Authenticated Messages for a Real-Time Fault-Tolerant Computer System

by

David Chi-Shing Chau

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

Author _____

David Chau
Department of Electrical Engineering and Computer Science
July 12, 2006

Certified by _____

Roger Racine
Thesis Supervisor, Charles Stark Draper Laboratory

Certified by _____

Professor Barbara Liskov
Thesis Advisor, Massachusetts Institute of Technology

Accepted by _____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Authenticated Messages for a Real-Time Fault-Tolerant Computer System

by

David Chi-Shing Chau

Submitted to the Department of Electrical Engineering and Computer Science
on July 12, 2006, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis develops a message authentication scheme for a new version of the X-38 Fault-Tolerant Parallel Processor (FTPP), a high-performance real-time computer system designed for applications that need extreme reliability, such as control for human spaceflight. This computer system uses multiple replicated processors to ensure that the system as a whole continues to operate correctly even if some of the processors should fail. In order to maintain a synchronized state, the replicated processors must vote among themselves to make sure that they are using identical data.

This thesis adds message authentication to the voting process. Using authenticated messages allows a system to achieve the same level of reliability with fewer replicas. This thesis analyzes where message authentication is needed in the voting process, then presents and evaluates several signature schemes for implementing message authentication. The X-38 FTPP uses radiation-hardened embedded processors, which have relatively limited computational power. Therefore, the challenge is to identify a scheme that is secure enough to guarantee that signatures cannot be forged, yet fast enough to sign messages at a high rate in real time.

Thesis Supervisor: Roger Racine
Title: Principle Investigator, Charles Stark Draper Laboratory

Thesis Advisor: Barbara Liskov
Title: Professor of Electrical Engineering and Computer Science,
Massachusetts Institute of Technology

Acknowledgments

I would like to thank Roger Racine for his guidance and support during this project. I would also like to thank Professor Liskov, for her advice and insight.

Thanks to Benji Sterling, for being a great lab partner.

Thanks to Anne Hunter, for smoothing out the bumps of the M.Eng. year.

Thanks to all the folks at Draper Lab, who were always eager to help this grad student, whether it was tech support or missing pendulum parts or office supplies that he needed.

Thanks to Daniel Myers and David Schultz, for giving me feedback on the thesis.

Thanks also to David Meyer, for explaining impenetrable math to me during the early hours of morning, and to Shuai Chen, for her daily encouragement and for helping me proofread this thesis.

Finally, finishing this thesis would have been much more painful without the support and encouragement of all my friends and my family. You all have my gratitude.

Cambridge, Massachusetts
July 12, 2006

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under the Software-Based Fault-Tolerant Computer Independent Research and Development (IR&D) program, charge no. 20317-001.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

David Chau

Contents

CHAPTER 1

Introduction

The computer system that controls a crew-carrying space vehicle must be extremely robust. Since a single system can be crippled by a single failure, a common design is to use multiple redundant systems, all running the same program and voting on the results. The challenge is to keep the replicas synchronized, and to arbitrate when they disagree.

This thesis implements and evaluates schemes for authenticated communication between the replicas of a fault-tolerant computer system. Message authentication adds digital signatures to messages to prevent the replicas from lying about the messages they receive from each other, leading to a simpler and more reliable system. Like a message authentication scheme for any other application, the one developed in this thesis needs to be secure against forgery. Unlike most other message authentication schemes, however, the one in this thesis also needs to fit within the unique constraints imposed by a real-time embedded system, constraints that limit the amount of running time and computational power available.

A variety of signature schemes are implemented and evaluated in this thesis. One of the results is that traditional signature schemes like RSA are too slow for most real-time embedded systems. Previous attempts to solve this problem have therefore used cryptographically insecure signatures instead, based on techniques such as simple

checksums and cyclic redundancy checks. However, this thesis will argue that cryptographically insecure signatures are not satisfactory for any system that aims to achieve the highest levels of fault tolerance. Recently, fast and secure signature schemes like the ones based on multivariate quadratic equations have become available, enabling this thesis to propose using cryptographically secure signatures for message authentication in real-time fault-tolerant computer systems. These new signature schemes appear very promising, although they are still slightly too slow for systems that need to send data at a very high rate. The work of this thesis can be applied to lower-rate fault-tolerant systems being built today, and to high-rate systems of the future when embedded processors have become faster.

1.1 Fault-tolerant computer systems

Some computer systems must not fail, especially if people's lives and safety depend on the correct operation of the system. For certain systems, a single unreplicated unit achieves the required level of reliability. However, an unreplicated system, even if carefully designed and extensively tested, is still vulnerable to random hardware faults. Therefore, an extremely robust system will likely need to be a replicated one.

Redundancy increases the chance of a system surviving faults. However, it also increases the complexity. How will the replicas communicate and coordinate with each other? More importantly, how will the system respond when the replicas, which are supposed to be running the same program and producing the same results, disagree with each other?

To analyze the behavior of the system as a whole when a replica fails, one needs to consider how the replica fails. In the best case, the faulty replica might realize

that it has entered an inconsistent state, and announce the error to the other replicas. The other replicas would then know to exclude the results from that faulty replica. A harder case to handle is when the faulty replica ceases to communicate altogether. The remaining replicas would need to infer that the faulty replica has failed, most likely using a timeout.

The hardest case to handle is when the faulty replica continues to communicate, reporting incorrect results. The faulty replica may even report one result to some of the non-faulty replicas, and a different result to the others. Since the non-faulty replicas must maintain identical internal state, they must vote among themselves to reach an agreement about how to handle the result from the faulty replica. This voting process will take additional rounds of message exchange, and since the identity of the faulty replica is not known a priori, the faulty replica will participate in the voting process too, giving it additional opportunities to confuse the non-faulty replicas.

In this worst-case scenario, one might as well treat the faulty replica as if it were malicious, and assume that it is deliberately choosing messages that will most confuse the other replicas. (In some systems, especially ones in which each replica is controlled by a different party, the faulty replica may actually be malicious.) This sort of behavior from the faulty replica, called a *Byzantine fault*, must be tolerated in a robust system.

Of course, building a reliable system requires more than just handling misbehaving replicas. If the replicas are not electrically isolated, for example, then a power glitch in one may cause a power glitch in the others. And if all of the replicas are running the same code, they will all be equally defenseless if the fault is caused by a mistake in that code.

1.2 Real-time embedded systems

A *real-time system* is one that must react in a strictly bounded amount of time. On a desktop computer, one real-time task might be to reload the sound buffer before it empties. In a space vehicle, the real-time task might be to fire a rocket. If the desktop computer fails to react on time, then the music it is playing will skip. If the computer controlling the space vehicle fails to react on time, then it will lose control of the spacecraft.

Being a real-time system imposes several constraints on the design of the system. One constraint is scheduling: since a computer system runs several tasks simultaneously in practice, a real-time design must make sure that every task can run when it needs to. A real-time system also constrains the algorithms available to choose from. An algorithm that runs quickly most of the time cannot be used if it runs slowly in the worst cases.

Most real-time control systems are also *embedded systems*, which further limits the resources available. A typical embedded processor might run an order of magnitude slower than a contemporary desktop processor. In addition, processors for space flight need to be radiation hardened, which again puts them a generation behind unhardened processors. Because of the speed difference, many algorithms that run acceptably quickly on desktop processors run too slowly on embedded processors.

1.3 The X-38 Fault-Tolerant Parallel Processor

The Fault-Tolerant Parallel Processor (FTPP) [RLB02] is an architecture for a fault-tolerant computer system on top of which applications can run, developed at the

Charles Stark Draper Laboratory. The X-38 FTPP is one incarnation of this architecture, developed for NASA's X-38 Crew Return Vehicle. This thesis attempts to develop a message authentication scheme for a new version of the X-38 FTPP.

Generically speaking, the FTPP consists of a number of *fault-containment regions* (also called *channels*), each of which contains a number of processors. The fault-containment regions are isolated from each other, and they are designed to fail independently. Each fault-containment region also has a *network element*, which is connected to each of the other network elements, and which provides communication between the fault-containment regions.

The processors in the FTPP can assume a variety of configurations. A processor may run alone, or it may run the same program as one or more of its replicas, each of which resides in a different fault-containment region. A program replicated on multiple processors needs to be only minimally aware that it is running on multiple processors; the network element ensures that each processor receives the same inputs and arbitrates when their outputs differ. The network element performs the multiple rounds of message exchange needed for the voting process.

In addition to fault-tolerant communication, the FTPP architecture also offers facilities for fault-tolerant time synchronization, recovery and reconfiguration when a replica fails, task scheduling, and other services as well.

1.4 Evolution to a software-based system

The X-38 FTPP is built out of mostly commodity parts, with the notable exception of the network element, which is custom hardware. Draper Laboratory is engaged in a project to build a new version of the FTPP that uses all commercial off-the-shelf parts,

a project that this thesis is part of. The functions of the network element, which are currently performed by dedicated hardware, will be implemented in software instead.

Moving to software has several advantages. It reduces the number of hardware parts that can fail, and it also makes modifications and testing easier. Embedded processors have only recently become powerful enough to perform the network element's job in software. In addition, other tools for building robust software-based fault-tolerant systems have also recently become available, such as partitioned real-time operating systems, which allow strong separation between software components in a system. A partitioned operating system strictly limits each component to its own share of memory, processor time, and other resources. (In contrast, a desktop operating system typically makes no such guarantee. For example, most desktop operating systems do not guarantee that a given process will always run within a certain period after an interrupt, regardless of whatever else the system is doing. Most desktop operating systems also do not prevent one process from consuming so much memory that other processes cannot allocate any.)

1.5 Authenticated messages

The move to a software-based system opens the opportunity for making some other changes to the FTPP as well. Among the most significant of these changes is the addition of authenticated messages to the message-voting process.

Message authentication allows the recipient of a message to verify that the message genuinely originated from the claimed source, and that the message has not been modified. Message authentication is achieved by augmenting the message with a digital

signature, which the recipient verifies. The most relevant benefit of message authentication to the voting process of a fault-tolerant system is that message authentication prevents one replica from lying to another replica about what message a third replica sent. Removing this particular vulnerability allows the system to be simplified while retaining the same level of fault tolerance. Message authentication also helps detect when messages have been corrupted in transit, but this is a more minor benefit, since any reasonably robust system must already resist message corruption.

This thesis examines the many digital signature schemes available, and evaluates their suitability for use in a real-time fault-tolerant system. Again, one of the more important constraints this project faces is that the scheme must operate quickly, on processors that have relatively limited computational power. To meet this constraint, this thesis explores some of the less common signatures schemes in addition to the well-known ones like RSA. It examines how to optimize the implementation of each of these schemes. Finally, this thesis also considers the message-voting process itself, to determine precisely which phases of it need message authentication.

1.6 Previous work

Fault-tolerant computer systems have been built for almost as long as computer systems have been, so it is impossible to give a complete account of all the research that has enabled and shaped this thesis. Instead, this section will give a brief summary of the influential systems that have been developed for applications that need extreme reliability, such as flight control.

1.6.1 Previous systems

The early attempts at building redundant fault-tolerant systems in the 1960s were implemented with simple replication of hardware. For example, a processor would have several memory units, and it would write the same data to all of them. Special circuitry would detect disagreements among the memory units [HLS87].

The Fault-Tolerant Multi-Processor (FTMP), which was developed beginning in 1975, used hardware for fault detection and masking. Processors were arranged in groups of three, as were memory units, and dedicated hardware performed the voting among them [HLS87]. The system was called a "multi-processor" because it ran several different programs simultaneously on different sets of processors. It was only later when computers had become faster that designs for redundant processors running only a single task at a time were considered.

The contemporaneous rival to the FTMP was Software Implemented Fault Tolerance (SIFT), which, as its name suggests, was a computer that used software to implement voting and error detection. SIFT was built using off-the-shelf minicomputers and microcomputers. The computers in SIFT were only loosely synchronized with each other, and they would vote among themselves only at the end of an iteration of a computation. This design made the analysis of the system simpler, since the analysis only had to consider whether one of the computers had failed or not, and not whether the individual components in each computer were failing. SIFT was also notable in that it advocated formal mathematical proofs of correctness for the system [WLG78]. Unfortunately, performing the error detection and fault recovery in software was extremely slow. One evaluation determined that SIFT spent 80% of its computing power on performing this overhead [PB86].

The Fault Tolerant Processor (FTP) in the 1980s was the successor to the FTMP. Unlike the FTMP, which was a multi-processor, the processors in the FTP acted as a single virtual processor. The FTP used hardware to ensure that all of the processors received the same input, and to vote on their outputs. Because of its simpler design, the FTP was much more efficient than the earlier systems, and it was easier to program as well [HLS87].

The Multicomputer Architecture for Fault-Tolerance (MAFT) was a system designed in the late 1980s for high performance. Each node in MAFT consisted of two parts, the operations controller and the application processor. The operations controllers, which were connected to each other, handled all of the communication and system management functions, leaving the application processor to focus on running the application itself. One interesting aspect of the MAFT architecture was that it was designed to allow different implementations for the replicas. For example, two replicas running the same program might each have been built by a different group, and thus they would produce slightly different results for a floating-point computation. The MAFT system was responsible for reconciling their results [KWFT88].

Finally, the Fault-Tolerant Parallel Processor (FTPP) is the current generation of architectures for fault-tolerant computer systems, and is the one that this thesis develops a message authentication scheme for. Like some of the earlier multi-processor architectures, the FTPP can run multiple tasks at the same time, with each task being run on a different group of replicated processors. For the most part, applications do not need to be aware that they are running on a replicated system; the FTPP handles the job of ensuring that they receive the same inputs. To reduce the complexity of the system, multiple processors share a single connection to the inter-processor network [HL91]. Chapter 3 describes the FTPP architecture in greater detail.

1.6.2 Previous message authentication designs

There have been several proposals for message authentication schemes for fault-tolerant systems, a few of them from work also done at Draper Laboratory. [Ga90] proposes using cyclic redundancy checks (CRCs) to sign messages, and considers how to implement such a scheme in hardware. [Cl94] combines CRCs with modular multiplicative inverses to form another signature scheme. Finally, [St95] designs a three-node Byzantine-resilient system called the Beetle that uses message authentication to allow it to tolerate a fault in any one node.

This thesis builds on these previous results, and considers the problem of developing a message authentication scheme for the FTPP. In contrast to these previous proposals, this thesis advocates using a cryptographically secure signature scheme for implementing message authentication, and systematically evaluates several such schemes.

1.7 Overview of this thesis

This chapter introduces the work of this thesis and explains its significance. It gives a background on real-time fault-tolerant computer systems, and highlights some of the challenges of implementing a message authentication scheme for it.

Chapter 2 elaborates on Byzantine fault tolerance. It formally presents the problem of reaching agreement in the presence of faults, and examines some of the possible configurations for a fault-tolerant system.

Chapter 3 gives a detailed description of the X-38 Fault-Tolerant Parallel Processor. It describes the features of the FTPP that the new software-based version will need to replicate.

Chapter 4 examines how message authentication can help the message-voting process in a fault-tolerant system. It explains the requirements that a suitable signature scheme will need to meet, and argues for the importance of cryptographic security.

Chapter 5 introduces the signature schemes themselves. It looks at RSA, DSA, and elliptic curve DSA, as well as two schemes based on systems of multivariate quadratic equations, SFLASH and TTS.

Chapter 6 describes the actual implementation of the signature scheme candidates. It describes the optimizations that were used, then evaluates and compares the performance of each scheme.

Chapter 7 concludes this thesis. It discusses the implications of the performance results, summarizes the work of this thesis, and explores possible avenues for future research.

CHAPTER 2

Byzantine Fault Tolerance

When a single standalone computer system cannot be made reliable enough to meet the demands of an application, one solution is to use multiple redundant systems, all performing the same task in parallel. The probability of all of the replicas in a redundant system failing is lower than the probability of an unreplicated system failing, assuming that the replicas are properly isolated from each other. Therefore, a redundant system can better tolerate faults.

Building a redundant system is challenging, however. If a replicated system consisting of multiple nodes is controlling a vehicle, for example, what should the vehicle do when the nodes give it conflicting commands? One possibility is to add yet another node, an arbitrator, to the system, and to let the arbitrator have the final say on what to do. If the arbitrator fails, however, then the whole system fails, and thus the resulting system is no more reliable than a single-node system. In systems where extreme fault tolerance is needed, one possible solution is to use mechanical voting instead: every node is connected to an actuator, and as long as the non-faulty nodes are the majority, they can physically overpower the faulty nodes that are giving bad commands. This solution handles the problem of conflicting outputs.

The problem of input is a little harder to solve. Consider a system with several nodes that needs to read values from a sensor. Two designs are possible. The first is to

give each node its own sensor. This creates the possibility that each node will read a different value. The varying readings might cause different nodes to do different things, eventually causing all the nodes to end up in wildly divergent states. The other possible design is to use a single sensor, and have it send its readings to all of the nodes. However, if the sensor is faulty, it may send different values to different nodes, again causing them to diverge. The problem in both designs is that they need a way to make all of the nodes agree with each other.

The problem of getting multiple nodes to agree has been formalized as one of several problems. In the *consensus problem*, each node starts with its own version of a value, and the goal to have all of the nodes agree on a single value at the end of the process. In the *interactive consistency problem* [PSL80], each node again starts with its own value, but the goal here is to have all the nodes agree on what value every node in the system has. That is, every node should end the process with the same vector $(v_1, v_2, v_3, \ldots)$, where v_i is the value that the ith node started with. Finally, in the *Byzantine generals problem* [LSP82], one node has a value that it wants to send to all the other nodes; at the end of the process, all of the other nodes should agree on the value that was sent. The challenge in each of these problems is to correctly handle the case where some of the participating nodes are faulty.

The Byzantine generals problem is the hardest of the three, in the sense that a solution to the Byzantine generals problem would also solve the other two problems [Fi83]: the interactive consistency problem can be solved by simply repeating the solution to the Byzantine generals problem multiple times, making one of the nodes be the sender each time. And once the solution to the interactive consistency problem has finished, each node will have the same vector of values, from which each node can

simply pick the majority value (or the mean value, or the result of any other choice function), thereby also solving the consensus problem.

This chapter describes the Byzantine generals problem, and discusses the implications that it has for building fault-tolerant systems. Although most of this thesis focuses on the X-38 Fault-Tolerant Parallel Processor (FTPP), which is a fault-tolerant computer system with a very specific architecture, this chapter aims to explore fault-tolerant systems more broadly. It will present some of the theoretical results on fault-tolerant systems, and will examine how different architectures give fault-tolerant systems different capabilities.

2.1 The Byzantine generals problem

The Byzantine generals problem received its name from the whimsical description in [LSP82]:

> We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with each other only by messenger. After observing the enemy, they must decide on a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that
>
> A. All loyal generals decide upon the same plan of action.
> B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

This is the consensus problem. [LSP82] then observes that this consensus problem can be reduced to the case where one general is the commanding general and the remaining

generals are lieutenant generals, and the commanding general needs to send a command to the lieutenants. Hereafter, the "Byzantine generals problem" will refer to the case of one commander sending a command to several lieutenants, and not to the overall consensus problem.

More formally, the problem describes a system consisting of a total of n generals, counting the commander. Among these n generals, f of them may be traitorous, including possibly the commander. The commander has a single-bit value to send to the lieutenants. The goal is to design a protocol to ensure that all of the loyal lieutenants receive the value that the commander has if the commander is loyal, or to ensure that all of the loyal lieutenants at least agree on some value at the end of the protocol if the commander is traitorous.

A traitorous general is a general who can violate the protocol. At each step in the protocol where a general is supposed to send a message, a traitorous general can send any message it wants, or no message at all. Several traitorous generals can collude with each other. Compared to earlier models for fault-tolerant systems, the Byzantine generals problem is notable because it allows faulty nodes (generals) to behave in arbitrary ways, even to the point of actively and maliciously trying to derail the protocol.

