are smaller than w. This latter reduction can be performed using a conditional subtraction. To make this explicit, the last line of the algorithm can be replaced by the lines:

```
    put r + d in r
    if r ≥ w: put r – w in r
```

The other modulo reduction is reducing a l+b-bit number modulo a l-bit number, where l is the number of bits of the modulus, and b is the number of bits in a block. If the number of bits per block is one, all modulo reductions become conditional subtractions. If the smart card has no division instruction, this is probably the most efficient solution.

If the number is transmitted least significant block first (that is "backwards" with respect to the previous algorithm), the reduction can be performed by the slightly more elaborate algorithm

* The smart card could receive several blocks at a time for efficiency reasons.

put 0,1 **in** r, m
for i **in** {1...*blocks*}:
  receive d
  put (d * m) mod w **in** d
  put (m * 2**b) mod w **in** m
  put (r + d) mod w **in** r

The same optimizations apply as above: the last line can again be replaced by a conditional subtraction, so that the algorithm takes two modulo reductions per block. Also, all modulo reductions can be replaced by a conditional subtraction if the block size is one bit. Another optimization is to store the possible values of m beforehand when the modulus is determined. Note that m follows the powers of 2^b mod w; these numbers can be precomputed and stored in ROM or EEPROM to save a modulo reduction per block.

[QS90] proposes to use multiple small verification moduli. Although this looks more efficient than using a single modulus, in the practice of a smart card without division instruction, it is not more efficient than using one large modulus. From a security point of view, it is better to use one large modulus.

The Attack

To analyze the security of this algorithm, we first show what the LC can do to cheat. Then, we compute the probability of successful cheating. This probability depends on the way the verification modulus w is determined. Finally, we provide an overview of the values for practical parameter values.

A cheating LC tries to get the smart card to accept a different value than the value that it is supposed to compute. We consider one round of the protocol, as shown in Figure 21. In this figure, p is a number that is computed from the known values. We assume than p is smaller than n^2. The smart card computes $\tilde{p}$ by performing all computations modulo w, as before.

An attack is successful if the LC successfully makes the smart card accept another value for r than p mod n. If the LC would know w, he could simply choose any value of r' in the range $\{1,\ldots,n-1\}$ and compute the value

$$q' = \frac{p - r'}{n} \pmod{w}$$

to get a pair (r', q') that is accepted by the smart card. The choice of the value r' is done so that this is useful for cheating subsequent protocols; this value depends on the application.

The value q' can be compute so that more than one value of w of the smart card results in a successful attack. Let S be the set of values for w that are taken into account. From this, we define the *breaking modulus*

$$t = \text{lcm } S$$

and compute q' as

$$q' = \frac{p - r'}{n} \pmod{t}.$$

This pair (q', r') is accepted by the SC if $w|t$. This means that the pair is accepted for $w \in S$, but also possibly for other values from the set M of possible verification moduli. For simplicity, we include these values in S.

The smart card checks the size of r and q. To make sure that this check succeeds, r' must be chosen small enough, and we make q' small enough by taking S so small that $t \leq n$. We assume here that p gets a value that is smaller than n^2. If p is allowed to take values higher than this, q must be allowed to get values higher than n, allowing more room for cheating.

The LC attacks the system by choosing a subset S of the set of all possible verification moduli M, and computing q' for the corresponding t. The probability of success of this attack depends on the way the verification modulus is chosen, and on the set S. We will now analyze this in detail.

SC		LC
$\tilde{q} \cdot \tilde{n} + \tilde{r} \stackrel{?}{\equiv} \tilde{p} \pmod{w}$	$\xleftarrow{r = p \bmod n, q = p \operatorname{div} n}$	compute p

Figure 21: One round of the second phase.

Choice of the Verification Modulus

The smart card must choose the secret verification modulus so that the attack shown above has a probability of success that is as low as possible. More specifically, the algorithm that chooses w must be so that the probability that $w \in S$ is low, for every set S for which lcm S is smaller than n.