The solvability of the Byzantine generals problem, and the complexity of the solution, depends on the characteristics of the network the generals use for communication with each other. Some of these distinguishing characteristics include whether the network is broadcast or point-to-point, whether the network offers synchronous or asynchronous communication, and whether the messages sent through the network are signed or unsigned.

2.1.1 Broadcast versus point-to-point networks

A point-to-point network is a network that gives every general a separate link to every other general, and each general can always tell which of its peers a message arrived from. In contrast, the generals cannot identify the senders of messages in a broadcast network.

In a broadcast network, if the generals are not allowed to sign their messages, agreement is impossible. A traitorous general could forge as many messages as it wants, from whichever general it wants. A recipient who is being targeted by a traitorous general would not be able to draw any useful conclusions from the set of messages that it receives.

On the other hand, if in a broadcast network the generals *can* sign their messages in a way that a traitorous general cannot forge, then the generals can always identify the senders of messages. Thus, message authentication can be used to make a broadcast network act like a point-to-point one. This capability can be useful when the number of generals becomes very large, since the number of point-to-point connections needed grows quadratically with the number of generals. A broadcast network can have fewer connections.

A broadcast network using signed messages is not exactly equivalent to a point-to-point network, however, since the two types of networks have different physical topologies, and topology does have an effect on the robustness of the system. In the simplest broadcast network, all of the generals are connected to each other through a single hub. In this case, the hub is a single point of failure. A more complicated and redundant network, like the Internet, is more resilient.

For the FTPP, which has a relatively small number of nodes, a point-to-point network makes the most sense.

2.1.2 Synchronous versus asynchronous networks

In a synchronous network, message transmission takes a fixed and known amount of time. In an asynchronous network, on the other hand, messages may be delayed arbitrarily, and they may arrive out of order. The Internet can be modeled as an asynchronous network, since there is no guarantee on how long a message will take to arrive. Point-to-point networks, on the other hand, are usually synchronous, since transmission on each link takes a fixed amount of time.

One of the more important theoretical results is that a single traitorous general can prevent agreement from being reached if the network is completely asynchronous [FLP85]. The traitorous general does not even need to actively send incorrect messages; it can prevent agreement by simply falling silent at the right moment during the agreement protocol. This theoretical result is a strong one, but it depends on the network being completely asynchronous. For example, it forbids the generals from having synchronized clocks, or from using any sort of timeouts. In practice, fault-tolerant computer systems can be built on asynchronous networks if these restrictions are relaxed.

The FTPP fortunately does not need to deal with the complexities of asynchronous networks, since its processors are time-synchronized with each other, and since it uses point-to-point links with constant transmission delays.

2.1.3 Oral versus written messages

According to the terminology in [LSP82], an *oral message* is one that is not signed. With an oral message, one general cannot prove to another general that a third general said something. *Written messages* are signed, so a general can use the signature to prove to another general that the signer really did write the message.

The distinction between oral and written messages is particularly relevant to systems using synchronous point-to-point networks. In such a system, solutions to the Byzantine generals problem are possible using both oral and written messages, but solving the problem with oral messages requires a larger proportion of loyal generals and a larger number of messages. One of the goals of this thesis is to convert the FTPP from a system that uses oral messages to a one that uses written messages.

In a synchronous point-to-point system that uses only oral messages, [PSL80] and [LSP82] show that agreement is possible only if $n \geq 3f + 1$, that is, the number of traitorous generals must be strictly fewer than one-third of the total number of generals. Furthermore, an agreement protocol must take at least $f + 1$ rounds, where a general cannot send the messages for a round until it has received all of the messages from the previous round. Finally, agreement is only possible if the connectivity of the network is greater than $2f$; that is, for every pair of generals i and j, there must be enough paths between them that i and j are still connected if any two vertices in the graph of the network are removed [Do82].

One consequence of these results is that there is no way for a set of three generals to reach agreement using only oral messages if one of them is traitorous.

Reaching agreement is easier if the generals can use written messages. In this case, agreement is possible for *any* number of generals, no matter how many of them are traitorous [LSP82]. (The definition for agreement requires that all the loyal lieutenants agree with each other, and that they agree on the value from the commander if the commander is loyal. A system with only one general in it vacuously meets this definition. A system with two generals, one loyal and one traitorous, meets this definition too, since there is no one else for the lone loyal general to agree with.) In practice, however,

the generals must produce a result for some external observer, and so the correct result should be the majority result. In this case, the loyal generals must outnumber the traitorous generals, so $n \geq 2f + 1$. Using written messages, it *is* possible for three generals to reach an agreement if one of them is traitorous.

The connectivity requirements for a network using written messages is lower too. The only requirement is that all of the loyal generals must have some way of communicating with each other without needing to send messages through a path controlled by a traitorous general. Therefore, in the network graph, the loyal generals just need to be connected, which means that the connectivity of the network needs to be at least $f+1$ [LSP82]. The minimum number of rounds is still $f+1$ for written messages, however, which is the same as for oral messages [DS83].

This discussion of oral and written messages in synchronous point-to-point systems has so far only considered the case of traitorous generals. In a real system, the links can be traitorous as well. A faulty link may modify messages in transit, or it may fail to deliver some of the messages altogether. However, to an outside observer, a traitorous link is indistinguishable from the case where one of the generals at either end of the link is traitorous. Although each link connects two generals, a single traitorous general is sufficient to produce exactly the same effect as a faulty link. Therefore, any protocol that can tolerate f traitorous generals can also tolerate f faulty links if none of the generals are traitorous [Fi83].

2.2 Implications for fault-tolerant systems

Some systems can only tolerate faults that cause nodes to stop responding. A *Byzantine-resilient* fault-tolerant system, on the other hand, must operate correctly even if the

faulty nodes act like traitorous generals and actively try to thwart the agreement protocol. This model makes sense if the system is distributed over the Internet, and each node is controlled by a different party, since some of the nodes may very well be actually malicious. However, Byzantine resilience is a desirable design goal even if the nodes are *not* expected to be malicious. It is very hard, perhaps impossible, to predict all the ways in which a system may fail, and to analyze all of their consequences. For the effects it produces, a particularly unfortunate and unexpected fault might as well be malicious. Byzantine resilience guarantees that the system continues to operate correctly no matter how a failed node behaves.

Designing a system to be Byzantine resilient can be more difficult than designing a system to handle only specific faults. At the same time, however, a Byzantine-resilient system is easier to analyze. By allowing the faulty nodes to behave in arbitrary ways, the designer no longer needs to perform a case-by-case analysis of the effects of every conceivable fault. Instead, an analysis of the system only needs to consider whether each node has failed or not. As [WLG78] explains in its description of the seminal SIFT fault-tolerant computer system:

> The study of fault-tolerant computing has in the past concentrated on failure modes of components, most of which are no longer relevant. The prior work on permanent "stuck-at-one" or "stuck-at-zero" faults on single lines is not appropriate for considering the possible failure modes of modern LSI [large-scale integration] circuit components, which can be very complex and affect the performance of units in very subtle ways. Our design approach makes no assumptions about failure modes. We distinguish only between failed and nonfailed units. Since our primary method for detecting

errors is the corruption of data, the particular manner in which the data are corrupted is of no importance.

CHAPTER 3

The X-38 Fault-Tolerant Parallel Processor

The Fault-Tolerant Parallel Processor (FTPP) is an architecture for high-performance fault-tolerant computer systems, developed at the Charles Stark Draper Laboratory. The X-38 FTPP is one implementation of this architecture, designed for the flight control of the NASA X-38 experimental crew return vehicle. The X-38 FTPP was built from mostly commercial off-the-shelf components, yet it achieves an availability of at least 99.999% [RLB02].

This chapter describes the FTPP architecture and the X-38 FTPP computer system. The description here draws from [HL91], [RLB02], [Ed02], [Bu01], and [CDSL02].

3.1 Architecture

Applications on the FTPP run on a virtual processor, where each virtual processor is actually a number of physical processors, each with their own memory, and all running the same program. An FTPP system has multiple virtual processors, allowing it to run several tasks simultaneously. In the X-38 FTPP, a virtual processor can consist of between one and four physical processors.

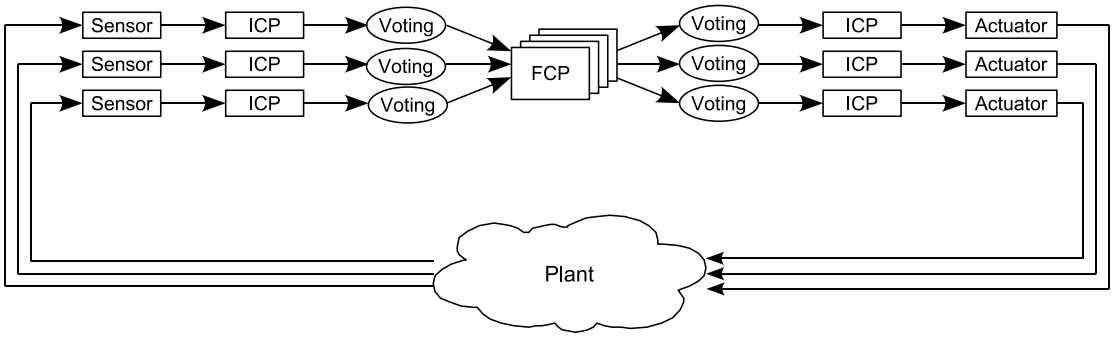

Figure 3-1: Conceptual architecture of the X-38 FTPP (based on the diagram in [Bu01]). Readings from sensors are collected by the ICPs, which are not replicated. The single-source data from each ICP is voted before being passed on to the processors of the replicated FCP, to ensure that each processor of the FCP uses identical input. The output from the processors of the FCP is voted again before being sent to the actuators, which are also controlled by a set of ICPs.

As an example, the virtual processor for an instrumentation control processor (ICP) consists of just one physical processor. An ICP performs input/output with devices outside of the FTPP, such as sensors and actuators. ICPs are not replicated, although it is certainly possible to have multiple sensors connected to different ICPs measuring the same quantity. The virtual processor for the flight-critical processor (FCP), on the other hand, is replicated across four physical processors. The FCP uses the data collected by the ICPs to perform the actual flight-control computations. Figure 3-1 illustrates how the FCP and ICPs cooperate to control a physical plant.

The physical processors that make up a virtual processor run the same program and should have the same state, but the physical processors are not synchronized at the machine instruction level. Instead, each processor's clock is allow to differ slightly from the other processors' clocks by some tolerance amount, and the processors vote among themselves mainly when they receive input or produce results. The different processors that make up a virtual processor are synchronized through the sending and receiving

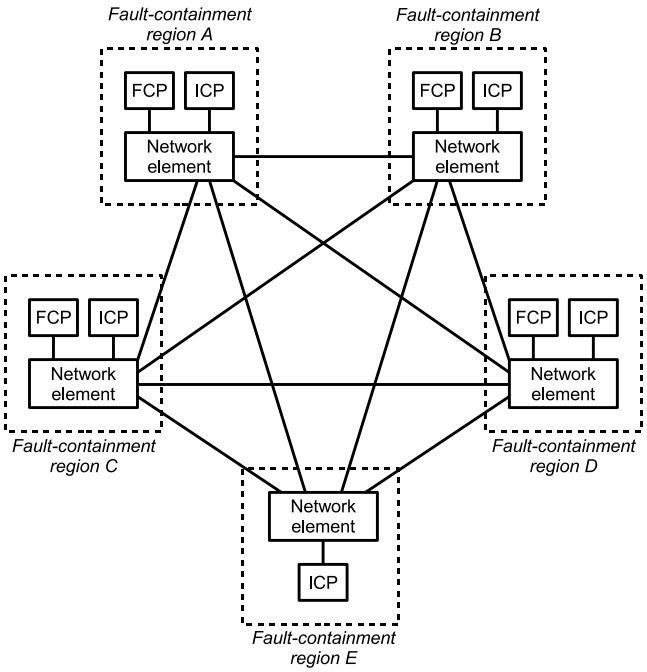

Figure 3-2: Physical architecture of the current version of the X-38 FTPP. The X-38 FTPP consists of five fault-containment regions, each of which contains a single physical processor from the four-processor FCP. Each fault-containment region also contains one or more ICPs, which are unreplicated. The network elements connect the different fault-containment regions. Since the FCP consists of only four physical processors, the fifth fault-containment region does not contain a physical processor from the FCP. In a version of the X-38 FTPP that uses message authentication, the fifth fault-containment region can be eliminated.

of messages.

Because of the large number of physical processors in the system, it would be impractical to connect all of them to each other directly. Instead, the physical processors are grouped into *fault-containment regions*, which are regions that are isolated from each other, and therefore are expected to fail independently. Each of the physical processors that makes up a virtual processor goes into a different fault-containment region.

The fault-containment regions are connected to each other through their *network elements*. The network element is responsible for handling all communication, whether it is between physical processors in the same virtual processors, or between different virtual processors. Each fault-containment region has a network element, and all of the network elements are connected to each other in a complete-graph topology, with point-to-point links between every pair of network elements. The processors interface with the network element through shared memory.

Figure 3-2 illustrates the physical architecture of the X-38 FTPP, showing the processors, the network elements, the fault-containment regions, and the connections between them.

The network element helps make the replicated nature of the system transparent to the applications. For the most part, an application can just assume that it is running on a single processor. The network element makes sure that every processor in the same virtual processor receives the same inputs.

The X-38 FTPP is largely built out of commercial off-the-shelf hardware. The main exception is the network elements, which are implemented in custom hardware because of the high performance required of them.

3.2 Capabilities

The X-38 FTPP provides several services to the applications that are running on it. The most significant of these is communication.

The X-38 FTPP offers several classes of communication primitives. A Class 1 exchange is a single-round exchange for voting data among the physical processors in a virtual processor, to ensure that all of the physical processors have the same value.

In a Class 1 data exchange, each processor sends its version of the value to every other processor. Then each processor takes the resulting vector of values that it receives, and picks the majority value as the value to use. The majority value is found by performing a bit-by-bit vote on the messages. (This is the conceptual description of the process; in reality, it is the network elements and not the processors that send the values to each other and vote on the results.)

A Class 2 data exchange sends a value from a single processor to a group of processors that must be in agreement about the value. This is the classic Byzantine commander and lieutenants problem that [LSP82] describes. A Class 2 exchange is used for both sending data from an ICP (single physical processor) to the FCP (multiple replicated processors), as well as for sending data from a member of the FCP to the rest of the FCP.

The Class 2 data exchange proceeds in two rounds. The FTPP must tolerate one fault during the process, and so it needs $f + 1 = 2$ rounds of message exchange, as shown in [LSP82]. In the first round, the source processor (really its network element) sends its value to each of the recipient processors. In the second round, the recipient processors reflect the value that they received to each other. Finally, each recipient takes the vector consisting of the copies of the message, and picks the majority value as the value to use.

The specification for the current version of the FTPP requires that a Class 2 data exchange, with its two rounds of messages, be completed in 200 μs. A less stringent requirement is that the system has at most 1 ms to make all of the needed data available to the tasks that run every 20 ms, which is the highest rate at which tasks are scheduled on the FTPP.

The FTPP provides several other services in addition to communication. It manages the startup process, during which it detects which processors are available, and puts them into a synchronized state. It schedules the tasks that run on each processor according to their priority and the frequency with which they need to run. It maintains a distributed clock across the multiple fault-containment regions, providing a consensus value for the current time to applications that need it. It also detects and reports the faults that occur.

3.3 Fault handling

The FTPP has multiple layers of fault detection, masking, and recovery. It performs self-testing to detect internal errors. It also uses voted message exchanges, as described earlier, to mask faults that affect any single processor.

The FTPP is designed to handle two non-simultaneous Byzantine faults. Here, a fault is defined as a failure of one fault-containment region in any way. Two non-simultaneous faults means after the first failed fault-containment region has been detected and isolated, the system can continue to operate correctly if a second fault-containment region fails. Of course, there is no guarantee that the system can always identify which fault-containment region has failed and remove it. If a second fault-containment region fails before the system can identify the first failure, then the two failures are considered simultaneous, and thus are beyond the scope of the system.

The requirements for the X-38 FTPP call for handling only one fault at a time. Making the system tolerate two simultaneous faults would be costly. As [LSP82] shows, in a system that uses non-authenticated messages, handling two simultaneous faults would require $3f+1=7$ fault-containment regions. Connecting all 7 fault-containment

regions to each other would require 21 links. Furthermore, performing a single-source message exchange would require $f + 1 = 3$ rounds of messages. Requiring that the two faults be non-simultaneous strikes a good balance between robustness and cost.

Even though the largest virtual processor only has four physical processors, the X-38 needs five fault-containment regions to handle two non-simultaneous faults. (With five regions, after the first one fails and is removed, the system will have four regions left, which is the $3f + 1$ regions that are needed to handle the next fault.) The fifth fault-containment region is degenerate in the sense that it only exists to participate in the message-voting process. If the X-38 FTPP had used authenticated messages instead, this fifth fault-containment region would not be needed.

When a fault does occur, the FTPP has several ways of dealing with it. The FTPP can reset the link between a pair of fault-containment regions, which often helps to clear up a transient error. It can remove a physical processor from a virtual processor, degrading the virtual processor to run with one fewer processor. If the processors vote to do so, the FTPP can reboot a faulty processor. The FTPP can also rewrite the failed processor's memory with a good copy, and reintegrate the processor into its virtual processor.

3.4 A new software-based FTPP

The current version of the X-38 FTPP uses custom hardware for its network element. However, it would be preferable to build the FTPP entirely out of commodity components, and to perform the functions of the network element with software. A software implementation is easier to build and change, and it would reduce the number of parts

that can fail. Therefore, Draper Laboratory is engaged in a project to build a new software-based FTPP, a project of which this thesis is part.

Along with the move to a software implementation of the network element, several other improvements will happen. First, the FTPP will get updated with newer and faster hardware. The faster hardware will enable the network elements to sign the messages that they send, which in turn will allow the system to use just four fault-containment regions instead of five. Also, the new version of the FTPP may use a partitioned operating system, which provides very strong separation between processes running on the same processor. This would allow the new FTPP to use a single processor to perform the work of several.

Moving to a software-based system introduces several new challenges as well. Implementing the network element in software means that less of the processor's time is available for other tasks. A software implementation must ensure that it does not add too much overhead. Signing messages to implement message authentication adds to the load on the processors too; this thesis is an attempt to find an acceptably efficient signature scheme. Finally, a software-based version of the network element will need new implementations of the communication and time-synchronization protocols, which is a problem that is addressed in the contemporaneous thesis [St06] by Reuben Sterling.

CHAPTER 4

Message Authentication

The current version of the X-38 Fault Tolerant Parallel Processor (FTPP) does not digitally sign the messages sent between the network elements. One of the tasks in the development of the new software-based version of the FTPP is to add message authentication using digital signatures. Authenticated messages prevent a network element from lying about what it heard from another network element, thereby allowing the design of the FTPP to be simplified. Whereas the current version of the FTPP needs five fault-containment regions, a version using message authentication can achieve the same reliability with just four fault-containment regions.

This chapter explains how the addition of message authentication allows the number of fault-containment regions to be reduced by one. It shows that only single-source messages need to be signed, and only by the original sender. It argues that the message authentication scheme must be cryptographically secure, even though a faulty processor is certainly not expected to actually attempt to break the scheme. Naturally, cryptographically secure schemes are more resource-intensive, which conflicts with the constraint of limited computational power. Finally, this chapter looks at some of the intricacies involved in implementing a secure message authentication scheme, such as the challenges of random number generation and the need to prevent message replays.

4.1 How message authentication helps

It must be emphasized that the main goal of adding message authentication is to simplify the design of the FTPP, by reducing the number of fault-containment regions from five to four, while still maintaining the same level of fault tolerance. Although message authentication also helps to detect random message corruption during transmission, detecting transmission errors is *not* the main reason for adding message authentication to the FTPP. In any case, the current version of the FTPP, without message authentication, is a robust fault-tolerant system, and therefore already has mechanisms for insuring message integrity.

Without message authentication, a system needs at least four nodes to tolerate a Byzantine fault in any single node [LSP82]. Therefore, a five-node system is needed to tolerate two non-simultaneous faults: after the first faulty node is detected and removed, the remaining system is a four-node system that can tolerate the second fault. (As mentioned earlier, however, there is no guarantee that the system *can* identify and remove the node experiencing the first fault. If the system cannot, then the second fault would be simultaneous with the first, and two simultaneous faults fall outside the specifications of the system. This is a limitation of the FTPP both with and without message authentication.) This section shows how message authentication enables a three-node system to tolerate one fault, thereby making a four-node system sufficient to tolerate two non-simultaneous faults.

4.1.1 Multi-source exchanges

Multi-source (Class 1) exchanges do *not* benefit from message authentication. In a multi-source exchange, every node presumably has the same value, and the nodes perform a

single-round message exchange to verify that they do in fact have the same value. Each node sends its value to the others, then chooses as the final result the value that the majority of the nodes sent.

This one-round exchange for multi-source messages can tolerate one faulty node for any system with $n \geq 3$ nodes, without needing authenticated messages: in a three-node system, each of the two non-faulty nodes will have its own copy of the message as well as the copy from the other non-faulty node, which makes a majority among three copies. Adding more nodes while still limiting the system to one faulty node obviously makes the majority more overwhelming.

4.1.2 Single-source exchanges

In a single-source (Class 2) exchange in an n-node system, one node wants to send a value to the other $n - 1$ nodes. If the sender is not faulty, then at the conclusion of the exchange, all the recipients must have the value that the sender actually sent. If the sender *is* faulty, all the recipients must at least arrive at the same value at the end of the exchange.

Without message authentication, no protocol can solve this problem in a three-node system. Consider a three-node system consisting of nodes A, B, and C, where node A is the sender. For simplicity, this analysis will assume that the values to be sent are single-bit. In case (i) of Figure 4-1, the sender A is attempting to send the value `1`. Node C is the faulty node, and it insists to node B that A sent a `0`. Since the sender A is not faulty, node B, which is also not faulty, is required to conclude that A sent a `1`.

However, node B has no way of knowing that node C was the faulty node. Had the sender A been faulty instead, as illustrated in case (ii) of Figure 4-1, node B

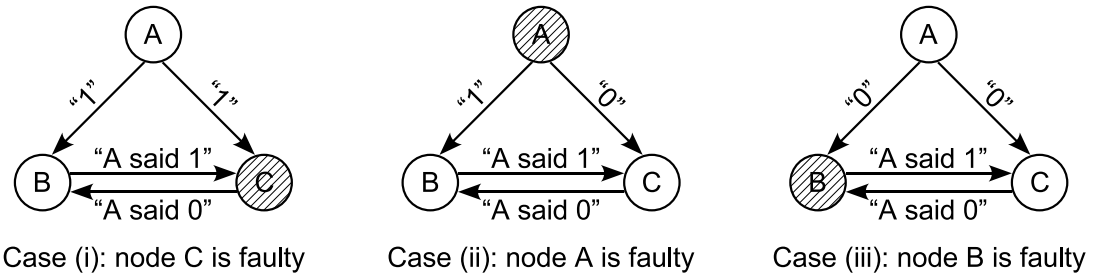

Figure 4-1: A three-node system with a single faulty node.

would have observed exactly the same thing. Therefore, in case (ii), node *B* would also conclude that the sender *A* sent a `1`. Since all the non-faulty nodes in the system must come to the same conclusion, the non-faulty node *C* must also conclude that a `1` was sent in this situation.

Node *C*, in turn, cannot distinguish case (ii) of Figure 4-1 from case (iii), where node *B* is the faulty node. Therefore, in case (iii), node *C* would also conclude that the sender *A* sent a `1`. But this conclusion is wrong, since node *A* was not faulty, and it was actually attempting to send a `0` instead.

These examples show that a three-node system cannot tolerate a single faulty node if the system does not use authenticated messages. A more rigorous proof is presented in [PSL80].

Adding message authentication fixes this shortcoming, and enables a three-node system to tolerate one fault. Using message authentication, a single-source exchange proceeds in two rounds: in the first round, the sender node signs its message, then sends it to all the other nodes. Then in the second round, the recipient nodes reflect the signed message they received to each other. Each recipient node should therefore end up with multiple copies of the message. The recipient nodes use the majority among the copies with valid signatures as the definitive version of the message from

the sender. (If there is no majority, the recipient nodes use a predetermined tiebreaker value instead.) Note that only the original sender node needs to sign the messages that it sends. The recipient nodes do not need to sign the message again when they reflect it to each other.

A case-by-case analysis shows why this protocol is correct. This protocol can tolerate one fault; therefore, either the sender node or one of the recipient nodes can be faulty. If the sender is faulty, then it may send different messages (validly or invalidly signed) to different recipients, or it may fail to send a message at all to some of the recipients. However, since only one faulty node is allowed, a faulty sender node implies that none of recipient nodes is faulty. Therefore, each of the recipient nodes will faithfully reflect the message it receives from the sender to each other. Every recipient node thus ends up with an identical set of messages. Taking the majority among this set gives each recipient node the same final value.

The other case is if one of the recipient nodes is faulty. A faulty node cannot forge messages, so the faulty recipient node can either fail to reflect the message from the sender, or it can reflect an invalidly signed message. The other non-faulty recipient nodes will simply ignore the invalidly signed reflections. Since the sender node is not faulty, the validly signed messages that each non-faulty recipient node ends up with will all be the same, and each recipient node is guaranteed to have at least one validly signed copy of the message. Therefore, when the recipient nodes take the majority among the copies with valid signatures, they will all come to the same value.

4.1.3 Authenticators instead of signatures

Signing and verifying messages using digital signatures is slow. True digital signatures use a different key for the signing and the verification process; as a result, anyone with

the public key can verify signatures, but only the possessor of the secret key can generate them. The cost of this flexibility is that signature schemes require a certain level of mathematical complexity, which makes them slow.

Message authentication codes (MACs) are an alternative to digital signatures. Like a signature, the MAC of a message is a function of the message and a secret key. Thus, a valid MAC can only be produced by someone who knows the secret key. Unlike signatures, however, the same secret key is used to verify the MAC. Because MACs are symmetric-key systems, signing and verifying messages using MACs is orders of magnitude faster than using public-key signatures. The tradeoff is that MACs cannot be used to prove that a message is authentic to a third party who does not know the secret key. Giving everyone a copy of the secret key is not an option either, since everyone would then be able to compute and forge MACs, making them useless for authentication.

One possible solution is to have a separate secret key for every pair of nodes in the system. This would allow any node to send authenticated messages to any other node in the system. If a sender node needs to allow several recipient nodes to verify a message, the sender node can append multiple MACs to the message, one for each recipient node. This vector of MACs is called an *authenticator*. [CL99a] and [CL99b] propose using authenticators instead of public-key signatures in fault-tolerant computer systems.

Unfortunately, authenticators cannot replace public-key signatures in the new version of the FTPP. Specifically, authenticators do not allow a three-node system to tolerate one faulty node. Therefore, authenticators cannot help reduce the number of fault-containment region the FTPP needs.

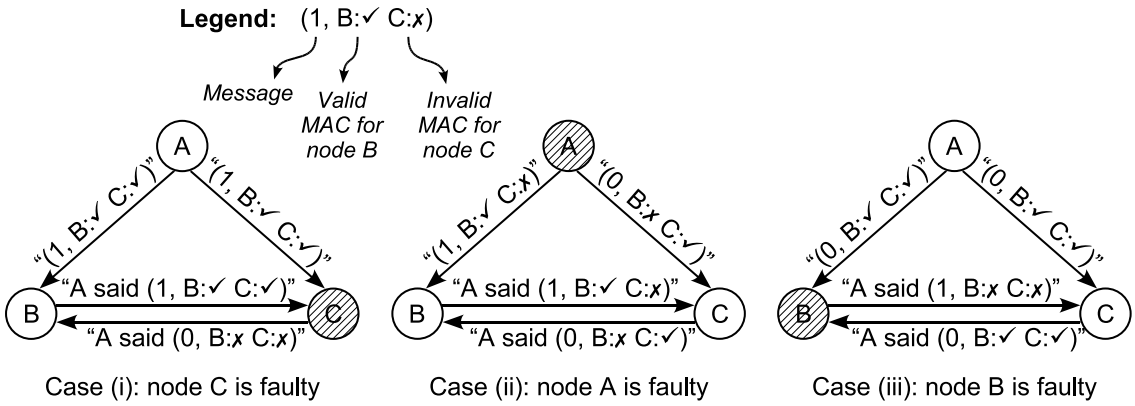

Figure 4-2: A three-node system using authenticators.

The key difference between authenticators and public-key signatures is that an authenticator cannot be used to prove to another party that a message is valid. A faulty sender node could generate an authenticator that contains valid MACs for some of the recipient nodes, but invalid MACs for the others. Thus, different recipients nodes can receive the same message with the same authenticator and come to different conclusions about its validity.

Figure 4-2 illustrates why authenticators are not sufficient for tolerating one faulty node in a three-node system. The system consists of the nodes A, B, and C, where A is the sender node. In case (i), the sender A is non-faulty, and sends a `1` with a valid authenticator to the recipients nodes B and C. Node C is faulty, and it reports to node B that the sender sent a `0` with an invalid authenticator instead. Since node B is non-faulty, it is required to agree with what the sender actually sent; therefore, node B ignores the false report from node C, and concludes that the sender sent a `1`.

In case (ii) of Figure 4-2, the sender node A is the faulty one instead. It sends to each recipient node an authenticator that has a valid MAC for one node, but not for the other. To node B, case (ii) is indistinguishable from case (i). Thus, node B will again

conclude that the sender sent a `1` in case (ii). Since node *C* is also not faulty, it must agree with node *B*; therefore, node *C* also concludes that a `1` was sent. Node *C* in turn cannot distinguish case (ii) from case (iii), so node *C* will conclude that a `1` was sent in case (iii) as well. This conclusion is wrong, however, since the sender *A* was not faulty, and it had actually sent a `0` instead.

4.1.4 What message authentication does not do

The main goal of adding message authentication is to reduce the FTPP from five fault-containment regions to four, and to do this, only the sender node in a single-source exchange needs to sign its messages. If the goal is to also guard against random corruptions to the messages while they are in transit, then the recipient nodes should sign the messages as well when they reflect the messages to each other. However, signing and verifying messages is slow, and thus should be done as sparingly as possible. Other mechanisms can be used to guard against random message corruption instead.

Another job that message authentication does not do is to catch all faulty nodes. Message authentication can detect that a faulty node exists somewhere in the system, but it cannot identify which node it is. If a recipient node claims that it received a message with an invalid signature, there is no way to determine whether the sender node is actually faulty, or whether the recipient node is lying. Message authentication does not give the system any additional power to identify and remove faulty nodes.

4.2 The need for cryptographic security

Because computational power and processor time are so dear in an embedded system, it is very tempting to skimp on the cryptographic security of the message authentication

scheme, and choose a fast but insecure one instead. After all, the nodes in the system are not malicious, and it is inconceivable that they would actively attempt to break the scheme. Perhaps a fault-tolerant system only needs to guard against random alterations to the messages and signatures. Several previous theses have taken this approach, proposing to use cryptographically insecure schemes for message authentication. This section will show that insecure schemes are can often be broken by simple and plausible random faults. They are therefore unsuitable even if the faulty nodes are not actively malicious.

As an example, consider the signature scheme proposed in [St95]. Like most signature schemes, this one follows the classic hash-and-sign paradigm. The hash function is a simple checksum: it splits the message into 32-bit chunks, and computes the 32-bit sum of those chunks. Then the signature function takes the checksum, and multiplies it by the secret key $k \pmod{2^{32}}$, where k is a 32-bit odd number. To verify the signature, the recipient multiplies the signature by the public key $k^{-1} \pmod{2^{32}}$, and compares the result to the checksum of the message.

This signature scheme is certainly very fast. On a typical processor, hashing would take one instruction for every four bytes of the message, and signing would take just a single additional instruction. Verifying the signature is equally quick. However, this scheme is insecure against both message corruption and outright forgery. In fact, it is easy to conceive how a forgery might happen:

Imagine that a fault-tolerant system is performing a single-source message exchange. The sender node has just sent its signed message to all of the recipient nodes, and the recipient nodes are about to reflect the message they received to each other. As it is retransmitting the message, one of the recipient nodes becomes faulty, and a single-bit

error in its processor's cache circuitry causes the node to swap two of the 32-bit words in the message. However, the checksum remains the same for this corrupted message, and the signature therefore remains valid. This failure of the signature scheme violates the requirement that no recipient node can forge a message from the sender node, and causes the message-voting protocol to yield the wrong result.

Perhaps a slightly more secure hash function can fix this problem. [Cl94] proposes using a cyclic redundancy check (CRC) instead of a checksum, while still using modular multiplication for the signature function. A CRC is more robust against message corruption than a checksum [PB61]. It can detect minor random changes to the message with high probability. However, when a node fails, the message corruptions that it produces may *not* be randomly distributed over the space of all possible messages. This means it is incorrect to assume, for example, that a 32-bit CRC has a $1/2^{32}$ probability of missing a corrupted message. Determining the actual probability would require a careful analysis of the CRC implementation itself. For example, in one common version of the CRC algorithm, left-shifting a message by some number of bits causes the CRC of the message to shift by the same number of bits (assuming that the upper bits of the CRC are zero). Therefore, if a node develops a fault that causes it to bit shift received messages and their CRCs before retransmitting them, the node will be able to forge corrupted messages that cannot be detected by the CRCs.

The modular multiplication used as the signature function is not immune to forgeries by random faults either. Since bit shifts are multiplications, modular multiplication suffers from the bit-shift vulnerability as well. A more complex failure is that if a function to compute multiplicative inverses exists somewhere in memory (such a function is needed to generate the key pair), then a faulty node might call the

function on another node's public key by accident, thereby generating the other node's secret key. The faulty node could then use the spuriously generated secret key to forge messages. Even if the key generation function is omitted from memory, the system is still not safe against this mode of failure. The algorithm to compute multiplicative inverses is quite simple, and machine instructions implementing this algorithm may appear within some other function that was written to perform a completely unrelated task. They may even appear inside data. A hardware fault could cause these instructions to be called, and cause a faulty node to forge signatures. Granted, the chance of such a fault happening is extraordinarily low. However, without a thorough analysis of the entire system, it is impossible to say whether the probability is low enough to be negligible.

As the final example of a cryptographically insecure message authentication scheme, consider the one proposed in [Ga90]. This scheme combines hashing and signing into a single function. Like the previous scheme, this scheme hashes the message using a CRC. Rather than using a separate function to sign the hash, however, this scheme instead assigns a different CRC polynomial to each node. The CRC polynomial serves as the key, and it is used for both signing and verification, making this scheme a symmetric signature scheme. Symmetric signatures schemes are particularly vulnerable to forgeries, since every node has the keys needed to forge signatures from every other node. If the nodes store the CRC polynomials in an array, for example, a fault during array index calculations could generate a forgery.

From these examples, it is clear that a Byzantine-resilient fault-tolerant system needs cryptographically secure message authentication. The fear is not that a faulty node might become malicious and deliberately attempt to forge signatures. However, a

signature scheme that is vulnerable to deliberate forgeries is also vulnerable to accidental forgeries. Some of the scenarios that lead to accidental forgeries are more likely than others, but a design that uses cryptographically insecure signatures would have to consider them all, and make sure that the probability of each one happening is sufficiently low. It may not even be possible to enumerate all the possible ways a forged message could be generated.

Using a cryptographically insecure message authentication scheme would require extensive failure-modes and effects analysis (FMEA), which is precisely what a Byzantine-resilient design aims to avoid. A cryptographically secure scheme, on the other hand, promises that there is no conceivable way a faulty node can produce a forged message in any feasible amount of time. No signature scheme is perfect, of course, and there is always a possibility that a node could fail in an extraordinarily unlikely way, generating a successful forgery by sheer luck, but the chances of this happening are much slimmer than the chances of the other ways the system could fail. It is also possible that a faulty node might break a cryptographically secure signature scheme in a way that is currently unknown to any cryptographer, but one hopes that faith in the science of cryptography is justified.

4.3 Message authentication design considerations

A cryptographically secure hash and signature function form the core of a message authentication scheme. However, to design a secure message authentication scheme for communication in a fault-tolerant system, other considerations need to be taken into account as well:

4.3.1 Preventing message replays

Cryptographically secure signatures prevent a faulty node from forging messages. Signatures do not prevent a node from storing a validly signed message from another node and retransmitting it later over and over again, however, since the signature remains valid.

The solution to this problem is to make messages expire so that they cannot be replayed. One way to make messages expire is to include a counter in the messages. The counter increases with each message sent, so no message should ever be repeated. The signature is calculated over the entire message, including the counter, to prevent a faulty node from forging the counter on a replayed message.

A side benefit of adding a counter to the messages is that it helps keep the nodes of the system synchronized. If the counter skips a number, for example, the recipient node can conclude that either it has missed a message or the sender node is faulty.

4.3.2 Randomness

Many signature schemes—including all the ones evaluated in this thesis—need a source of randomness during the calculation of the signature. In some schemes, good randomness is critical. For example, DSA (described in section 5.4) needs a randomly generated nonce for each message to be signed. If the same nonce is ever used for two different messages, the recipient can deduce the signer's secret key. In fact, even if only a few bits of the nonce can be predicted, an adversary can discover the signer's secret key given enough signed messages [BGM97, HS01, NS02].

Generating random numbers is hard. A cryptographically secure random number generator needs to generate a stream of random numbers that an adversary can-

not predict, even after having observed portions of the stream. The standard random number generators supplied with most programming languages and libraries are not cryptographically secure. One option is to add a special hardware device for random number generation, but this option is often infeasible. On desktop computers, cryptographically secure random number generators generally use the timings of interrupts, hard drive head movements, keystrokes, and mouse movements as a source of entropy [ESC05]. Unfortunately, timings are an unsuitable source of entropy in a real-time fault-tolerant system. The clocks in all of the nodes are tightly synchronized with each other, so they can hardly be considered unpredictable. Furthermore, if the system is operating correctly, each of the nodes should receive exactly the same inputs and interrupts. For random number generation in fault-tolerant systems like the FTPP, a different source of entropy is needed.

One solution is to simply include a "pool" of entropy in each node. Each node would be programmed with a small number of random bytes generated externally. These random bytes would serve as the seed to a cryptographically secure expansion function that can output as many new random bytes as the node needs. As long as the nodes do not know the contents of each other's entropy pools, the random numbers generated this way are cryptographically secure and unpredictable.

4.3.3 Collision resistance of hashes

The signature function in most signature schemes expect an input of a fixed length. However, the messages to be signed can be arbitrarily long. In practice, therefore, signature schemes typically hash the messages first, then compute the signatures on the fixed-length hashes.

Like the random number generator and the signature function itself, the hash function also needs to be cryptographically secure. Given the hash of a message, a faulty node must not be able to find another message with the same hash, since the other message with the same hash would be a forgery. Finding a new message that yields a given hash is called a *preimage* attack; all cryptographically secure hash functions must be preimage resistant.