Let M be the set of all possible values of the verification modulus. There are two ways to choose w from this set. The obvious way is to let the smart card pick the verification modulus itself each time the protocol is executed. This makes it harder for the LC to guess the used modulus. A disadvantage of this idea is that the choice for M is restricted to sets the SC can handle. For example, M cannot be chosen as a set of primes, because testing for a prime is too hard for a smart card.

The simpler way is to always use the same value of w in a given smart card (but randomly chosen for every card). This is the solution also taken in the SmartCash system. This has as advantage that several values can be stored in the smart card ROM:

- the verification modulus;
- the residue of the system modulus $\tilde{n}$;
- the reduced values of m in the least-significant-block-first algorithm discussed above.

This saves RAM space and calculation time. A disadvantage of this idea is that the smart card issuing authority must be very careful with the information about the moduli. If somebody can get hold of the list, he can find the modulus by elimination. Preferably, the moduli must be generated with a physical random generator, and the list with moduli

must not be stored at all.

There is also a mixed solution as proposed by [QS90], where a smart card contains several prime moduli, and chooses one of these at random. This gives a small security improvement at the cost of storage space.

In the next section, we will discuss several choices for the set M.

Prime numbers

Assume that the set of all verification moduli M contains prime numbers only. For the moment, we assume that the verification modulus w is chosen using a nonuniform distribution proportional to its logarithm:

$$\text{For all } p \text{ in } M\text{: } \Pr(w = p) = c \cdot \log p\,.$$

The constant c must be so that the sum of all these probabilities is 1. We get

$$c = \frac{1}{\sum_{p \in M} \log p}\,.$$

In the case where M is the set of prime numbers in a given range I, we can use the approximation*

$$\sum_{p \in M} \log p = \sum_{x \in I} \log x \cdot \begin{cases} 1 \text{ if } x \text{ is prime} \\ 0 \text{ if not} \end{cases} \approx \sum_{x \in I} \log x \cdot \frac{1}{\log x} = \#I\,,$$

where $\#I$ is the number of integers in the range. The approximation is based on the density $1/\log x$ of the primes around x.

If the verification modulus is chosen from this M, the most efficient way for the LC to attack is to choose a set S so that $\Pr(w \in S)$ is as high as possible, while

$$t = \operatorname{lcm} S = \prod S$$

is smaller than n. It is easy to see that

$$\Pr(w \in S) = \sum_{p \in S} c \cdot \log p = c \cdot \log \prod_{p \in S} p = c \cdot \log t\,,$$

so that we can see that the maximum probability for $t \le n$ is less than $c \cdot \log n$.

Using unequal probabilities for the different primes seems unnecessary complicated. If the size of the primes in the interval is approximately equal (for example, all primes have the same number of bits), the probabilities are almost equal. In this case, the primes can be chosen uniformly random, which is much simpler. This is not a good idea if the primes are of unequal size. If a uniform distribution is chosen for primes of unequal size, a good attack strategy for the LC is to choose the small primes of M as elements of S.

For example, if the verification modulus is chosen as a random b-bit prime, the probability of a successful attack is approximated by the simple formula

$$\frac{\log n}{\#\{2^{b-1}, \ldots, 2^b - 1\}} = \frac{\log n}{2^{b-1}}\,,$$

where n is again the modulus of the RSA system. The SmartCash system uses a prime number of 64 bits and an RSA modulus of 512 bits, giving a probability of $3{,}8 \cdot 10^{-17}$.

* A mathematically more accurate approximation (with the same outcome) is possible using the Stieltjes integral; this falls outside the scope of this thesis.

This result shows that for a 512-bit RSA modulus, the primes must be at least 10 bits to get any security at all; the suggestion of [QS90] to use small primes does not give protection. This formula also illustrates that using one large prime yields more security than several small ones.