For most applications, there is a second attack on the hash function that must be prevented: the *collision* attack. Even if inverting the hash function for a *given* hash is difficult, it may be easy to find two distinct messages with the same hash, when the desired hash is not fixed. Because of the so-called "birthday paradox" (the counterintuitive result that, in a group of just 23 people, the probability of at least two of them having the same birthday exceeds 50%), finding a collision takes about the square root as much work as finding a preimage, even for cryptographically secure hash functions. This means that a collision-resistant hash must be twice as long as a merely preimage-resistant hash, against an adversary with the same computational power.

For example, one might posit that a faulty node can compute one million hashes per second. Then in one year of operation, the node has a 1/584,942 chance of finding a preimage to a 64-bit hash, which is an acceptably low probability. To achieve the same resistance against collisions, however, the hash would need to be 128 bits long.

Fortunately, for the message-voting process, collision resistance is *not* needed. In a single-source message exchange, if the sender is faulty, it may send different validly signed messages to different recipient nodes. The message-voting process can already deal with this misbehavior, so whether or not the messages have the same hash gives the sender no additional power to undermine the message-voting process. And if a

recipient node is faulty, it still cannot forge signatures from the sender node, so it too has no use for hash collisions. Therefore, the hash function for message authentication does not need to be collision resistant, and thus can use shorter hashes.

That said, there are no cryptographically secure hashes in common use that are only preimage resistant, but not collision resistant.* Therefore, a system that needs a cryptographically secure hash function will have to settle for the longer and somewhat slower collision resistant hashes, whether collision resistance is needed or not.

4.3.4 Key management

In some distributed computer systems where nodes join and leave the system frequently, key management is a major problem. If the nodes do not know each other's public keys beforehand, and if there is no trusted authority to vouch for every node's key, then the nodes must use some key exchange protocol to distribute their keys to each other. However, since a faulty node may try to give different public keys to different nodes, the key exchange protocol itself must be fault tolerant. Therefore, the key management problem turns into another agreement problem.

Fortunately, key management for the FTPP will be much easier. The builders of the system can simply generate the keys for each fault-containment region beforehand, and load them into the processors' memory along with the program code and data.

*Actually, the MD5 hash function [Ri92] is in fact preimage resistant but not collision resistant in practice [WFLY04], but this weakness was not intentional. In any case, although MD5 produces relatively short hashes compared to other popular hash functions, the hashes are still longer than they would be if MD5 only needed to be preimage resistant.

CHAPTER 5

Signature Schemes

A digital signature guarantees that the message it accompanies genuinely came from the sender. True digital signatures are a function of both the message itself and a secret piece of information that only the sender knows. Signatures therefore serve two purposes. Like checksums, CRCs, and hashes, signatures ensure *message integrity*. If the message is modified, the signature will not be valid anymore, and the recipient will detect that the message has been corrupted. Unlike checksums and hashes, however, signatures also guarantee *authenticity*: since only the signer knows its secret key, no one else can forge messages from the signer. Both of these properties are needed in a fault-tolerant computer system that depends on authenticated messages.

This chapter describes the various signature schemes that were considered for the new software-based version of the Fault-Tolerant Parallel Processor (FTPP). RSA and DSA are signature schemes based on the modular exponentiation of large integers. Elliptic curve DSA is an analogue of DSA based on the multiplication of points on an elliptic curve over a finite field. Finally, SFLASH and TTS are signature schemes based on multivariate quadratic equations.

This chapter first gives an overview of how digital signatures work, then proceeds to introduce the signature schemes themselves.

5.1 Overview of digital signatures

The digital signature schemes described here are *public-key signature schemes*. This means that the process of signing a message and the process of verifying the resulting signature use distinct keys. The *secret key* (also called the *private key*) is used for signing, and the *public key* is used for signature verification. The secret key is known only to the signer, whereas the public key is known to everyone who needs to verify the signatures. In a system with multiple nodes, all of which need to sign messages, every node would have its own pair of secret and public keys. One of the requirements for a secure digital signature scheme is that there should be no way for an adversary to deduce the secret key, even if the adversary has access to the public key and a large number of signed messages.

Generically speaking, a digital signature scheme consists of three parts: key generation, message signing, and signature verification. *Key generation* produces a key pair consisting of a secret key and a public key. *Message signing* takes a message and a secret key, and produces the signature. *Signature verification* takes a message, its signature, and a public key as inputs, and indicates whether the signature is valid for the given message and public key.

A secure signature scheme is invulnerable to forgery. Without the secret key, an adversary should not be able to produce any message-signature pair such that, under the corresponding public key, the signature is valid for the message. This forgery resistance implies that digital signatures also ensure message integrity, since a corrupted message with a valid signature would be a forgery. For message authentication in the FTPP, both properties are useful, but forgery resistance is the main goal, and detection of message corruption is a beneficial side effect.

The signing function in most signature schemes typically takes a message of a short, fixed length as the input. This length is usually much shorter than the length of the messages the system actually wants to send. One possible solution is to break the message up into many segments, and sign each segment individually. However, signature functions are slow, so it is best to sign as few times as possible. The solution used in practice in most systems is to *hash* the message first, then use the fixed-length hash as the input to the signature function. To maintain the forgery resistance and message corruption detection properties of the signature, it must be difficult for an adversary to generate a (new) message that has a given hash. Suitable hash functions include MD5 [Ri92] and SHA-1 [FIPS 180-1].

Because all of the signature functions considered here are complex mathematical functions, it is very difficult to say exactly how secure they are. The best estimates are merely educated guesses.* The mathematical objects used in these signature schemes have a lot of structure, so the best attacks are faster than brute force. For example, there are obviously faster ways to factor an integer than trying to divide the integer by every number less than it. Thus, the security level of a signature function is not just a straightforward function of its key size.

The problem of evaluating the security of these signature schemes is made even more complicated by the fact that the schemes are very different from each other, so it is difficult to compare them directly. Therefore, to enable the evaluation of these schemes, their estimated security level is usually expressed as the number of operations needed to break an equally secure (symmetric) encryption function by brute force. For example,

*Note, however, that all of these signature functions are believed to be cryptographically secure. The security level of these schemes, which cryptographers are trying to quantify, is not a choice between "easy to forge" and "hard to forge," but rather a choice between "very hard to forge" and "astronomically hard to forge."

if a signature scheme has a security level of 2^{64}, then forging a signature is about as difficult as breaking an encryption function with 64-bit keys. If a processor can try one million keys per second, then in a year of continuous operation, it has a 1/584,942 chance of finding the correct 64-bit key for the encryption function. Therefore, this 1/584,942 probability is also taken to be the probability of forging a signature in a year's time for a signature scheme with a security level of 2^{64}. This probability is acceptably low, but to include a safety margin, most applications specify a security level of at least 2^{80}. All of the signature schemes considered here claim to meet this level of security.

5.2 Notation

In the signature schemes described here, m generally denotes a message to be signed, and σ is the resulting signature. SK is the secret key, and PK is the public key. A signing function is represented as $\mathsf{sign}_{SK}(m)$, and the signature verification function is $\mathsf{verify}_{PK}(m, \sigma)$. Finally, $\mathsf{hash}(m)$ denotes a hash function applied to a message. The variables in some signature schemes such as DSA have conventional names popularized by standards documents; the presentation here will try to follow those conventions.

When manipulating messages and bit strings, $x \oplus y$ means the exclusive-or of x and y, and $x \,||\, y$ denotes the concatenation of x and y. The length of a string x is denoted by $|x|$.

5.3 RSA

RSA [RSA78], named after its inventors R. Rivest, A. Shamir, and L. Adleman, is a widely-used signature scheme that operates by performing modular exponentiations

on large integers. (RSA can also be used as a public-key encryption scheme.) It is conjectured, but not proven, that forging RSA signatures is as hard as factoring large integers. RSA can use keys of arbitrary length, with longer keys being more secure but slower.

5.3.1 Key generation

RSA performs arithmetic modulo $n = pq$, where p and q are prime numbers. Thus, to generate an RSA key pair, first pick large primes p and q. Conventionally, the size of their product n is considered to be the size of the RSA key. The best known way to forge signatures is to factor n, so n should be large enough to be infeasible to factor. Recommendations for the size of n vary. For a security level of 2^{80}, the recommended size ranges from 760 bits long [Si00] to 1,536 bits long [Pr03a]. Many applications have settled on a key length of 1,024 bits. Since several factoring algorithms have a run time that is a function of the size of the smallest factor, p and q should be approximately the same size for maximum security. However, the difference between p and q should also not be too small, since there are also factoring algorithms whose run time depends on the size of the difference between the factors.

Pick a small integer e relatively prime to both $p-1$ and $q-1$. This is the exponent that will be used for verifying signatures; the public key is thus $PK = (n, e)$. A small e makes signature verification faster, but a very small value such as $e = 3$ may be insecure. For example, when $e = 3$, an adversary can discover half of the bits of the secret key [Bo99]. It is not known whether any of these low-exponent weaknesses allow forgeries in practice, but many implementations choose $e \geq 65{,}537$ to be safe.

Finally, compute d such that $ed \equiv 1 \pmod{\text{lcm}(p-1, q-1)}$. The d that solves

this congruence can be found by the extended version of the Euclidean GCD algorithm. Then the secret key is $SK = d$.

5.3.2 Signing

To sign a message $m < n$, compute the signature $\sigma = m^d \bmod n$. Send σ with the message. The signature is as long as the modulus, so for 1,024-bit RSA, the signature is 128 bytes long. RSA produces the longest signatures among the schemes considered here. Longer signatures take longer to transmit, which is a liability when the amount of time available is limited.

5.3.3 Verification

Compute $\sigma^e \bmod n$, and compare the result to the original message m. The signature is valid if the two are the same.

This verification procedure works because $\sigma^e \equiv (m^d)^e \pmod n$. For every element $a \in \mathbb{Z}_{n=pq}$ (including the ones that are multiples of p or q), the sequence $a, a^2, a^3, \ldots \pmod n$ has a period that is a divisor of $\operatorname{lcm}(p-1, q-1)$. The key generation process picked e and d such that $ed \equiv 1 \pmod{\operatorname{lcm}(p-1, q-1)}$, therefore, $m^{de} \equiv m^{k \operatorname{lcm}(p-1, q-1)+1} \equiv m \pmod n$.

5.3.4 PSS encoding

The RSA signature scheme as described is *not* secure against all types of forgery. Because RSA is multiplicative, that is, $\mathsf{sign}(m_1 m_2) = \mathsf{sign}(m_1)\mathsf{sign}(m_2)$, an attacker can derive new signatures from the signatures of other messages that it has. For example, if an attacker has the message m and its valid signature σ, the attacker can pick an arbitrary

value x and compute the new message $m' = mx^e \bmod n$ and the new valid signature $\sigma' = \sigma x \bmod n$. Even more simply, the attacker can just pick any value $\sigma' \in \mathbb{Z}_n$, then compute $m' = (\sigma')^e \bmod n$, and σ' will be a valid signature of m'.

The cause of this problem is that the attacker can too easily generate messages to fit a signature. The attacker cannot choose the message that it forges, but if it just works backwards from an arbitrarily chosen signature, it will be able to produce *some* forged message. The solution to this problem is to restrict the set of messages that are valid. If the set of valid messages is only a minuscule subset of the set of all possible messages, then the attacker will have essentially no chance of producing a valid message from a randomly chosen signature.

To some extent, the hash-then-sign method described earlier in the overview already implements this solution. To review, under the hash-then-sign method, the "message" that is the argument to the signature function is not the message itself, but rather a hash of the message. That is, using hash-then-sign, the signature is computed as $\sigma = (\mathsf{hash}(m))^d \bmod n$, rather than as $m^d \bmod n$. Since a secure hash function is hard to invert, even if the attacker can forge a valid signature on some hash output $\mathsf{hash}(m)$, the attacker still has no way of actually producing the message m itself. To verify the signature, the recipient computes $\sigma^e \bmod n$, as described before, but compares the result to the hash of the received message instead of to the message itself.

The problem with this simple hash-then-sign scheme is that there is no proof that this change to RSA makes it immune to forgery. The argument just presented, although intuitive, is not a rigorous proof. The concern is mostly theoretical, but without a proof of security, it is conceivable that an attacker could forge signatures, even if the hash function is secure, and even if the RSA problem (that of performing the signing

function without knowing the secret key) is hard to solve.* To produce a provably secure signature scheme, a more complex manipulation of the message, beyond simply hashing the message then signing the hash, is needed.

RSA-PSS is a way of signing messages with RSA that is provably secure against forgery. It applies a more complex regimen of hashing, padding, and salting (adding randomness) to the message to produce a signature scheme that can be proven secure, assuming that the RSA problem is hard, and assuming that the hash function is secure. RSA-PSS is based on, and named after, the Probabilistic Signature Scheme proposed in [BR96]. RSA-PSS itself is described in [PKCS1v2.1], and a proof of its security is given in [Jo01].

Under RSA-PSS as specified in [PKCS1v2.1], the message is first hashed, then prepended with a fixed padding of eight zeros and appended with a randomly generated salt:

$$m' = \texttt{0x00 00 00 00 00 00 00 00} \;||\; \mathsf{hash}(m) \;||\; \mathit{salt}$$

The length of the salt is up to the implementation, but one recommendation is to make it the same length as the hash output.

Next, hash the resulting string again, and use the result as the input to a mask-generation function (MGF), to get the mask: $\mathit{mask} = \mathit{MGF}(\mathsf{hash}(m'))$. The mask-generation function is:

$$\mathit{MGF}(x) = \mathsf{hash}(x \,||\, c) \;||\; \mathsf{hash}(x \,||\, (c+1)) \;||\; \mathsf{hash}(x \,||\, (c+2)) \;||\; \ldots$$

where c is a four-byte counter that begins at zero. The mask-generation function outputs as many bytes as necessary to make $|\mathit{mask}| + |\mathsf{hash}(m')| + 1$ just shorter than the byte length of the RSA modulus.

*More formally, the concern is that, even though inverting hash or finding collisions in it is hard, and even though finding x for a given y such that $y = x^e \bmod n$ is hard, an attacker may nevertheless be able to find some m and σ such that $\mathsf{hash}(m) = \sigma^e \bmod n$, without knowing d.

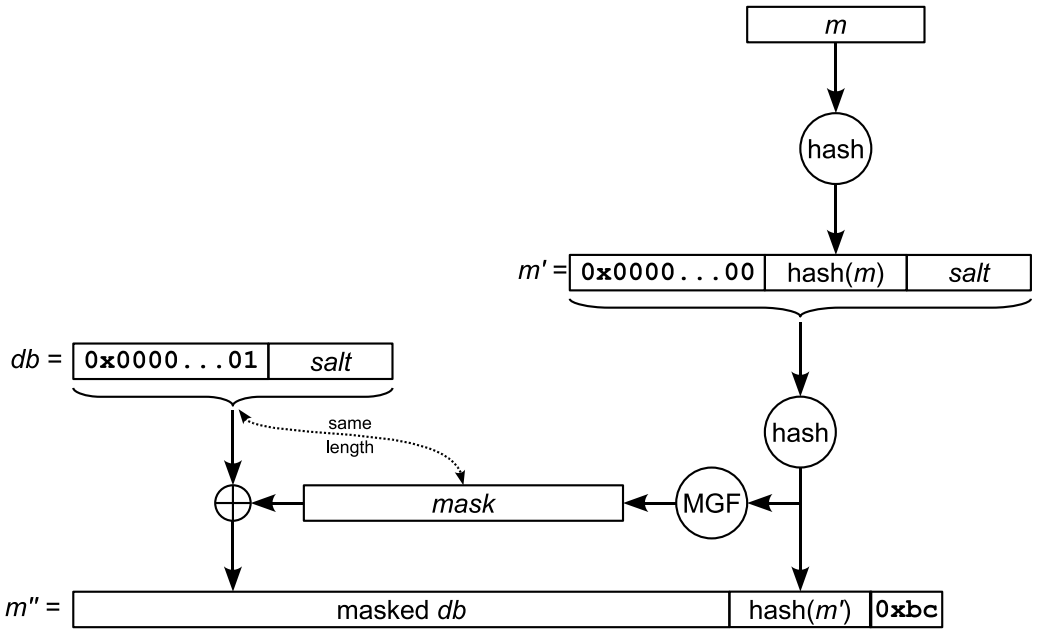

Figure 5-1: PSS encoding (based on illustration from [PKCS1v2.1]).

Then, construct the data block $db = \texttt{0x00 00 . . . 00 01} \,||\, salt$, where *salt* is the same salt as generated earlier. Use as many leading zeros as necessary to make *db* the same length as the mask.

Finally, construct the encoded message $m'' = (db \oplus mask) \,||\, \mathsf{hash}(m') \,||\, \texttt{0xbc}$. (The final byte `0xbc` is just for compatibility with other standards.) This encoded message should be just shorter than the RSA modulus n. The signature is then $\sigma = (m'')^d \bmod n$. Figure 5-1 illustrates the PSS encoding process.

Verifying an RSA-PSS signature is essentially performing the encoding process in reverse. First, use the RSA verification function to recover the encoded message: $m'' = \sigma^e \bmod n$. Check that the final byte of m'' is `0xbc`. The bytes immediately before that are $\mathsf{hash}(m')$. Using $\mathsf{hash}(m')$, recover the mask: $mask = MGF(\mathsf{hash}(m'))$. Since the first bytes of m'' are $db \oplus mask$, xor-ing with the mask again recovers the data

block *db*. Verify that the leading bytes of *db* are zeros, followed by a one. Finally, the last bytes of *db* are the salt. Knowing the salt, hash the received message, and reconstruct $m' = \texttt{0x00 00 00 00 00 00 00 00} \,||\, \mathsf{hash}(m) \,||\, salt$. Hash the reconstructed m' again, and make sure the result is identical to the $\mathsf{hash}(m')$ embedded in the encoded message m''. If the signature passes all of these tests, then it is valid.

5.4 DSA

The Digital Signature Algorithm (DSA) is a public-key signature scheme specified in the U. S. Federal Government's Digital Signature Standard [FIPS 186-2]. Like RSA, DSA operates by performing modular exponentiations on large integers, although the moduli are prime rather than composite. Forging DSA signatures is widely believed to be as difficult as solving the discrete logarithm problem, although this equivalence has not been proven.

5.4.1 Key generation

DSA operates on a subgroup of the multiplicative group of integers modulo a large prime p. The subgroup is generated by an element g, which has order q, a smaller prime number. Solving the discrete logarithm problem in a group whose size is b bits long has roughly the same difficulty as factoring a b-bit number, so a p of a certain size offers about the same level of security as an RSA modulus of the same size [LV01]. Thus, for a security level of 2^{80}, a p of about 1,024 bits is appropriate. The size of q should be about the same as the size of the hash function output; [FIPS 186-2] specifies a size of 160 bits for q, since it mandates the use of the hash function SHA-1, which also has an output of 160 bits.

The size of the signature will be twice the size of q. DSA signatures are significantly shorter than RSA signatures in practice.

The numbers p, q and g are *system parameters* in DSA: they do not necessarily belong to any single user, as they are public and can be reused. To generate them, first pick a prime q. Then find a larger prime p such that $p-1$ is a multiple of q; therefore, $p = aq+1$ for some integer a. Next, find a generator g that will generate the subgroup: pick an $h \in \mathbb{Z}_p^*$ such that $h^a \bmod p > 1$, and let $g = h^a \bmod p$. The subgroup generated by g will have order q.

To generate a user's key pair, pick a random x between 1 and $q-1$, and let $y = g^x \bmod p$. Then the public key is $PK = y$, and the secret key is $SK = x$.

5.4.2 Signing

To sign the message m, first pick a random nonce k between 1 and $q-1$. Then compute:

$$r = (g^k \bmod p) \bmod q \qquad\qquad s = k^{-1}(\mathsf{hash}(m) + x\,r) \bmod q$$

The signature is $\sigma = (r, s)$. In the extraordinarily unlikely event that r or s is zero, pick a new k and recompute the signature. Note that r and s are elements of neither $\mathbb{Z}_p^*$ nor its subgroup generated by g. Instead, r and s can be thought of more as exponents to g.

It is extremely important that k be picked randomly and unpredictably. An adversary who receives two messages with the same k can recover the signer's secret key! Given the two signatures

$$s_1 = k^{-1}(\mathsf{hash}(m_1) + x\,r_1) \bmod q$$

$$s_2 = k^{-1}(\mathsf{hash}(m_2) + x\,r_2) \bmod q$$

the adversary has two equations with two unknowns (k^{-1} and x). The adversary can easily solve for the secret key x. Furthermore, if the adversary can predict even a few

bits of k, [NS02] shows how the secret key can be recovered. An implementation of DSA must take care to use a good random number generator.

RSA is vulnerable to trivial forgeries without PSS encoding. DSA, which already incorporates hashing into the signing process, avoids this problem, and therefore does not need another layer of encoding.

5.4.3 Verification

To verify a signature $\sigma = (r, s)$ on the message m, first check that $0 < r < q$ and that $0 < s < q$. Then compute:

$$u_1 = s^{-1} \mathsf{hash}(m) \bmod q$$

$$u_2 = s^{-1} r \bmod q$$

$$v = (g^{u_1} y^{u_2} \bmod p) \bmod q$$

Accept the signature as valid if $v = r$.

This verification procedure is correct because

$$g^{u_1} y^{u_2} \equiv g^{(\mathsf{hash}(m)/s \bmod q)} y^{(r/s \bmod q)} \equiv g^{\mathsf{hash}(m)/s} y^{r/s} \pmod p,$$

since g has order q. And since $y = g^x$, the equation reduces further to

$$\equiv g^{\mathsf{hash}(m)/s} g^{xr/s} \equiv g^{(\mathsf{hash}(m)+xr)/s} \pmod p.$$

Therefore, $v \equiv (g^{u_1} y^{u_2} \bmod p) \equiv (g^{(\mathsf{hash}(m)+xr)/s} \bmod p) \pmod q$. Next, as computed by the signer, $s = k^{-1}(\mathsf{hash}(m) + xr) \bmod q$, therefore, $(\mathsf{hash}(m) + xr)/s \equiv k \pmod q$. Substituting into v again yields $v \equiv g^k \bmod p \pmod q$. This is equivalent to r.

5.5 Elliptic curve DSA

Elliptic curve DSA (ECDSA) is a variation on DSA that replaces the multiplicative group of integers modulo p with the group of the points on an elliptic curve over a

finite field. The security of ECDSA is conjectured to be based on the difficulty of the elliptic curve version of the discrete logarithm problem. ECDSA is standardized in [ANSI X9.62], and is described in [JMV01].

DSA uses a subgroup of the multiplicative group of integers modulo a prime, and ECDSA uses a subgroup of the group of points on the elliptic curve. ECDSA signatures are about the same length as DSA signatures. Unlike DSA, however, the size of the subgroup in ECDSA is typically not much smaller than the size of the entire group. The entire group of points, in turn, has approximately the same size as the size of the finite field over which the elliptic curve is defined. Therefore, a single number can describe the magnitude of all three quantities. The convention is to just give the magnitude (in bits) of the size of the finite field.

Solving the discrete logarithm problem on an elliptic curve is believed to be much harder than solving the problem with integers. Therefore, ECDSA can use a smaller group size than DSA and still achieve the same level of security. For a security level of 2^{80}, ECDSA with a 160-bit field is estimated to be approximately as secure as DSA with a 1,024-bit p, and as secure as RSA with a 1,024-bit modulus. For the same level of security, ECDSA has smaller key sizes than DSA, and runs faster than RSA [JM98].

5.5.1 Overview of elliptic curve cryptography

Performing cryptography with integers is straightforward, since all the needed mathematical operations are already defined. Performing cryptography on elliptic curves requires more setup. The process for ECDSA is as follows [SEC1]:

1. Select an underlying finite field $\mathbb{F}_q$. If a prime field is chosen, $q = p$ for some

prime p; if a binary field is chosen, $q = 2^m$. (Fields where $q = p^m$, for $p > 2$, also exist, but they are too cumbersome to use in most cases.)

For a prime field $\mathbb{F}_p$, the elements can be represented as the integers modulo p, and arithmetic on the elements is performed as the customary modular arithmetic. For a binary field $\mathbb{F}_{2^m}$, one possible representation is to treat the elements as bit strings of length m, where the bits form the coefficients of a $(m-1)$-degree polynomial. Addition and multiplication are then defined as addition and multiplication of polynomials, except that arithmetic on the coefficients is performed modulo 2. Since the multiplication of two polynomials produces a longer polynomial, the resulting polynomial is shrunk back down by taking the remainder after division with a mth-degree reduction polynomial, which also needs to be specified.

The specification of how to represent the elements of $\mathbb{F}_q$ constitute the *field representation*, denoted *FR*.

2. Define an elliptic curve E on the field. The elliptic curve is the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy the curve equation. For prime fields, the equation has the form $y^2 = x^3 + ax + b$; for binary fields, it has the form $y^2 + xy = x^3 + ax^2 + b$. Both a and b are elements of the field, and the additions and multiplications in the equations are field operations. The number of points that satisfy the curve equation is denoted $\#E(\mathbb{F}_q)$; this quantity has roughly the same magnitude as the size of the field itself.

   The points on the curve become a group when a suitable definition of "addition" is specified for them. First, define a new zero point $\mathcal{O}$, which serves as the identity element for addition. Thus, $\mathcal{O} + \mathcal{O} = \mathcal{O}$, and $P + \mathcal{O} = \mathcal{O} + P = P$,

for any point P. The point $\mathcal{O}$ is not an actual point on the curve itself, but it is needed to make the group closed under addition. For a nonzero point $P = (x, y)$, negation is defined as $-P = (x, -y)$ for prime fields and $-P = (x, x + y)$ for binary fields.

Finally, the addition of two points needs to be defined. Given the two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, where $P_1, P_2 \neq \mathcal{O}$ and $P_1 \neq -P_2$, their sum is $P_3 = (x_3, y_3)$. For a prime field:

$$x_3 = \lambda^2 - x_1 - x_2 \qquad \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1}, & \text{if } P_1 = P_2 \end{cases}$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

where a is one of the constants that parameterizes the elliptic curve equation. For a binary field:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \qquad \lambda = \begin{cases} \frac{y_1 + y_2}{x_1 + x_2}, & \text{if } P_1 \neq P_2 \\ x_1 + \frac{y_1}{x_1}, & \text{if } P_1 = P_2 \end{cases}$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

(The form of the equations given here differs slightly from the more customary presentations that give separate equations for x_3 and y_3 depending on whether $P_1 = P_2$. The form shown here comes from [BJ02].)

Multiplication of a point by a scalar is defined as repeated addition: that is, $kP = \underbrace{P + P + \cdots + P}_{k}$.

3. Given the group consisting of the points on an elliptic curve, pick a base point G that generates a subgroup of size n, where n is prime. Since G generates a subgroup, n must be a divisor of $\#E(\mathbb{F}_q)$, the size of the entire group. Therefore, $\#E(\mathbb{F}_q) = hn$, where h is the *cofactor*. The cofactor is generally small ($h = 1$ is common); thus, the size of the subgroup is about the same as the size of the entire group.

To summarize, the parameters that define an ECDSA system consist of the underlying finite field $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$ of size q, the chosen field representation FR, the elliptic curve equation parameterized by a and b, the generator G, the subgroup size n, and the cofactor $h = \#E(\mathbb{F}_q)/n$.

Unfortunately, generating these system parameters is extremely complex and time consuming. Many implementations therefore choose from a set of standardized pre-generated parameters instead. Recommended elliptic curves and parameters can be found in [RECFGU] and [SEC2].

5.5.2 Key generation

Given the system parameters (q, FR, a, b, G, n, h), pick a random integer d between 1 and $n-1$. Let the point $Q = dG$. Then the user's secret key is $SK = d$, and the public key is $PK = Q$.

5.5.3 Signing

Signing with ECDSA is almost perfectly analogous to signing with DSA, with exponentiation replaced by scalar point multiplication. First, pick a random nonce k between 1 and $n-1$. As with DSA, k must be unpredictable by anyone else. Then to sign the message m, compute:

$$r = (\text{the } x\text{-coordinate of } kG, \text{ converted to an integer}) \bmod n$$

$$s = k^{-1}(\mathsf{hash}(m) + d\,r) \bmod n$$

The signature is $\sigma = (r, s)$. In the unlikely event that r or s is zero, pick another nonce k and try again.

5.5.4 Verification

To verify the signature $\sigma = (r, s)$ on a message n, first check that $0 < r < n$ and that $0 < s < n$. Then compute:

$$u_1 = s^{-1}\,\mathsf{hash}(m) \bmod n$$

$$u_2 = s^{-1} r \bmod n$$

$$V = u_1 G + u_2 Q$$

If $V = \mathcal{O}$, reject the signature. Otherwise, let

$$v = (\text{the } x\text{-coordinate of } V, \text{ converted to an integer}) \bmod n.$$

Accept the signature as valid if $v = r$.

The proof of correctness for this verification procedure follows from the proof of correctness of DSA, by analogy.

5.6 Multivariate quadratic signature schemes

Multivariate quadratic (MQ) cryptography is cryptography based on systems of second-order polynomial equations of many variables. Proposals for cryptographic schemes based on algebraic (as opposed to number-theoretic) systems have existed since at least 1983 ([IM85] surveys some of the early proposals), but none of them have attracted as much attention as the popular schemes like RSA have. MQ signature schemes promise to run many times faster than the other schemes described earlier. However, they have the disadvantage of using very large keys (several kilobytes long), and they are much harder to describe than the number-theoretic schemes. Also, many of the early MQ schemes turned out to be insecure. The following overview of how MQ signature schemes work draws from [WP04], [DS04], and [Wo05].

The public key in an MQ signature scheme consists of the coefficients to a set of polynomial functions of many variables. The signature is the values of those variables. To verify a signature, the recipient applies the polynomial functions to the variables, getting a set of result values. The signature is valid if the resulting values match the received message:

$$PK = \begin{cases} p_0(x) = p_0(x_0, x_1, x_2, \ldots) & m = (m_0, m_1, m_2, \ldots) \\ p_1(x) = p_1(x_0, x_1, x_2, \ldots) & \sigma = (\sigma_0, \sigma_1, \sigma_2, \ldots) \\ p_2(x) = p_2(x_0, x_1, x_2, \ldots) & \text{Let } v = \big(p_0(\sigma),\, p_1(\sigma),\, p_2(\sigma), \ldots\big); \\ \qquad\qquad \ldots & \text{signature valid if } v = m. \end{cases}$$

In an MQ signature scheme, the polynomial equations that make up the public key are second-order equations. This means that each of the functions p_i has the form:

$$p_i(x_0, x_1, x_2, \ldots) = \sum_{j,k \,|\, j \le k} \zeta_{i,j,k} x_j x_k + \sum_j \nu_{i,j} x_j + \rho_i$$

In other words, the polynomial consists of the coefficient ζ times each pair of two variables, plus the coefficient ν times each variable, plus the constant ρ. Note that the variables and coefficients are not integers, but elements of some small finite field $\mathbb{F}$. All of the operations are field operations. The number of coefficients in each equation grows quadratically with the number of variables, and there are many equations, so the public key is quite large.

Solving the equations would allow an adversary to forge signatures. Fortunately, solving a system of multivariate quadratic equations is NP-hard. (The earliest proof of this, for a binary field, is attributed to a private communication from L. G. Valliant and to an unpublished manuscript by A. S. Fraenkel and Y. Yesha; proofs also appear in [Wo05].) This does not imply that forging an MQ signature is NP-hard, of course, since the difficulty of forgery depends on how exactly the coefficients of the

equations are chosen. Nonetheless, the NP-hardness of solving multivariate quadratic equations is evidence that MQ signature schemes may be more secure than schemes based on factoring or the discrete logarithm problem (such as RSA or DSA), since these latter problems have *not* been proven to be NP-hard.

Since solving the equations directly is NP-hard, the signer must have another way of computing signatures. The secret key contains the polynomial functions of the public key, but in a decomposed form that is much easier to invert.

The secret key consists of three *maps*, which when combined compute the same function as the public key. The public key can be thought of as a single function that takes an input vector (the signature) and returns another vector (the message). Each of the three maps are also functions that take and return vectors. Thus, if verify is the function computed by the public key, then

$$\mathsf{verify}(\sigma) = (\phi_3 \circ \phi_2 \circ \phi_1)(\sigma),$$

where ϕ_1, ϕ_2, and ϕ_3 are the three decomposed maps. To sign a message, therefore, the signer works backwards, and computes $\sigma = \phi_1^{-1}(\phi_2^{-1}(\phi_3^{-1}(m)))$.

The maps ϕ_1 and ϕ_3 are *affine*, meaning that each element of the output vector is just a linear combination of the elements in the input vector, plus a constant. Thus, these two maps can be written as a matrix multiplication plus a constant vector:

$$\phi_1(x) = M_1 x + c_1 \qquad\qquad \phi_3(x) = M_3 x + c_3$$

where M_1 and M_3 are matrices, and c_1 and c_3 are vectors. They are usually generated randomly. The input x is also a vector. Again, the matrices and vectors contain field elements, not integers. Since ϕ_1 and ϕ_3 are affine, inverting them is easy:

$$\phi_1^{-1}(y) = M_1^{-1}(y - c_1) \qquad\qquad \phi_3^{-1}(y) = M_3^{-1}(y - c_3)$$

The purpose of ϕ_1 and ϕ_3 is to obscure the operation of the central map ϕ_2. Since ϕ_1 and ϕ_3 are linear, their combination is also linear—and a linear system

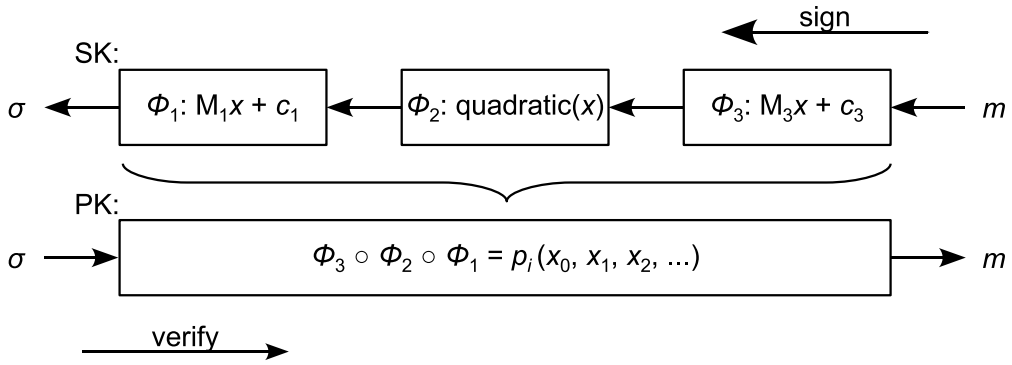

Figure 5-2: Overview of an MQ signature scheme. The secret key consists of the maps ϕ_1, ϕ_2, and ϕ_3, and the public key is the composition $\phi_3 \circ \phi_2 \circ \phi_1$, given as a set of second-order polynomial functions p_i. Signing a message m is done by computing $\phi_1^{-1}(\phi_2^{-1}(\phi_3^{-1}(m)))$. Verifying a signature σ is done by comparing verify(σ) to m, where verify is the function specified by the public key.

of equations would be easy for an attacker to solve. Therefore, ϕ_2 must contribute the nonlinearity that makes the overall signature verification function quadratic. The central map ϕ_2 should be easy for the signer to invert, but when composed with ϕ_1 and ϕ_3 to give the form of the verification function in the public key, the inversion should not be obvious anymore. Different MQ signature schemes use different types of functions for ϕ_2.

Thus, the signer's secret key consists of the matrices and constant vectors that make up the two outer maps, plus the parameters of the central map, which depend on the MQ scheme chosen: $SK = (\phi_1, \phi_2, \phi_3)$. The public key is the composed form of the three maps: $PK = \phi_3 \circ \phi_2 \circ \phi_1$.

Generating the public key from the secret key is not hard. Again, the public key consists of a vector of polynomial functions with the form:

$$p_i(x_0, x_1, x_2, \ldots) = \sum_{j,k \,|\, j \leq k} \zeta_{i,j,k} x_j x_k + \sum_j \nu_{i,j} x_j + \rho_i$$

To find the values of the coefficients, first find the ρ's by computing $\phi_3 \circ \phi_2 \circ \phi_1$ on

the zero vector $x = (0, 0, 0, \ldots)$. The result is a vector giving ρ_i for every i. Finding the ν's and ζ's is slightly more complicated. Setting the jth element of the input vector to 1 will yield $\zeta_{i,j,j} + \nu_{i,j} + \rho_i$. Setting the jth element of the input vector to a different value k will yield $\zeta_{i,j,j}k^2 + \nu_{i,j}k + \rho_i$. From these two results, the value of $\zeta_{i,j,j}$ and $\nu_{i,j}$ can be solved for. Finally, find the values for $\zeta_{i,j,k}$ where $j \neq k$ by setting every pair of elements in the input vector to 1, one pair at a time.

Figure 5-2 summarizes the structure of an MQ signature scheme.

5.7 SFLASH

SFLASH is an MQ signature scheme designed to run on devices with limited computational power, such as smart cards. SFLASH claims to offer a security level of 2^{80}, while running much faster than an 1,024-bit RSA. The version of SFLASH described here is the second version of SFLASH [CGP02], called SFLASH^{v2} by its inventors.

SFLASH uses the finite field $K = \mathbb{F}_{128}$, specified as $\mathbb{F}_2[X]/(X^7 + X + 1)$. The vectors in SFLASH are 37 elements long. Therefore, M_1 and M_3 are 37×37 square matrices, and c_1 and c_3 are 37-element vectors, all filled with elements from K. The signature vector σ is also 37 elements long, so SFLASH has a signature length of 259 bits. The public key, however, is truncated after the first 26 polynomial functions. Thus, the verification function given by the public key is a function that maps $K^{37} \rightarrow K^{26}$. Leaving out the last 11 polynomials from the public key presumably makes the system more secure, since it gives the attacker fewer equations to work with. It also makes the public key shorter.

The central map ϕ_2: $K^{37} \rightarrow K^{37}$ adds the nonlinearity that makes SFLASH quadratic. It is defined somewhat unusually:

First, let L be the extension field $K[Y]/(Y^{37} + Y^{12} + Y^{10} + Y^2 + 1)$, where $K = \mathbb{F}_{128}$ is the field specified earlier. In other words, an element $x \in L$, where $x = (x_0, x_1, \ldots, x_{36})$, represents the coefficients of a 36$^{\text{th}}$-degree polynomial, and each coefficient is itself a member of K. Addition and multiplication in L are defined as the addition and multiplication of the polynomials, with the arithmetic on the coefficients performed in K. Since multiplying two 36$^{\text{th}}$-degree polynomials yields a 72$^{\text{nd}}$-degree polynomial, the resulting polynomial is reduced modulo the reduction polynomial $Y^{37} + Y^{12} + Y^{10} + Y^2 + 1$ to bring it back into L. Exponentiation is defined as repeated multiplication.

Note that a vector from K^{37} can be mapped trivially to an element from L, and vice versa: the ith element in the 37-element vector simply becomes the ith coefficient in the 36$^{\text{th}}$-degree polynomial. In an implementation, of course, both objects would be stored as an array of values, and no actual conversion is necessary.

Given these preliminaries, the central map in SFLASH is defined as:

$$F(A) = A^{128^{11}+1}$$

where A is an element in L. That is, the central map ϕ_2 takes a vector from K^{37}, converts it into an element $A \in L$, computes $A^{128^{11}+1}$, then returns the result converted back into another 37-element vector from K^{37}.