Range of integers

If we want a different verification modulus each time the protocol is executed, the smart card must be able to compute the modulus itself. Since the smart card is not able to do a prime test, we need a simpler set M. The simplest set to choose is an range I of integers, with equal probabilities for all elements. It is simple to generate a random element from such an interval, especially if the range is of the form $\{2^{b-1},\ldots,2^b-1\}$.

Unfortunately, this set M does not give such a nice theory as the previous version. One problem is that the optimal attack strategy for the LC is hard to compute. Instead of choosing a set S and computing t, we now describe the attack strategy by the value of t, so we define S as the set of divisors of t that lie in M:

$$\{d \in I | d \text{ divides } t\}.$$

The probability of success of the attack is equal to $\#S/\#I$. Ideally, we must find that value of t for which the set S is largest.

Clearly, t must be a number with many small divisors. A class of numbers that can be used for this are the *highly composite numbers* [Ram15, Ram27]. A number is called highly composite if no smaller number has more divisors. The first few highly composite numbers are:

$$2\ (2),\ 4\ (3),\ 6\ (4),\ 12\ (6),\ 24\ (8),\ 36\ (9),\ \ldots$$

where the number of divisors of each number is shown in parentheses.

A subclass of these numbers, the *superior highly composite numbers*, can be constructed simply using the formula

$$\prod_{p \text{ prime}} p^{\left\lceil \frac{1}{p^y-1}-1 \right\rceil},$$

where y is a parameter between zero and one.

Although the highly composite numbers have a maximal number of factors for their size, they are not guaranteed to have a maximal number of factors in the given range I. However, they perform better than all other numbers we tried. We do not know of numbers that give a higher probability of success for the attack.

Once such a number t is chosen, the probability of a successful attack can be written as

$$\Pr(w \in S) = \frac{\#S}{\#I} = \frac{\#\{d \in I | d \text{ divides } t\}}{\#I},$$

where $\#\{d \in I | d \text{ divides } t\}$ is the number of divisors of t in the interval I. We compute this number using the distribution of randomly chosen divisors of t:

$$\#\{d \in I | d \text{ divides } t\} = \#\{\text{divisors of } t\} \cdot \Pr(d \in I).$$

So, to compute $\Pr(w \in S)$, we need to know three numbers:

- The number of elements of I is given.
- The number of divisors of t can be computed from the factorization of t.
- The probability that a random divisors of t is in I is computed using
$$\Pr(d \in \{a,\ldots,b\}) = \Pr(d < b) - \Pr(d < a).$$

The computation of the last two probabilities requires computation of the cumulative distribution of random divisors of t. This will be done with a straightforward statistic computation. If the factorization of t is written as

$$\prod_i p_i^{\alpha_i},$$

a random divisor of t has the probability distribution

$$\prod_i p_i^{\mathrm{ur}\{0\ldots\alpha_i\}},$$

where "ur" stands for the uniform random distribution on a set. The logarithm of a random divisor of t has a probability distribution

$$\sum_i \log p_i \cdot \mathrm{ur}\{0\ldots\alpha_i\}.$$

Since this is a sum of distributions, we know from the Central Limit Theorem that this can be approximated by a normal distribution. The average and variance are not hard to compute, since the distributions are easy:

$$\mu = \sum_i \tfrac{\alpha_i}{2} \cdot \log p_i = \tfrac{1}{2}\log t$$

$$\sigma^2 = \sum_i \tfrac{\alpha_i \cdot (\alpha_i + 2)}{12} \cdot (\log p_i)^2$$

but, unfortunately, the resulting approximation of the number of divisors is not accurate enough for our purposes. A better approximation is performed by the so called *Edgeworth expansion* [KS69, or any other advanced statistics book]. This approximation uses higher order moments to approximate a probability distribution. The computation of the parameters is a lot of work, and falls outside the scope of this thesis. Results of such an expansion are given later on.