Since outer maps ϕ_1 and ϕ_3 are linear, the central map ϕ_2 must be quadratic to make the composition $\phi_3 \circ \phi_2 \circ \phi_1$ be quadratic. Therefore, $F(A)$ must scramble up the coefficients of A quadratically: if $B = F(A)$, where $A = a_{36}Y^{36} + a_{35}Y^{35} + \cdots + a_0$, and $B = b_{36}Y^{36} + b_{35}Y^{35} + \cdots + b_0$, then it must be the case that $b_i = \sum_{j,k} c_{j,k} a_j a_k$ for some constants $c_{j,k}$. But why should $F(A)$ be quadratic at all? Conceivably, the coefficients b_i could end up as any arbitrary function of the a_i's.

First, consider A^2, where again $A = a_{36}Y^{36} + a_{35}Y^{35} + \cdots + a_0$. Then $A^2 = \sum_i a_i^2 Y^{2i} + \sum_{i \neq j} 2a_i a_j Y^i Y^j$ (before reduction modulo reduction polynomial). Since the coefficients are in K, however, and since $K = \mathbb{F}_{128}$ is a binary field, $2a_i a_j = 0$. Therefore, all the $Y^i Y^j$ terms where $i \neq j$ disappear, leaving $A^2 = \sum_i a_i^2 Y^{2i}$. And since raising something to the 128th power is just repeated squaring, $A^{128} = \sum_i a_i^{128} Y^{128i}$. But $a_i^{128} = a_i$ in $\mathbb{F}_{128}$, leading to the result $A^{128} = \sum_i a_i Y^{128i}$. This result has not been reduced yet, but reduction modulo the reduction polynomial is linear with respect to the coefficients. Therefore, A^{128} *yields a linear transformation of the coefficients.*

Finally, $F(A) = A^{128^{11}+1} = A^{128^{11}}A$. Since $A^{128^{11}}$ is a linear transformation of the coefficients, multiplying by A again gives a quadratic transformation of the coefficients. Therefore, the central map $F(A)$ is quadratic.

5.7.1 Key generation

The secret key consists of the matrices and column vectors that define the outer maps $\phi_1(x) = M_1 x + c_1$ and $\phi_3(x) = M_3 x + c_3$. So to generate the secret key, pick a pair of random 37×37 matrices for M_1 and M_3, and a pair of random 37-element vectors for c_1 and c_3. Since signing involves the the computation of ϕ_1^{-1} and ϕ_3^{-1}, the matrices M_1 and M_3 must be invertible. The easiest way to generate them is to fill them with random values and attempt to invert them, repeating until invertible matrices are obtained.

The public key consists of the coefficients of the first 26 functions of the composition $\phi_3 \circ \phi_2 \circ \phi_1$, so $PK = \left(\zeta_{i,j,k} \mid (0 \leq j \leq k, 0 \leq k < 37), \nu_{i,j} \mid (0 \leq j < 37), \rho_i\right)$, for all $0 \leq i < 26$. The secret key is $SK = (M_1, c_1, M_3, c_3)$.

Note that since the last 11 polynomial functions of the composition $\phi_3 \circ \phi_2 \circ \phi_1$ are truncated from the public key, the last 11 rows of M_3 and the last 11 elements of c_3

do not affect the validity of the signatures. Also, [GSB01] points out that c_1 and c_3 can be deduced from the public key; consequently, the designers of SFLASH call these two vectors "semi-public," as they do not actually need to be kept secret [CGP03].

5.7.2 Signing

To sign a message m, first pick a random nonce $r \in K^{11}$. In other words, r is a random vector of 11 elements from the field $K = \mathbb{F}_{128}$. (The SFLASH specification in [CGP02] includes a secret string Δ in the secret key, and uses a series of hashes to combine Δ and m to yield the nonce r. However, it is easier to just randomly generate a new r for each signature.)

Next, hash the message: $\mathsf{hash}(m) \to h$, where $h \in K^{26}$. SFLASH, like DSA, specifies hashing the message as part of the signing process. Since the hash is 26 elements long, and the nonce is 11 elements long, the concatenation $h \,||\, r$ is a vector in K^{37}.

The signature is computed as $\sigma = \phi_1^{-1}(\phi_2^{-1}(\phi_3^{-1}(h \,||\, r)))$. The signature is also a vector in K^{37}. Since ϕ_1 and ϕ_3 are affine, inverting them is straightforward. Inverting the central map ϕ_2 is a more interesting process. The central map is implemented as $F(A) = A^{128^{11}+1}$, where $A \in L$. Since the size of L is 128^{37}, its multiplicative group has size $128^{37} - 1$. Therefore, the inverse of F is $F^{-1}(B) = B^{(128^{11}+1)^{-1} \pmod{128^{37}-1}}$.

5.7.3 Verification

To verify the signature $\sigma \in K^{37}$ on the message m, compute $v = \mathsf{verify}(\sigma)$, where verify is the set of verification functions described by the coefficients in the public key. Since the public key is truncated after the first 26 polynomial functions, v will be only 26 elements long. However, these 26 elements should exactly match the hash of the message,

since the signature was computed on $\mathsf{hash}(m) \,||\, r$, and $\mathsf{hash}(m)$ gave a 26-element hash. Accept the signature as valid if $v = \mathsf{hash}(m)$.

5.8 TTS

The Tame Transformation Signature (TTS) scheme is an MQ signature scheme that uses a *"tame-like"* map for its central map. A tame-like map $y = f(x)$, where x and y are vectors of elements from some field, is easy to invert for explicit values for y, but giving an expression in symbolic form for the inverse map f^{-1} is difficult because it would have too many terms [CY03]. TTS claims to be even faster than SFLASH, while still offering a 2^{80} level of security.

Several versions of TTS have been proposed ([CYP02], [CY03], [YCC04]), but these early versions of the signature scheme suffer from various cryptographic weaknesses [YC04, DSY06]. The version of TTS described here is called Enhanced TTS (20,28) or TTS/5, and is specified in [YC05]. At the time of this writing, this new version of TTS is believed to be secure.

TTS uses the field $K = \mathbb{F}_{256}$. Like SFLASH, the output vector of the verification function specified by the public key is shorter than the signature vector, to make the quadratic equations harder to solve. Whereas SFLASH accomplishes this shortening by simply reducing the number of polynomial functions revealed in the public key, TTS shortens the output in the central map itself. The input to the verification function $\mathsf{verify}(\sigma) = (\phi_3 \circ \phi_2 \circ \phi_1)(\sigma)$ is 28 elements long; the output is 20 elements long, hence the name TTS (20, 28). The dimensions of the three maps are therefore:

$$\phi_1 : K^{28} \to K^{28} \qquad \phi_2 : K^{28} \to K^{20} \qquad \phi_3 : K^{20} \to K^{20}$$

The dimensions of the maps in TTS are smaller than in SFLASH, which is one of the reasons TTS runs faster.

Following the general pattern of MQ signature schemes, TTS specifies affine transformations for the outer maps: $\phi_1(x) = M_1 x + c_1$ and $\phi_3(x) = M_3 x + c_3$. The only wrinkle here is that M_1 and M_3 have different sizes, as do c_1 and c_3.

The central map is more interesting. If $x = (x_0, x_1, \ldots, x_{27})$ is the input vector, then $y = \phi_2(x)$ is defined as:

$$\begin{cases} y_i = x_i + \sum_{j=1}^{7} \alpha_{i,j} x_j x_{8+(i+j \bmod 9)}, \text{ for } i = 8 \ldots 16 \\ y_{17} = x_{17} + \alpha_{17,1} x_1 x_6 + \alpha_{17,2} x_2 x_5 + \alpha_{17,3} x_3 x_4 \\ \qquad + \alpha_{17,4} x_9 x_{16} + \alpha_{17,5} x_{10} x_{15} + \alpha_{17,6} x_{11} x_{14} + \alpha_{17,7} x_{12} x_{13} \\ y_{18} = x_{18} + \alpha_{18,1} x_2 x_7 + \alpha_{18,2} x_3 x_6 + \alpha_{18,3} x_4 x_5 \\ \qquad + \alpha_{18,4} x_{10} x_{17} + \alpha_{18,5} x_{11} x_{16} + \alpha_{18,6} x_{12} x_{15} + \alpha_{18,7} x_{13} x_{14} \\ y_i = x_i + \alpha_{i,0}\, x_{i-11}\, x_{i-9} + \sum_{j=19}^{i-1} \alpha_{i,j-18}\, x_{2(i-j)-(i \bmod 2)}\, x_j + \alpha_{i,i-18}\, x_0\, x_i \\ \qquad + \sum_{j=i+1}^{27} \alpha_{i,j-18}\, x_{i-j+19}\, x_j, \text{ for } i = 19 \ldots 27 \end{cases}$$

The coefficients $\alpha_{i,j} \in K$ are part of the secret key. Notice that i starts at 8, following the convention in [YC05], making the output vector $y = (y_8, \ldots, y_{27})$ 20 elements long. In this central map, each of the y_i's depends on only a few of the x_i's from the input vector, which makes the central map easy to invert. However, the surrounding maps ϕ_1 and ϕ_3 will mix the x_i's so that each of the y_i's will depend on all of the x_i's.

This central map is quadratic with respect to the the elements of the input vector, so the composite map $\phi_3 \circ \phi_2 \circ \phi_1$ also quadratic. In contrast to SFLASH, TTS is a "truer" multivariate quadratic signature scheme: whereas SFLASH's central map relies on the properties of an extension field of K to scramble the input vector

quadratically, the central map in TTS directly computes a quadratic transformation of the elements in the input vector.

5.8.1 Key generation

The secret key consists of the parameters for the outer maps $\phi_1(x) = M_1 x + c_1$ and $\phi_3(x) = M_3 x + c_3$, as well as the coefficients $\alpha_{i,j}$ for the central map. To create a key pair for TTS, generate a random invertible 28×28 matrix for M_1, a random 28-element vector for c_1, and a random invertible 20×20 matrix for M_3, all with elements from $K = \mathbb{F}_{256}$. Do not generate the vector c_3 yet. Also choose random values for the $\alpha_{i,j}$'s in the central map ϕ_2.

As a small optimization, TTS specifies that the polynomial functions in the public key should have no constant terms—that is, $\rho_i = 0$ for all i. This optimization reduces the size of the public key, and makes signature verification slightly faster. The vector c_3 is used to make the constant terms disappear: computing $\phi_3 \circ \phi_2 \circ \phi_1$ on the zero vector yields the vector of ρ_i's, so set c_3 to make the ρ_i's be zero.

The public key is $PK = \left(\zeta_{i,j,k} \mid (0 \le j \le k, 0 \le k < 28), \nu_{i,j} \mid (0 \le j < 28)\right)$, for $0 \le i < 20$. The secret key is $SK = (M_1, c_1, M_3, c_3, \alpha_{i,j})$, for $8 \le i < 28$, with the range of j varying for each i.

5.8.2 Signing

To sign a message m, compute the signature $\sigma = \phi_1^{-1}(\phi_2^{-1}(\phi_3^{-1}(\mathsf{hash}(m))))$, where hash is a hash function that returns a vector in K^{20}. The outer maps ϕ_1 and ϕ_3 are affine, so inverting them is simple.

Inverting ϕ_2 requires inverting the tame-like central map. That is, given a vector $y = (y_8, \ldots, y_{27})$, the signer needs to find a vector $x = (x_0, \ldots, x_{27})$ such that

$y = \phi_2(x)$. The inversion is performed in parts. First, pick random values for $x_1, \ldots, x_7$, and solve for $x_8, \ldots, x_{16}$. This involves solving a system of nine linear equations for nine unknowns, which can be done by Gaussian elimination. There is a chance that the randomly-chosen values for $x_1, \ldots, x_7$ will give a system of equations that has no solution (or too many), in which case, a new set of random values will have to be chosen.

Next, substitute the values for $x_1, \ldots, x_{16}$ into the equations for y_{17} and y_{18}, and solve for x_{17} and x_{18}. These two equations will always be solvable. Finally, choose a random value for x_0 and solve for the nine remaining variables $x_{19}, \ldots, x_{27}$. Here again, a bad choice for x_0 will render the equations unsolvable, in which case, a new value for x_0 will need to be tried.

Solving nine simultaneous equations can be done relatively quickly, so signing with TTS is fast. The fact that the signer will occasionally generate a set of unsolvable equations and need to retry is problematic, however. In a real-time system, operations need to happen in a fixed amount of time, even in the worst case. Unfortunately, the worst-case time for computing a TTS signature is infinite, since the signer may conceivably be unlucky enough to perpetually choose values that make the equations unsolvable.

However, the speed advantages of TTS are too compelling to immediately dismissing this signature scheme. As [YC05] points out, if all but one of the variables to be randomly chosen are fixed, then at most 9 out of the possible 256 values for the remaining variable will make the resulting system of equations unsolvable. Therefore, the signer generates an unsolvable system of equations 9/256 of the time, at most. In practice, a randomly-generated system of equations is solvable about 255/256 of the time.

Thus, although the probability of failing to generate a signature never becomes zero, it can be made arbitrary small by budgeting more time for the signature process. And unlike some of the other possible failures in a fault-tolerant system, the failure probability of the signature process is fixed and known. Furthermore, failing to compute the signature on time only affects the signer node; the failure does not disable the message-voting protocol for the other nodes.

To conclude, the fact that the signature process takes a non-deterministic amount of time is indeed a shortcoming, but it is a shortcoming that can be strictly quantified and managed.

5.8.3 Verification

In contrast to the signing process, the signature verification process is completely deterministic, and takes a fixed amount of time. The verification process for TTS is the same as the one for SFLASH. To verify the signature vector σ on the message m, compute $v = \mathsf{verify}(\sigma)$, where verify is the verification function specified by the coefficients in the public key. Accept the signature as valid if $v = \mathsf{hash}(m)$.

CHAPTER 6

Implementation and Results

Adding authenticated messages to a fault-tolerant computer system allows the system to achieve the same level of reliability with fewer nodes, making the system less costly and reducing the number of components that can fail. A system like the X-38 Fault-Tolerant Parallel Processor (FTPP) needs to send data at a very high rate; thus, the biggest constraint on a message authentication implementation is its speed. Although it is possible to estimate how fast a certain signature scheme will run by analyzing the algorithms it uses, the only reliable way to evaluate its speed is to actually implement it and time it. Therefore, this thesis implements and benchmarks each of the signature scheme candidates.

This chapter describes the implementation of RSA-PSS, DSA, elliptic curve DSA, SFLASH, and TTS. It explains the optimizations that were used, and reveals how long each of these signature schemes takes to sign a message.

6.1 Existing results

One of the more comprehensive surveys of cryptographic performance is [Pr03b] from the NESSIE project, which gives benchmarks for various signature schemes. Unfortunately, the results it gives are somewhat hard to interpret, as they are a combination

Scheme	Signing time (ms)	Verification time (ms)	Source
RSA-PSS	40.5	4.5	[Pr03b, table 13]. 1,024-bit n, $e = 3$. Normalized from a Pentium Celeron running at 450 MHz.
DSA	7.9	17.4	[Da04]. 1,024-bit p. Signing uses precomputation. Normalized from a Pentium 4 running at 2.1 GHz.
ECDSA	16.9	22.7	[Pr03b, table 37]. 163-bit binary field. Normalized from cycle count on Pentium 3.
SFLASH	4.5	1.3	[Pr03b, table 13]. Normalized from a Pentium 3 running at 500 MHz.
TTS	0.1	0.2	[YC05]. Normalized from a Pentium 3 running at 500 MHz.

Table 6.1: Performance of signature schemes from various previously published benchmarks. Times are normalized to a processor running at 300 MHz. Because these benchmarks were performed on different architectures, these results are only useful for a very rough comparison of the signature schemes.

of cycle counts for various processor/compiler combinations and submitter-supplied timings on heterogeneous architectures. Also, DSA was not evaluated. Another source of performance data is [Da04], which gives benchmarks for the cryptographic primitives implemented in the Crypto++ library. Finally, the papers proposing the various signature schemes often make performance claims as well.

Although comparing benchmarks performed on different architectures and obtained from different sources is dubious, having a baseline for evaluating the implementations described in this chapter is still useful. Table 6.1 presents the signature scheme timings collected from the various sources. Each of the signature schemes in the table claims a security level of at least 2^{80}. The timings in this table are normalized to a processor running at 300 MHz by simply scaling the timings from the sources according to the frequencies of the processors used by the sources. The frequency of

300 MHz was chosen because the processor used for the implementations described in this chapter also runs at 300 MHz.

The existing benchmarks for the five signature schemes evaluated here come from different sources that used different processors, compilers, and programming languages. As a result, the timings in Table 6.1 give only a very rough comparison of how the signature schemes perform. One of the contributions of this thesis is that it will present benchmarks of the all of the signature schemes on a single platform, allowing a more valid comparison of the signature schemes.

6.2 Testing platform

The signature scheme implementations described in this chapter were tested on an Embedded Planet 405 single-board computer [EP405] with a PowerPC processor running at 300 MHz. This processor was chosen because it is a good representative for the level of processing power that will be available in the next few years in radiation-hardened processors. The computer runs the INTEGRITY real-time operating system [INTEG] from Green Hills Software. The implementations of the signature schemes were compiled with an optimizing compiler [GHSC] also from Green Hills Software.

RSA and DSA were implemented using the large-integer support from the GNU Multiple Precision (GMP) arithmetic library [GMP]. The GMP library uses assembly-language routines for its core operations. Since GMP lacks support for elliptic curve arithmetic, however, ECDSA was implemented using the MIRACL multiprecision library [MIRACL] instead, which unfortunately does not offer assembly-language optimizations for the PowerPC. SFLASH and TTS were implemented using a finite field arithmetic library written by the author.

Although complete implementations for some of the more popular signature schemes like RSA already exist, they were not used in this evaluation for several reasons. Signature schemes like RSA and DSA are simple enough that implementing them using a large-integer library is not much harder than trying to port an existing implementation to the test platform. Furthermore, the easiest-to-port implementations are written in plain C, whereas the implementations developed for this thesis can take advantage of the assembly-language routines in the large-integer library. Finally, implementing the signature schemes from scratch makes it easier to test the effects of different optimizations.

6.3 Methodology

For each signature scheme, the key generation, signing, and signature verification processes were implemented. Once loaded onto the test board, the software would proceed to generate a new key pair, then use it to sign and verify messages, repeating the process enough times to get an accurate measurement of how long the signing and verification processes take. (In a production system, the keys would have been generated offline instead and hardcoded into the board, but for testing, it was easier to generate the keys on the board directly.) The messages to be signed were usually 1 KB–long strings of random data. The elapsed times were measured using the high-resolution timer on the test board, which claims to have a resolution of 2.3 ns or better.

6.3.1 Hash performance

All of the signature schemes that were tested specify hashing the message as part of the signing process. Therefore, all of the timings reported here include the time it

Message length (bytes):	128	256	384	512	640	768	896	1,024
Time (μs):	8.59	13.74	18.88	24.03	29.18	34.32	39.47	44.62

Table 6.2: SHA-1 performance on messages of various lengths. (Note that the times are in microseconds, not milliseconds.)

takes to hash the message as required by the signature scheme. Since many of the schemes called for a 160-bit hash output, the SHA-1 hash function [FIPS 180-1] was used in the implementations of all of the schemes. The SHA-1 implementation used was borrowed from an assembly-language implementation [Ma05] submitted to the Git version control system project.

In general, the signing and verification functions take many times longer than hashing the message, so the hashing time is insignificant, and the schemes would have performed about the same had a different hash function been chosen. Table 6.2 shows how long SHA-1 takes to hash messages of different lengths on the test platform.