A slight improvement for the smart card is to remove numbers with small factors from the set M. The smart card could generate random numbers that are not divisible by small factors by a few trial divisions. This method ensures that it is no use for the LC to include small factors in t, so security is improved slightly.

Practical parameter values

In practice, one wants to have a low probability that the LC is able to get the smart card to sign an invalid number. The probability that is needed varies in practice, depending on the application, between 10^{-6} and 10^{-18}. We show parameter values that give practical probabilities. Results are summarized in Table 12 and Table 13.

Table 12 summarizes the case where the verification modulus is a prime number of a fixed number of bits, and the modulus is 512 bits. The table is directly derived from the formula explained above:

$$\Pr(w \in S) = \frac{\log n}{2^{b-1}}.$$

The SmartCash system uses a verification modulus of 64 bits, giving a probability of $3.8 \cdot 10^{-17}$.

$\Pr(w \in S)$	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}
prime (bits)	30	40	50	60	70

Table 12: Prime verification modulus.

Table 13 shows the computation of the security where the verification modulus is randomly chosen from a range of integers using the Edgeworth expansion. The modulus is 512 bits, as before, and the breaking modulus t is taken to be the product

$$\prod_{p \text{ prime}} p^{\left\lfloor \frac{1}{p^{0.119}-1} \right\rfloor},$$

which is

$$2^{11} \cdot 3^7 \cdot 5^4 \cdot 7^3 \cdot 11^3 \cdot 13^2 \cdot 17^2 \cdot 19^2 \cdot 23^2 \cdot 29^2 \cdot 31 \cdot \ldots \cdot 331 \cdot 337;$$

this is about $3.35 \cdot 10^{153}$, or 510 bits. The number of factors of t can easily be calculated:

$$\#\{\text{divisors of } t\} = \prod_i (\alpha_i + 1) = 537907057189370525122560 \approx 5.38 \cdot 10^{23}.$$

The Edgeworth expansions for $\Pr(d \in I)$ are computed for degrees 2 (the normal approximation), 3, 5, and 7. Since the even term of the expansion are zero, degrees 4, 6 and 8 do not give new information. The results are shown in the top three rows of Table 13.

The probabilities shown are the probability that a randomly chosen divisor of t is smaller than 2^{64} respectively 2^{63}, and the difference between these two. Note the negative probabilities for the degrees 3 and 5; this is a result of the inaccuracy of the approximation. From this we can see that, in order to get an accurate result, the Edgeworth approximation must be computed to a degree higher than 7. Computing higher degrees than this turned out to be too much work, since no software was available. The value of $\Pr(d \in I)$ determines the accuracy of the computation: the lower its value, the more terms of the Edgeworth expansion need be calculated.

The bottom row of Table 13 shows the resulting probability of a successful attack. Probably, the actual probability is lower than 10^{-6}, making the protocol useable for some practical applications.

degree of approximation	2 (normal)	3	5	7
$\Pr(d < 2^{64})$	$6.65 \cdot 10^{-11}$	$-6.71 \cdot 10^{-11}$	$-4.78 \cdot 10^{-11}$	$3.35 \cdot 10^{-11}$
$\Pr(d < 2^{63})$	$5.33 \cdot 10^{-11}$	$-5.61 \cdot 10^{-11}$	$-4.01 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$
$\Pr(d \in I)$	$1.32 \cdot 10^{-11}$	$-1.10 \cdot 10^{-11}$	$-0.77 \cdot 10^{-11}$	$0.62 \cdot 10^{-11}$
$\Pr(w \in S)$	$7.71 \cdot 10^{-7}$	$-6.43 \cdot 10^{-7}$	$-4.50 \cdot 10^{-7}$	$3.02 \cdot 10^{-7}$

Table 13: Edgeworth expansion for 64-bit verification modulus.

If n becomes more than 512 bits, t can be chosen higher, so that the probability of attack increases. In this case, the accuracy of the computation decreases. This means that a longer verification modulus is necessary, but the accuracy of the computation is so low than no clear statements can be made anymore without doing a higher degree Edgeworth expansion.

If the set M excludes numbers divisible by 2, 3, 5, and 17 (testing for 3, 5 and 17 can be done simultaneously by adding up the bytes and looking up in a bit table), we get the probabilities $1.20 \cdot 10^{-7}$ for the normal approximation and $0.40 \cdot 10^{-7}$ for the degree 7 Edgeworth approximation. This is only little better than using the entire interval for M.

Conclusion

In the practice of cheap smart cards, it is possible to make protocols where the smart cards must check those computations done by another computer. This might be a cheaper solution to systems where the smart card itself must do the calculations. The protocol is particularly interesting for privacy protecting payment systems, where RSA is needed.

Although [QS90] suggests to use several small verification moduli, this is not secure: numbers smaller than 10 bits cannot be used as verification modulus, even if multiple primes are used.

The verification modulus is most easily precomputed and stored in the smart card, but at the cost of complexity, it is also possible to let the smart card compute the modulus itself. Determining the security level in the latter case is very complicated. For practical applications, it seems best to use a fixed verification modulus.

There are a few simple extensions of this protocol:

- Doing other calculations: the smart card can verify every calculation based on addition, subtraction, multiplication and division, not only RSA calculations.
- Reducing numbers p higher than n^2: this saves protocol rounds at the cost of transmission time and a little security.
- Let the smart card perform calculations of its own, for example hash functions, during the calculation. These values can be incorporated in the computation of the LC. There is a privacy threat here, since these numbers cannot be verified by the LC. This problem can be solved by special protocols.
- Choosing a different set M. Choosing a good set M that allows the smart card to choose its own modulus randomly, while giving a good security, is still an open problem.

The security analysis of the case in which the smart card chooses the verification modulus randomly yields some interesting problems. The first problem is computing a good breaking modulus for an attack: we computed a good approximation. The hardest problem is the computation of the success probability for the attacker given this modulus. Our method using the Edgeworth expansion of a probability distribution is complicated and inaccurate; finding a better way to compute it is still an open problem. The third problem, finding a good set M that allows the smart card to increase the security, is mentioned above.

References

[BC90] **J. N. E. Bos** and **D. Chaum**: *SmartCash: A Practical Payment System*, Report CS-R9035, Centrum voor Wiskunde en Informatica (Amsterdam, August 1990).

[BCHMS89] **B. den Boer**, **C. Chaum**, **E. van Heyst**, **S. Mjølsnes** and **A. Steenbeek**: *Efficient Offline Electronic Checks*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, April 1989), pp. 294-301.

[CFN88] **D. Chaum**, **A. Fiat** and **M. Naor**: *Untraceable Electronic Cash*, Advances in Cryptology: Proc. Crypto '88 (Santa Barbara, August 1988), pp. 319-327.

[Cha82] **D. Chaum**: *Blind Signatures for Untraceable Payments*, Advances in Cryptology: Proc. Crypto '82 (Santa Barbara, August 1982), pp. 199-203.

[Cha85] **D. Chaum**: *Blind Signature Systems*, Advances in Cryptology: Proc. Crypto '85 (Santa Barbara, August 1985), pp. 18-27.

[Cha89] **D. Chaum**: *Online Cash Checks*, Advances in Cryptology: Proc. Eurocrypt '89 (Houthalen, Belgium, April 1989), pp. 288-293.

[Cha90] **D. Chaum**: *Privacy Protected Payments: Unconditional Payer and/or Payee Untraceability*, Proc. Smartcard 2000, pp. 69-93.

[GPV91] **R. Govaerts**, **B. Preneel** and **J. Vandewalle**: *Cryptographically Secure Hash functions: an Overview*, to be published.