6.3.2 Random number generation

All of the signature schemes tested require some source of randomness during the message signing process. An unpredictable source of randomness is crucial for the security of many of the signature schemes. As section 4.3.2 of this thesis explains, since the nodes in a synchronized replicated system have very few sources of unpredictable randomness available to them, the best way to generate random numbers is to include an "entropy pool" in each node, and to use a cryptographically secure expansion function to turn the pool into as many bytes of randomness as needed.

For the implementations described in the chapter, an entropy pool was not used. Instead, the implementations simply relied on the `rand()` function from the C standard library. The actual production system would use an entropy pool, but

a cryptographically secure expansion function would not be very much slower than `rand()`, so the benchmarks presented in this chapter would still be applicable.

6.4 RSA-PSS implementation

The most time-consuming part of the RSA algorithm is performing the modular exponentiations. Each exponentiation is implemented using the square-and-multiply algorithm (or using a sequence of multiplications based on the addition chain for the exponent), and each modular multiplication in turn involves a large-integer multiplication followed by a large-integer division.

Because modular exponentiation is so slow, one common optimization is to speed up the computation of the signature using the Chinese remainder theorem. Since p and q are relatively prime, one can compute $\sigma_p = (m^d \bmod p) = (m^{d \bmod (p-1)} \bmod p)$ (due to Fermat's little theorem) and $\sigma_q = (m^d \bmod q) = (m^{d \bmod (q-1)} \bmod q)$, then combine them into $\sigma = m^d \bmod pq$ using the Chinese remainder theorem. This optimization cuts the size of both the modulus and the exponent in half, giving a pretty drastic speed improvement.

However, one of the disadvantages of using this optimization is that if either σ_p or σ_q is miscalculated, the recipient of the signature can deduce the factorization of n, and hence learn the signer's secret key [Le96]. If σ_p is miscalculated but σ_q is not, for example, then $\sigma^e \not\equiv m \pmod{p}$, but $\sigma^e \equiv m \pmod{q}$. Therefore, $\sigma^e - m$ is a multiple of q but not of p, which means that $\gcd(\sigma^e - m)$ reveals a factor of n.

Fortunately, in an environment like the FTPP, such a vulnerability is less of a worry. Even if one node does miscalculate the signature, it is extraordinarily improbable that a second node would try to exploit the fault—and in any case, the

Modulus size (bits):	256	512	768	1,024
Signing time (ms):	0.87	4.37	11.38	24.69
Verification time (ms):	0.17	0.44	0.80	1.31

Table 6.3: RSA-PSS performance with moduli of various sizes, $e = 65{,}537$, using the Chinese remainder theorem optimization.

FTPP only needs to tolerate one fault at a time. Thus, the Chinese remainder theorem optimization can safely be implemented.

Note that the Chinese remainder theorem optimization can speed up message signing, but the same optimization cannot be used for the verification process, since the recipient does not know the factors p and q. Therefore, it makes sense to make the verification exponent e smaller than the signing exponent d.

The benchmarks described here tested RSA with a variety of modulus sizes. Most applications today that need cryptographic security use a modulus of at least 1,024 bits. A modulus of 512 bits can be factored on a large distributed computer system, and a modulus of 256 bits can be factored in a under an hour on a single desktop computer. A modulus shorter than 160 bits would have trouble accommodating the entire SHA-1 hash and the PSS encoding. Table 6.3 gives the performance of the RSA signature and verification functions using moduli of different sizes.

6.5 DSA implementation

The DSA implementation used a q of 160 bits, to match the SHA-1 hash function. The size of p varied. Solving the discrete logarithm problem for a p of a certain size is approximately as difficult as factoring an RSA modulus of the same size, so the discussion on the security of RSA moduli of different sizes applies to the of p in DSA as well.

One possible optimization for DSA is the precomputation technique described in [FIPS 186-2]. The DSA signature is $\sigma = (r, s)$, where $r = (g^k \bmod p) \bmod q$, with k being a randomly-chosen integer. Since r and k do not depend on the message, an implementation can precompute a large set of r's and k's (as well as the inverses of the k's, which will be needed for computing s). However, this precomputation optimization is not practical for a system like the FTPP, since a node in the system would quickly use up all the precomputed values that any reasonable amount of memory can store. Therefore, this optimization was not implemented for the benchmarks.

Another form of precomputation tries to speed up the exponentiation computation itself. For example, the computation of a^b, for a fixed base a and a variable exponent b, can be sped up by precomputing $\{a^2, a^4, a^8, \ldots\}$. Then the value of a^b can by computed by simply multiplying together the appropriate precomputed powers of a according to the binary expansion of b. This technique is faster than exponentiation by squaring and multiplying, since using the precomputed powers is equivalent to eliminating the squaring steps. It is possible to extend this technique and use a larger set of precomputed powers, trading off space for speed [BGMW92].

As part of the signature verification process, the recipient needs to compute $(g^{u_1} y^{u_2} \bmod p) \bmod q$, where y is the signer's public key. To speed up this computation, the recipient could precompute the powers of y as well. However, doing so would require a lot of memory, since the recipient would have to precompute the powers of y for every possible signer. Instead, a different optimization can be applied to the verification process: a variation on the square-and-multiply algorithm can compute the product of two exponentiations about as quickly as computing a single exponentiation.

The classic square-and-multiply algorithm to compute a single exponentiation a^b, for $a \neq 0$, is:

Size of p (bits):	256	512	768	1,024
Signing time (ms):	0.93	2.41	4.39	6.98
Verification time (ms):	2.81	7.31	13.18	23.71

Table 6.4: DSA performance with p of various sizes, using precomputed powers of g and Straus's algorithm.

$r \leftarrow 1$
for i **from** msb(b) **down to** 0:
    $r \leftarrow r^2$
    if $b_i = 1$ **then** $r \leftarrow r \cdot a$
end for
return r

where msb(b) gives the index of the most-significant bit of b, and where b_i denotes the ith bit of b, with $i = 0$ being the least-significant bit.

Straus's algorithm [St64] allows $a_1^{b_1} a_2^{b_2} \cdots a_n^{b_n}$ to be computed using just one pass over all of the exponents simultaneously. (This optimization is also sometimes called "Shamir's trick" because [El84] credits Shamir for pointing it out.) For example, the product of two powers $a^b c^d$, for $a, b \neq 0$, can be computed by:

$r \leftarrow 1$
for i **from** max(msb(b), msb(d)) **down to** 0:
    $r \leftarrow r^2$
    if $b_i = 1 \wedge d_i = 1$ **then** $r \leftarrow r \cdot ac$
    else if $b_i = 1$ **then** $r \leftarrow r \cdot a$
    else if $d_i = 1$ **then** $r \leftarrow r \cdot c$
end for
return r

Note that since ac is constant, its value only needs to be computed once.

The implementation of DSA that was benchmarked uses both of the precomputation optimization and Straus's algorithm. Table 6.4 gives the performance results for DSA with different sizes of p.

6.6 ECDSA implementation

ECDSA can be implemented on either a prime field or a binary field. The benchmark program that comes with the MIRACL library indicated that a prime field implementation would be slightly faster, so a prime field was chosen for this benchmark.

Since ECDSA is an analogue of DSA, many of the optimizations for DSA also apply to ECDSA as well. In particular, the optimizations of using precomputation in the signing process and using Straus's algorithm in the signature verification process carry over directly to ECDSA. Additional optimizations specific to ECDSA also exist.

Many of the optimizations involve choosing system parameters that satisfy certain special forms. For example, with the prime field $\mathbb{F}_p$, reduction modulo p is faster when the binary form of p consists of almost all ones (a generalized Mersenne prime). Binary fields, which are somewhat more complicated, offer many more parameters that can be specially tuned. One of the disadvantages of using special choices for the parameters is that the implementations become extremely parameter-specific, which is undesirable if an application needs to interoperate with other applications that use different parameters.

Another avenue for optimization which is not parameter-specific is the representation of the elliptic curve points themselves. This thesis has so far described a point as a pair of coordinates $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$. This representation is the *affine-coordinate* representation. However, other coordinate systems are possible as well. The formulas for adding a pair of points and for doubling a point have faster-to-compute forms in other coordinate systems. Using *projective coordinates*, for example, a point (x, y) is represented as the triple (X, Y, Z), where $x = X/Z$ and $y = Y/Z$. The addition and doubling formulas do not use division in projective coordinates, making them faster.

Signing time:	6.98 ms
Verification time:	20.08 ms

Table 6.5: ECDSA performance with a 160-bit prime field, using projective coordinates, precomputation, and Straus's algorithm.

[CMO98] compares different coordinate systems, and advocates mixing coordinate systems for maximum efficiency.

Table 6.5 gives the performance of ECDSA, optimized with precomputation and Straus's algorithm. The implementation used the MIRACL library for elliptic curve arithmetic. MIRACL does not have assembly-language optimizations for the PowerPC, but it does offer point representations in projective coordinates.

The specific elliptic curve used for this benchmark was the "`secp160k1`" curve from [SEC2]. It is a curve on a 160-bit prime field.

6.7 A library for finite-field arithmetic

The remaining two signature schemes are multivariate quadratic (MQ) schemes. MQ schemes offer a lot of room for micro-optimizations, since they generally repeat small operations a large number of times. Therefore, small changes to the way a frequently-used operation is written can have large effects on the overall performance.

In order to evaluate the two MQ signature schemes, the author implemented a library for performing finite-field arithmetic, written in C. Both SFLASH and TTS use small fields—$\mathbb{F}_{128}$ and $\mathbb{F}_{256}$, respectively—so field elements were represented as single bytes, and vectors of field elements were represented as arrays of bytes.

The most frequently used operation is field multiplication, so a lot of effort was applied to making it as efficient as possible. Multiplication was implemented using log-

arithms: $xy = g^{\log_g x + \log_g y}$, for a suitable generator g. The logarithms were performed using log/antilog tables. (An alternative is to build a large table that gives the product of every possible pair of x's and y's in a single lookup, but such a table would not fit into the processor's cache very well.)

One of the goals in implementing multiplication was that the code be entirely branchless: the code should not use any conditional statements, which were measured to be considerably slower. This is challenging because multiplication by zero needs special handling, since log(0) cannot be defined in any way that gives the correct semantics for multiplication. The multiplication function was implemented as:

```
static inline unsigned mult(unsigned x, unsigned y,
      const unsigned char *logTbl, const unsigned char *antilogTbl) {
   return antilogTbl[logTbl[x] + logTbl[y]] & -((x != 0) & (y != 0));
}
```

where `logTbl` is the table of logarithms and `antilogTbl` is the table of powers. The `antilogTbl` is twice the size of the `logTbl`, to prevent needing to compute the sum of the logarithms modulo $|\mathbb{F}| - 1$, where $\mathbb{F}$ is the field.

Some quantities in an MQ signature scheme are always used as multipliers (the coefficients in the public key, for example, or the inverse matrices M_1^{-1} and M_3^{-1} in the secret key), so they are stored in logarithm form to save a lookup during multiplication; a separate version of the multiplication function is defined for them. To allow zeros to be stored, log(0) is stored as 0, and log(1) is stored as $|\mathbb{F}| - 1$.

The finite-field arithmetic library also implements functions that operate on vectors and matrices of field elements, such as matrix multiplication and Gaussian elimination on a system of equations. The dimensions of the matrices and vectors need to be passed to the functions as arguments, but the library also offers the option of

hardcoding them through `#define`s so that the compiler's constant propagator can optimize them away.

6.8 SFLASH implementation

SFLASH is notable in that it needs to operate on the L, the 37^{th}-degree extension of the field $K = \mathbb{F}_{128}$. The finite-field library implemented functions to multiply elements from L efficiently.

The most interesting problem in SFLASH is implementing the inverse of the central map F. The inverse is $F^{-1}(B) = B^{(128^{11}+1)^{-1} \pmod{128^{37}-1}}$, where $B \in L$. The exponent $(128^{11}+1)^{-1} \pmod{128^{37}-1}$ is a huge number, and using square-and-multiply exponentiation is too slow.

[ACDG03] describes some of the techniques for optimizing F^{-1}. One key insight is that X^{128} (and the higher powers X^{128^n}) is a linear transformation of the coefficients in X, so it can be implemented by a matrix multiplication. Furthermore, the matrix would only consist of ones and zeros, which allows further optimizations.

Using the linear properties of X^{128} in L, [ACDG03] gives an addition chain for computing $B^{(128^{11}+1)^{-1} \pmod{128^{37}-1}}$ using a small number of multiplications:

$$A_1 = B^4 \qquad A_5 = A_4^2 \times A_4 \qquad A_9 = (A_5 \times A_8)^{128^8}$$

$$A_2 = BA_1 \qquad A_6 = A_3^4 \qquad A_{10} = A_6^{128^7} \times A_6 \times A_8 \times A_9$$

$$A_3 = A_1^4 \qquad A_7 = A_5 \times ((A_6^{128})^{128})^{128} \qquad A_{11} = ((A_{10}^{128^7})^{128^8} \times A_9)^{128^7} \times A_{10}$$

$$A_4 = A_2 \times A_3 \qquad A_8 = ((A_7^{128} \times A_7)^{128} \times A_7)^{128}$$

Then $A_{11} = B^{(128^{11}+1)^{-1} \pmod{128^{37}-1}}$.

SFLASH specifies that the hash output is a vector in K^{26}, which is 182 bits long.

Signing time:	2.13 ms
Verification time:	0.75 ms

Table 6.6: SFLASH performance.

Since SHA-1 only produces 160 bits of output, it needs to be called twice. The first 182 bits of $\mathsf{hash}(m) \,||\, \mathsf{hash}(\mathsf{hash}(m))$ is then used as the message hash.

A minor wrinkle in implementing SFLASH is that it uses a field $K = \mathbb{F}_{128}$ that is seven bits wide, but the rest of the system produces bytes that are eight bits wide. Therefore, a conversion routine is needed to pack and unpack the bits.

Table 6.6 presents the performance of the SFLASH signature scheme, using the optimizations described here.

6.9 TTS implementation

TTS is somewhat simpler to implement than SFLASH. Inverting the central map in TTS requires solving a randomly-generated system of nine equations for nine unknowns, which was implemented using Gaussian elimination.

Since the systems of equations are randomly generated, they are not always solvable. Unfortunately, there is no way to detect whether a system of equations is solvable much faster than actually solving them. The Gaussian elimination routine that was implemented returns an error if it detects that the system of equations is unsolvable, signaling the implementation to try again. This detection happens about half way through the Gaussian elimination process, when the algorithm has reduced the matrix containing the coefficients of the equations into a triangular form.

Table 6.7 gives the timings for signing and signature verification in TTS. It also gives how long it takes to generate a random system of equations, attempt to solve

Signing time:	0.208 ms
Verification time:	0.426 ms
Unsolvable equations time:	0.028 ms

Table 6.7: TTS performance. The unsolvable equations time is the time it takes to set up a system of nine equations for nine unknowns, attempt to solve them, and realize that they are unsolvable.

them, and realize that they cannot be solved. A fault-tolerant system using TTS will need to allow time for enough retries that the probability of still failing to obtain the signature becomes acceptably low.

6.10 Conclusion

Table 6.8 summarizes the results of all of the tested signature schemes. The final column "sign + 3 verifies" is the time it takes to compute one signature and verify it three times. This represents the amount of time that the message authentication would take in a four-node system for a single-source message exchange: the sender node needs to sign its message, and each of the recipient nodes needs to verify the signature on the three copies of the message it receives.

The performance of the signature scheme implementations developed for this thesis is generally in line with the benchmarks from the other sources presented earlier. Most of the differences can probably be attributed to the different platforms that were used. Looking at the ratios between signing and verification times, the ratio for the 1,024-bit RSA implementation in this thesis appears to be much larger than the one for the implementation in [Pr03b, table 13]. However, this ratio seems very sensitive to the key size, so the discrepancy can be explained if the sensitivity is slightly stronger for the implementation in this thesis. Another notable discrepancy is that the ratio between

Signature scheme	Signature length (bytes)	Signing time (ms)	Verification time (ms)	Sign + 3 verifies time (ms)
RSA-PSS-256	32	0.87	0.17	1.38
RSA-PSS-512	64	4.37	0.44	5.69
RSA-PSS-768	96	11.38	0.80	13.78
RSA-PSS-1024	128	24.69	1.31	28.62
DSA-256	40	0.93	2.81	9.36
DSA-512	40	2.41	7.31	24.34
DSA-768	40	4.39	13.18	43.93
DSA-1024	40	6.98	23.71	78.09
ECDSA	40	6.98	20.08	67.22
SFLASH	33	2.13	0.75	4.38
TTS	28	0.208	0.426	1.486

Table 6.8: Summary of performance results for all signature schemes.

the signing time and verification time for the ECDSA implementation in this thesis is smaller than the ratio in [Pr03b, table 37]. However, it is unclear if the implementation in [Pr03b, table 37] uses precomputation for signing; if it does not, then its signing time would understandably be larger.

As expected, the MQ signature schemes gave the best performance among the signature schemes with security levels of at least 2^{80}, with TTS being the fastest, although the insecure 256-bit RSA had the fastest overall time for one signature and three verifications. And as noted earlier, the time allotted to TTS would need to be increased to reduce the probability of failing to compute a signature.

Disappointingly, none of the signature schemes tested would be able to complete a single-source message exchange in under 1 ms, and certainly not in the 200 μs that the current version of the FTPP without message authentication takes [Bu01]. Perhaps with additional optimizations, the signature and verification times could be reduced by a factor of two or so, but an order-of-magnitude improvement would be needed before

these signature schemes become usable.

These results indicate that cryptographically secure signature schemes, or even somewhat insecure ones like RSA and DSA with reduced key sizes, are too slow for a system that needs to send messages at a very high rate. If a fault-tolerant computer system must use authenticated messages, then the only currently viable solution is to resort to an insecure signature scheme.

CHAPTER 7

Conclusions

This thesis considered the practical implementation of a message authentication scheme for a Byzantine-resilient fault-tolerant computer system. It shows that a system using message authentication can use fewer nodes than a system not using message authentication while still achieving the same level of reliability.

Specifically, this thesis attempted to develop a message authentication scheme for a new version of the X-38 Fault Tolerant Parallel Processor (FTPP), a fault-tolerant computer system from Draper Laboratory that is designed for demanding applications like human spaceflight. The FTPP achieves its reliability by running its programs on multiple replicated nodes that maintain synchronized state. The current version of the FTPP, which does not use authenticated messages, is a five-node system that can handle the non-simultaneous failures of any two nodes. A new version of the FTPP could be equally fault tolerant with only four nodes if it uses message authentication.

The FTPP uses a two-round voting protocol to come to a consensus on messages that originate from a single node. In the protocol, the node with a message to send broadcasts it to all of its peers, then its peers reflect the message they receive to each other. At the end of the protocol, each node ends up with multiple copies of the message, and it chooses the most common copy to be the correct copy of the message.

The voting protocol is necessary because a faulty sender node may send different messages to different recipient nodes, but all of the recipient nodes need to agree on the message they received, since all of the nodes are supposed to be operating in the same state. Examining the voting process, this thesis concluded that messages only need to be signed during the first round, and only by the original sender.

A Byzantine-resilient system should tolerate faults in the nodes of the system, even if they fail in unexpected and unlikely ways. For this reason, this thesis strongly recommended cryptographically secure signature schemes for message authentication. With an insecure scheme, certain sequences of faults can lead to a node forging a message from another node. Only a cryptographically secure scheme can guarantee that a faulty node cannot forge signatures, even if it were actively trying to do so.

This thesis then presented several candidates for the signature scheme to be used to implement message authentication. A suitable signature scheme should be cryptographically secure, but it also must be fast, two requirements that conflict with each other. The FTPP needs to send messages at a very high rate, and that rate will only increase over time as applications produce and consume more and more data. Fortunately, computer processors have also been becoming faster. Although an all-software implementation of message authentication would have been out of the question for the original version of the FTPP, it was hoped that more modern processors have become powerful enough to allow the implementation of a cryptographically secure signature scheme in a new version of the FTPP.