[KS69] **M. G. Kendall** and **A. Stuart**: *The advanced theory of Statistics*, Vol. 1, Third edition, Charles Griffin & Company Limited, London.

[MKI88] **T. Matsumoto**, **K. Kato** and **H. Imai**: *Speeding Up Secret Computations with Insecure Auxiliary Devices*, Advances in Cryptology: Proc. Crypto '88 (Santa Barbara, CA, August 1988), pp. 497-506.

[QS90] **J.-J. Quisquater** and **M. de Soete**: *Speeding up smart card RSA computations with insecure coprocessors*, Proc. Smartcard 2000, to appear.

[Ram15] **S. Ramanujan**: *Highly Composite Numbers*, Proc. London Mathematical Society **2**, (no. XIV, 1915), pp. 347-409.

[Ram27] **S. Ramanujan**: *Highly Composite Numbers*, Collected Papers of Srinivasa Ramanujan, Cambridge university press, 1927, pp. 78-128.

Korte Omschrijving

Dit proefschrift bestaat uit vijf hoofdstukken die verschillende onderzoeksonderwerpen uit de cryptologie behandelen. Cryptologie is de tak van wiskunde die zich bezig houdt met informatiebeveiliging. De hoofdstukken van het proefschrift hebben met elkaar gemeen dat ze allemaal gaan over het practisch toepassen van privacy-gerelateerde cryptografische protocollen. De begrippen "privacy" en "efficiëntie" spelen een rol door het hele proefschrift.

Privacy

Vandaag de dag hebben allerlei organisaties grote bestanden met privé-informatie over hun klanten. De klanten hebben geen toegang tot deze informatie, en ze kunnen ook niet bepalen waar deze voor wordt gebruikt. Het combineren van informatie uit meerdere bestanden kan gebruikt worden om informatie te krijgen over mensen. Dit is niet alleen een theoretische mogelijkheid, het gebeurt al: bedrijven verkopen bestanden aan elkaar voor reclamedoeleinden, kredietbepalingen, statistische toepassingen en dergelijke.

Aan de andere kant hebben bedrijven te lijden onder oplichting door individuen die valse gegevens verstrekken. De bedrijven reageren met steeds strengere maatregelen om dit te voorkomen, zoals identiteitskaarten met streepjescodes en televisiecamera's bij balies. Op deze manier raken de klanten steeds meer privacy kwijt. Bedrijven kunnen niet eens aantonen dat ze de privacy van hun klanten beschermen, zelfs als ze dat (willen) doen.

In 1985 heeft David Chaum een artikel geschreven dat een oplossing geeft voor dit probleem. Deze oplossing maakt gebruik van moderne cryptografie, met name public-key systemen. Bij deze oplossing houdt iedere klant zelf zijn gegevens bij zich, zodat hij alleen gegevens hoeft te verstrekken die een bedrijf nodig heeft. Sindsdien is er veel onderzoek op dit gebied gedaan, waaronder onderzoek in dit proefschrift.

Efficiëntie

Het meest gebruikte argument tegen moderne cryptografie was altijd dat public-key systemen "te duur" waren. Hoewel de technologie het gebruik van dit soort systemen steeds dichterbij brengt, blijft er een behoefte aan goedkope systemen. Dit boek bevat

een hoofdstuk dat de snelheid van deze berekeningen verhoogt, en een hoofdstuk dat methoden laat zien om public key cryptografie te gebruiken zonder het gebruik van dure apparaten aan de kant van de gebruiker.

Overzicht

De vijf hoofdstukken gebruiken privacy en efficiëntie op verschillende manieren. We behandelen de hoofdstukken hier in het kort.

Het eerste van deze hoofdstukken, "Detecting Disrupters in Untraceable Sending", is een uitwerking van het zogenaamde Dining Cryptographers protocol dat berichten verzendt met geheimhouding van de identiteit van de zender. Het probleem bij dit systeem is dat verstoring van boodschappen (al dan niet kwaadwillend) moet worden voorkomen zonder dat de anonimiteit van de andere boodschappen verloren gaat. Drie nieuwe protocollen worden getoond die een efficiencyverbetering geven over de protocollen uit de literatuur. De hoeveelheid te zenden informatie, die erg van belang is in dit protocol, wordt in het bijzonder veel kleiner gemaakt.

Het tweede hoofdstuk, "An Efficient Voting Scheme", is een toepassing van het bovengenoemde Dining Cryptographers systeem voor het houden van een referendum. Door gebruik te maken van het tegelijkertijd verzenden van de informatie door alle partijen is het mogelijk om stemmen te tellen en hun geldigheid te bewijzen in een zeer korte tijd, terwijl de anonimiteit van de stemmers perfect bewaard blijft.

Het derde hoofdstuk, "Addition Chain Heuristics", gaat over het efficiënt berekenen van (producten van) RSA vercijferingen. (RSA is het meest gebruikte public-key cryptosysteem.) Dit gebeurt door de benodigde vermenigvuldigingen zo te rangschikken dat er minder nodig zijn. Dit "addition chain" probleem is al oud, maar de practische toepassing in computers is nieuw. Het bepalen van deze volgorde kan van de voren gebeuren (precomputation) of tijdens het uitvoeren van de berekening (real-time). Het hoofdstuk toont een aantal manieren om machtsverheffingen en producten van machten op beide manieren efficiënt uit te rekenen.

Het vierde hoofdstuk, "Provably Secure Signatures", toont een nieuw systeem voor digitale handtekeningen. Van dit systeem kan (in tegenstelling tot RSA) worden bewezen dat de kennis over oude handtekeningen niet helpt om nieuwe te maken. Er waren al systemen waarvan dit kon worden bewezen, maar deze gebruikten lange handtekeningen, en maakten het nodig veel informatie over oude handtekeningen te bewaren. Het nieuwe protocol is ongeveer net zo efficiënt als RSA, hetgeen een grote besparing in geheugen oplevert ten opzichte van de andere systemen. Ook is het zo flexibel dat het voor speciale toepassingen kan worden gebruikt.

Het vijfde hoofdstuk "Verification of RSA Computations on a Small Computer", geeft een analyse van een eenvoudig protocol dat wordt gebruikt om berekeningen van een computer te controleren met een eenvoudige smartcard. (Een smartcard is een computertje ter grootte van een creditcard.) Het voordeel van deze methode is dat een goedkope smartcard kan worden gebruikt, terwijl berekeningen kunnen worden gebruikt die anders alleen met veel duurdere smartcards gedaan kunnen worden. Het protocol wordt gebruikt in privacy-beschermende electronische betalingssystemen.

Dankwoord

Dit proefschrift kon niet tot stand komen zonder de hulp van de vele mensen die mij op een of andere manier hebben geholpen. Verschillende mensen hebben stukken ervan proefgelezen en hun commentaar gegeven. Om te beginnen zijn dat mijn collega's Eugène van Heyst, Maarten van der Ham, Ray Hirschfeld en Torben Pedersen. Verder hebben Jan van de Craats, Johan van Tilburg, Josh Benaloh en Eduard de Jong geheel belangeloos proefgelezen, en daarmee de tekst helpen verbeteren. Jack van Lint heeft als commissielid nog erg veel verbeteringen weten te vinden. De drukker Jan Schipper heeft me geholpen er ondanks tegenslagen een fraai geheel van te maken.

Ik wil vooral Henk van Tilborg bedanken, die vele versies heeft gelezen en mij door de formaliteiten van het promoveren heen hielp, en David Chaum, die mij geholpen heeft om een onderzoeker te worden. Ik wil ook mijn vrouw Jolanda bedanken die mij er door heen sleepte, en zorgde dat het allemaal de moeite waard was.

Index