This thesis implemented and evaluated the signature schemes RSA, DSA, elliptic curve DSA, SFLASH, and TTS. These signature schemes have widely varying signing and verification times, signature lengths, and memory requirements. Unfortu-

nately, all of them turned out to be too slow. Even the fastest of them would need to run an order of magnitude faster to satisfy the message rates of the FTPP.

The signature schemes implemented in this thesis are still useful, since they would be suitable for a low-message-rate fault-tolerant system that needs authenticated messages. For the FTPP, however, one must conclude that cryptographically secure signature schemes are not yet feasible on the current generation of embedded processors.

7.1 Cryptographic security versus speed: a compromise

A message authentication implementation for the FTPP must settle for a cryptographically insecure signature scheme. However, an appropriate scheme must still be secure enough that producing forged messages does not become trivial. This thesis cannot recommend any particular candidate for the signature scheme at this point, but a suitable one should have these properties:

1. Different messages should produce different signatures. With very high probability, changing even one bit of the message should result in a signature that looks very different. No simple modification to a message-signature pair should produce a new message-signature pair that is valid. This requirement suggests that the signature scheme use a hash function like SHA-1 or MD5. From the benchmark testing done for this thesis, it appears that a cryptographically secure hash function will run fast enough, even on moderately slow processors like the ones that will be used in the new version FTPP. If a secure hash function is too slow, using a CRC may be acceptable. Something like a

simple checksum, however, is not suitable as a hash function, since it is too easy to corrupt the message in a way that does not change its checksum.

2. The output of the hash function, and the signature itself, should be at least 64 bits long, and preferably longer. A signature of this length ensures that even if a faulty node were to transmit randomly generated messages for the duration of the FTPP's operating lifetime, it would still be unable to forge a valid signature with any realistic probability.
3. The signing and verification process should be different, and they should depend on different data. This suggests a public-key signature scheme, where the secret key is known only to one node. The danger with a scheme in which the key for signing and the key for signature verification is the same is that something as simple as a single-bit error in an array index could cause a node to sign with another node's key, thereby forging a signature.
4. The procedure to generate the secret key from the public key, or to forge a signature, should not be simple. RSA with a reduced-size modulus, for example, meets this requirement: even if the modulus can be easily factored, it is extraordinarily unlikely that a series of faults would precisely implement the steps of a factoring algorithm.

A cryptographically insecure but sufficiently complex signing function combined with a secure hash function may yield a signature scheme that meets all of these requirements. Candidates for the insecure signing function include RSA with very short moduli and functions based on matrix multiplication (where signing is multiplication by a secret matrix, and verification is multiplication by its inverse). However, more analysis is needed before any signature scheme can be recommended.

7.2 Possibilities for future research

One avenue for future research, of course, is to continue the work of this thesis and find a faster signature scheme that can be used for message authentication in the FTPP.

However, the signature schemes described in this thesis, even though they are too slow for the FTPP, can still be put to use elsewhere. TTS, for example, would make a good candidate for a system with lower message rate requirements. Another possibility is to use special hardware to accelerate these signature schemes. The goal of the FTPP is to build a fault-tolerant system without using custom hardware, but other systems may not have the same restriction.

This thesis is part of a larger effort at Draper Laboratory to build a better fault-tolerant computer system, and there are certainly many opportunities for further work as part of that effort. The focus this year has been on communication within the system, but future work will address topics like fault detection and recovery.

Looking more broadly, this thesis covers several general themes. Each of them is rich with opportunities for further research:

Cryptography for specific applications Much of cryptography is focused on information security. This thesis is somewhat unusual in that uses cryptography for *reliability* instead. Many distributed systems can be simplified if the member nodes can be prevented from cheating, which is where cryptographic techniques become useful.

Looking at the cryptographic algorithms themselves, it is clear that many of them have been developed to meet specific requirements. For example, encryption and signature schemes based on elliptic curves offer the same benefits as the more classical schemes like RSA, but the elliptic curve schemes can use smaller keys. The

multivariate quadratic schemes, in turn, can run much faster. It is reasonable to expect that further research will someday produce secure signature schemes fast enough to use in demanding systems like the FTPP.

Implementation and optimization techniques During the course of this thesis, a lot of effort was devoted to making the signature schemes run as quickly as possible. However, the benchmarks in this thesis are obviously not the final word on how fast signature schemes can be made to run. More research on optimization techniques can be expected to produce both small improvements and algorithmic breakthroughs that give dramatic speedups.

Building more robust systems Currently, fault-tolerant computer systems still occupy only a small niche in computer engineering. One might encounter a fault-tolerant computer system when one gets on an airplane, or when one uses an electronic medical device, but in general, most of the computer systems one encounters are not even remotely as reliable as a system like the FTPP. Not all systems need to be as reliable, of course, and not all systems can afford to be. However, more research into building systems that do need to be extremely reliable will yield techniques that can be applied to ordinary systems as well, ultimately making all computer systems more reliable.

Bibliography

[ACDG03] M.-L. Akkar, N. Courtois, R. Duteuill, L. Goubin, "A Fast and Secure Implementation of Sflash," *Proceedings of the 6th International Workshop on Practice and Theory in Public Key Cryptography*, Lecture Notes in Computer Science, vol. 2567, Springer-Verlag, 2003, 267–278.

[ANSI X9.62] ANSI X9.62: "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)," American National Standards Institute, 1998, revised 2005.

[BBB05] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid, *Recommendation for Key Management—Part 1: General*, NIST Special Publication 800-57, National Institute of Standards and Technology, 2005.

[BGM97] M. Bellare, S. Goldwasser, D. Micciancio, " 'Pseudo-Random' Number Generation within Cryptographic Algorithms: the DSS Case," *Advances in Cryptology—Crypto '97 Proceedings*, Lecture Notes in Computer Science, vol. 1294, Springer-Verlag, 1997, 277–291.

[BR96] M. Bellare, P. Rogaway, "The Exact Security of Digital Signatures—How to Sign with RSA and Rabin," *Advances in Cryptology—Eurocrypt '96 Proceedings*, Lecture Notes in Computer Science, vol. 1070, Springer-Verlag, 1996, 399–416.

[Bo99] D. Boneh, "Twenty Years of Attacks on the RSA Cryptosystem," *Notices of the American Mathematical Society*, vol. 46, no. 2, 1999, 203–213.

[BGMW92] E. Brickell, D. Gordon, K. McCurley, D. Wilson, "Fast Exponentiation with Precomputation," extended abstract, *Advances in Cryptology—Eurocrypt '92 Proceedings*, Lecture Notes in Computer Science, vol. 658, 1993, 200–207.

[BJ02] E. Brier, M. Joye, "Weierstraß Elliptic Curves and Side-Channel Attacks," *Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems*, Lecture Notes in Computer Science, vol. 2274, Springer-Verlag, 2002, 335-345.

[Bu01] D. Buscher, book manager, *X-38 Fault Tolerant Parallel Processor Requirements*, revision 6.2, NASA, 2001.

[CL99a] M. Castro, B. Liskov, "Authenticated Byzantine Fault Tolerance Without Public-Key Cryptosystems," Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[CL99b] M. Castro, B. Liskov, "Practical Byzantine Fault Tolerance," *Proceedings of the Third Symposium on Operating-Systems Design and Implementation*, 1999, 173–186.

[CDSL02] Charles Stark Draper Laboratory, *Software Design Description for the X-38 Fault Tolerant System Services*, vols. 1–8, revision A, 26 September 2002.

[CY03] J.-M. Chen, B.-Y. Yang, "A More Secure and Efficacious TTS Signature Scheme," full version, presented at the International Conference on Information Security and Cryptology 2003 (shorter version published in proceedings), Cryptology ePrint Archive, report 2003/160, `http://eprint.iacr.org/2003/160`, 2003, revised 3 January 2004.

[CYP02] J.-M. Chen, B.-Y. Yang, B.-Y. Peng, "Tame Transformation Signatures with Topsy-Turvy Hashes," *Proceedings of the Second International Workshop for Asian Public Key Infrastructures*, 2002, 93–100.

[Cl94] A. L. Clark, "An architecture study of a Byzantine-resilient processor using authentication," Master of Science thesis, Massachusetts Institute of Technology, 1994.

[CMO98] H. Cohen, A. Miyaji, T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," *Advances in Cryptology—Asiacrypt '98 Proceedings*, Lecture Notes in Computer Science, vol. 1514, Springer-Verlag, 1998, 51–65.

[CGP02] N. Courtois, L. Goubin, J. Patarin, "SFLASH, a fast asymmetric signature scheme for low-cost smartcards," available from `http://www.minrank.org/sflash/`, 2002.

[CGP03] N. Courtois, L. Goubin, J. Patarin, "SFLASH^{v3}, a fast asymmetric signature scheme," Cryptology ePrint Archive, report 2003/211, `http://eprint.iacr.org/2003/211`, 2003.

[Da04] W. Dai, "Speed Comparison of Popular Crypto Algorithms: Crypto++ 5.2.1 Benchmarks," `http://www.eskimo.com/~weidai/benchmarks.html`, modified 23 July 2004.

[DS04] J. Ding, D. Schmidt, "Multivariable Public-Key Cryptosystems," Cryptology ePrint Archive, report 2004/350, `http://eprint.iacr.org/2004/350`, 2004.

[DSY06] J. Ding, D. Schmidt, Z. Yin, "Cryptanalysis of the new TTS scheme in CHES 2004", *International Journal of Information Security*, online issue, 4 April 2006.

[Do82] D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms*, vol. 3, no. 1, March 1982, 14–30.

[DS83] D. Dolev, H. R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal on Computing*, vol. 12, no. 4, November 1983, 656–666.

[ESC05] D. Eastlake III, J. Schiller, S. Crocker, "Randomness Requirements for Security," RFC 4086, 2005.

[Ed02] A. Edsall, "The X-38 Fault Tolerant Parallel Processor: High Reliability Flight Control Based on COTS Computer Hardware," presentation at the AIAA/IAP Symposium on Future Reusable Launch Vehicles, 11–12 April 2002.

[El84] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Advances in Cryptology—Crypto '84 Proceedings*, Lecture Notes in Computer Science, vol. 169, Springer-Verlag, 1985, 10–18.

[EP405] Embedded Planet 405 PowerPC Single Board Computer, `http://www.embeddedplanet.com/products/ep405.asp`.

[FIPS 180-1] FIPS PUB 180-1: "Secure Hash Standard," Federal Information Processing Standards Publication, National Institute of Standards and Technology, 1995.

[FIPS 186-2] FIPS PUB 186-2: "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication, National Institute of Standards and Technology, 2000.

[Fi83] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)," *Proceedings of the 1983 International Foundations of Computation Theory Conference*, Lecture Notes in Computer Science, vol. 153, 1983, 127–140.

[FLP85] M. Fischer, N. Lynch, M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, April 1985, 374–382.

[Ga90] R. Galetti, "Real-Time Digital Signatures and Authentication Protocols," Master of Science thesis, Massachusetts Institute of Technology, 1990.

[GSB01] W. Geiselmann, R. Steinwandt, T. Beth, "Attacking the Affine Parts of SFLASH," *Proceedings of the 8th IMA International Conference on Cryptography and Coding*, Lecture Notes in Computer Science, vol. 2260, Springer-Verlag, 2001, 355–359.

[GMP] GNU Multiple Precision (GMP) Arithmetic Library, `http://www.swox.com/gmp/`.

[GHSC] Green Hills Software Optimizing C Compilers, `http://www.ghs.com/products/c_optimizing_compilers.html`.

[HL91] R. Harper, J. Lala, "Fault-Tolerant Parallel Processor," *Journal of Guidance, Control, and Dynamics*, vol. 14, no. 3, May–June 1991, 554–563.

[HLS87] A. Hopkins, Jr., J. Lala, T. B. Smith III, "The Evolution of Fault Tolerant Computing at the Charles Stark Draper Laboratory, 1955–85" *The Evolution of Fault-Tolerant Computing*, Dependable Computing and Fault-Tolerant Systems, vol. 1, Springer-Verlag, 1987, 121–140.

[HS01] N. A. Howgrave-Graham, N. P. Smart, "Lattice Attacks on Digital Signature Schemes," *Designs, Codes and Cryptography*, vol. 23, no. 3, 2001, 283–290.

[IM85] H. Imai, T. Matsumoto, "Algebraic Methods for Constructing Asymmetric Cryptosystems," *Proceedings of the 3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, Lecture Notes in Computer Science, vol. 229, Springer-Verlag, 1985, 108–119.

[INTEG] INTEGRITY Real-Time Operating System, `http://www.ghs.com/products/rtos/integrity.html`.

[JM98] D. Johnson, A. Menezes, "Elliptic Curve DSA (ECDSA): An Enhanced DSA," *Proceedings of the 7th USENIX Security Symposium*, 1998.

[JMV01] D. Johnson, A. Menezes, S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, 2001, 36–63.

[Jo01] J. Jonsson, "Security Proofs for the RSA-PSS Signature Scheme and Its Variants," draft 1.1, presented at the Second Open NESSIE Workshop, Cryptology ePrint Archive, report 2001/053, `http://eprint.iacr.org/2001/053`, 2001, revised 3 July 2001.

[KWFT88] R. Kieckhafer, C. Walter, A. Finn, P. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, April 1988, 398–405.

[KC04] P. Koopman, T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks," *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004, 145–154.

[LSP82] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, 382–401.

[Le96] A. K. Lenstra, "Memo on RSA signature generation in the presence of faults," available at `http://cm.bell-labs.com/who/akl/rsa.doc`, 1996.

[LV01] A. K. Lenstra, E. Verheul, "Selecting Cryptographic Key Sizes," *Journal of Cryptology*, vol. 14, no. 4, 1999, revised 2001, 255–293.

[Ma05] P. Mackerras `<paulus@samba.org>`, "[PATCH] optimized SHA1 for powerpc," message sent to the Git mailing list, 22 April 2005.

[MIRACL] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL), `http://indigo.ie/~mscott/`.

[NS02] P. Nguyen, I. Shparlinski, "The Insecurity of the Digital Signature Algorithm with Partially Known Nonces," *Journal of Cryptology*, vol. 15, no. 3, 2002, 151–176.

[PB86] D. Palumbo, R. Butler, "A Performance Evaluation of the Software-Implemented Fault-Tolerance Computer," *Journal of Guidance, Control, and Dynamics*, vol. 9, no. 2, March–April 1986, 175–180.

[PSL80] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, no. 2, April 1980, 228–234.

[PB61] W. W. Peterson, D. T. Brown, "Cyclic Codes for Error Detection," *Proceedings of the Institute of Radio Engineers*, vol. 49, no. 1, January 1961, 228–235.

[PKCS1v1.5] "PKCS #1: RSA Cryptography Standard," version 1.5, *Public-Key Cryptography Standards*, RSA Laboratories, `http://www.rsasecurity.com/rsalabs/node.asp?id=2125`, 1993.

[PKCS1v2.1] "PKCS #1: RSA Cryptography Standard," version 2.1, *Public-Key Cryptography Standards*, RSA Laboratories, `http://www.rsasecurity.com/rsalabs/node.asp?id=2125`, 2002.

[Pr03a] B. Preneel *et al.*, *NESSIE Security Report*, version 2.0, NESSIE Deliverable D20, Information Societies Technology (IST) Programme of the European Commission, 2003.

[Pr03b] B. Preneel *et al.*, *Performance of Optimized Implementations of the NESSIE Primitives*, version 2.0, NESSIE Deliverable D21, Information Societies Technology (IST) Programme of the European Commission, 2003.

[Pr98] M. J. Prizant, "High-Speed Communicator for Fault Tolerant Systems," *Proceedings of the 17th Digital Avionics Systems Conference*, vol. 1, 1998, D25-1 – D25-7.

[RLB02] R. Racine, M. LeBlanc, S. Beilin, "Design of a Fault-Tolerant Parallel Processor," *Proceedings of the 21st Digital Avionics Systems Conference*, vol. 2, 2002, 13D2-1 – 13D2-10.

[RECFGU] "Recommended Elliptic Curves for Federal Government Use," National Institute of Standards and Technology, 1999.

[Ri92] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, 1992.

[RSA78] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, February 1978, 120–126.

[SEC1] "SEC 1: Elliptic Curve Cryptography," version 1.0, Standards for Efficient Cryptography Group, `http://www.secg.org/download/aid-385/sec1_final.pdf`, 2000.

[SEC2] "SEC 2: Recommended Elliptic Curve Domain Parameters," version 1.0, Standards for Efficient Cryptography Group, `http://www.secg.org/download/aid-386/sec2_final.pdf`, 2000.

[Si00] R. Silverman, "A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths," *RSA Laboratories Bulletin*, no. 13, `http://www.rsasecurity.com/rsalabs/node.asp?id=2088`, 2000, revised 2001.

[St06] R. M. Sterling, "Synchronous Communication System for a Software-Based Byzantine Resilient Fault Tolerant Computer," Master of Engineering thesis, Massachusetts Institute of Technology, 2006.

[St95] D. Stine, "Digital Signatures for a Byzantine Resilient Computer System," Bachelor and Master of Science thesis, Massachusetts Institute of Technology, 1995.

[St64] E. Straus, "Addition Chain of Vectors," solution to problem 5125, *American Mathematical Monthly*, vol. 71, no. 7, August–September 1964, 806-808.

[WFLY04] X. Wang, D. Feng, X. Lai, H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD," presented at the Rump Session of Crypto 2004, Cryptology ePrint Archive, report 2004/199, `http://eprint.iacr.org/2004/199`, 2004.

[WLG78] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. M. Melliar-Smith, R. Shostak, C. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, vol. 66, no. 10, October 1978, 1240–1255.

[Wo05] C. Wolf, "Multivariate Quadratical Polynomials in Public Key Cryptography," Doctor of Engineering *(doctoraat in de ingenieurswetenschappen)* thesis, Katholieke Universiteit Leuven, Belgium, 2005.

[WP04] C. Wolf, B. Preneel, "Applications of Multivariate Quadratic Public Key Systems" extended version, third revision, Cryptology ePrint Archive, report 2004/263, `http://eprint.iacr.org/2004/263`, 2004, revised 6 August 2005.

[YC05] B.-Y. Yang, J.-M. Chen, "Building Secure Tame-like Multivariate Public-Key Cryptosystems: The New TTS," *Proceedings of the 10th Australasian Conference on Information Security and Privacy*, Lecture Notes in Computer Science, vol. 3574, 2005, 518–531.

[YC04] B.-Y. Yang, J.-M. Chen, "TTS: Rank Attacks in Tame-Like Multivariate PKCs," third revision, Cryptology ePrint Archive, report 2004/061, `http://eprint.iacr.org/2004/061`, 2004, revised 29 September 2004.

[YCC04] B.-Y. Yang, J.-M. Chen, Y.-H. Chen, "TTS: High-Speed Signatures on a Low-Cost Smart Card," *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, vol. 3156, Springer-Verlag, 2004, 371–385.

Colophon

The text of this thesis is set in URW Garamond, supplemented with mathematical symbols from the Math Design typeface. The thesis was typeset in LaTeX, using a heavily-modified version of the MIT thesis template originally by S. Gildea, P. Nuth, and K. Sethuraman. Mathematical equations were typeset with the aid of the packages from $\mathcal{AMS}$-LaTeX. Diagrams were drawn in OpenOffice. The final output, with margin kerning, was rendered with pdfTeX.

MIT no longer requires theses be double-spaced. However, the weight of decades, if not centuries, of tradition was too overwhelming to ignore, so this thesis was set in double space.