8-1-2009

Hardware-Assisted Secure Computation

Alexander Iliev
Dartmouth College

Hardware-Assisted Secure Computation

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Alexander Iliev

DARTMOUTH COLLEGE

Hanover, New Hampshire

August 2009

Examining Committee:

(chair) Sean W. Smith, Ph.D.

M. Douglas McIlroy, Ph.D.

Peter Winkler, Ph.D.

Apu Kapadia, Ph.D.

Brian W. Pogue, Ph.D.

Dean of Graduate Studies

Computer Science Technical Report TR2009-659

Abstract

The theory community has worked on Secure Multiparty Computation (SMC) for more than two decades, and has produced many protocols for many settings. One common thread in these works is that the protocols cannot use a Trusted Third Party (TTP), even though this is conceptually the simplest and most general solution. Thus, current protocols involve only the direct players—we call such protocols self-reliant. They often use blinded boolean circuits, which has several sources of overhead, some due to the circuit representation and some due to the blinding.

However, secure coprocessors like the IBM 4758 have actual security properties similar to ideal TTPs. They also have little RAM and a slow CPU. We call such devices Tiny TTPs. The availability of real tiny TTPs opens the door for a different approach to SMC problems. One major challenge with this approach is how to execute large programs on large inputs using the small protected memory of a tiny TTP, while preserving the trust properties that an ideal TTP provides. In this thesis we have investigated the use of real TTPs to help with the solution of SMC problems.

We start with the use of such TTPs to solve the Private Information Retrieval (PIR) problem, which is one important instance of SMC. Our implementation utilizes a 4758.

The rest of the thesis is targeted at general SMC. Our SMC system, Faerieplay, moves some functionality into a tiny TTP, and thus avoids the blinded circuit overhead.

Faerieplay consists of a compiler from high-level code to an arithmetic circuit with special gates for efficient indirect array access, and a virtual machine to execute this circuit on a tiny TTP while maintaining the typical SMC trust properties. We report on Faerieplay's security properties, the specification of its components, and our implementation and experiments. These include comparisons with the Fairplay circuit-based two-party system, and an implementation of the Dijkstra graph shortest path algorithm. We also provide an implementation of an oblivious RAM which supports similar tiny TTP-based SMC functionality but using a standard RAM program. Performance comparisons show Faerieplay's circuit approach to be considerably faster, at the expense of a more constrained programming environment when targeting a circuit.

Acceptance Speech

Thank you, thank you everyone in the academy for bestowing on me the honor of this degree. In accepting, I have to thank the many people who have been instrumental for me reaching this far.

For most of my research career, Sean Smith was my advisor and mentor. At the beginning, Sean provided research ideas and problems, while I was still unfamiliar with this most important part of the research enterprise. He was unshakably confident in the quality and usefulness of our work, which was important for me when I had doubts. He has great faith in the ability and motivation of his students, and stays hands off to an incredible extent. He was very patient during my long thesis writing period, when any attempt to force progress would probably have aborted the mission.

Before Sean, David Kotz gave me a start in the research direction in his mobile agents group, where I worked for several undergraduate years. Sean and I got inspired for our hw-PIR work by Dmitri Asonov's PET 2002 paper on practical PIR.

John Marchesini was my closest friend during our time at Dartmouth, and also happened to be my office mate. We understood each other in a way I know is quite rare and to be treasured. Some of my other lab mates and friends with whom I spent much time on work and fun—Patrick Tsang, Scout Sinclair, Apu Kapadia, Sergey Bratus, Anna Shubina, Nihal D'Cunha. Apu brought with him a focus on publication which was useful to see in someone closer to my level, and he made every effort to involve others in the lab with his research ideas. With me in London, Scout offered to be my agent in Hanover and got my thesis printed, signed and delivered.

My committee made suggestions and worked with me to revise the text, which made the presentation clearer, more complete and more precise.

My wife Iga took the brunt of my absence as I wrote, with one and then two energetic boys to handle on her own during this extensive time. My father was instrumental in the start of my studies in the US, as he emphasized that this is the correct path, and helped financially to make it happen. My mother likewise kept helping as my independence occasionally faltered.

Preface

Several of the chapters in this thesis are derived from our previously published papers. Chapter 2 was published in the *2003 PKI workshop* [46], Chapter 3 was published in the *3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems* [47]. Parts of Chapter 2 and Chapter 3 also appeared in *IEEE Security and Privacy* [48].

This research was supported in part by NSF grant CNS-0524695, the Bureau of Justice Assistance grant 2005-DD-BX-1091, and NSF infrastructure grant EIA-98-02068. The views and conclusions do not necessarily reflect those of the sponsors.

I can be contacted at alexander.iliev.01@alum.dartmouth.org

The code associated with this thesis is available under
`http://www.cs.dartmouth.edu/~trust/faerieplay/`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the developed world, much of a person's life and a company's business is now computerized. This brings with it benefits like increased efficiency and scalability, while extensive computer networks allow for greater collaboration possibilities than existed before. However, running so much personal and company business through computers and networks brings about the potential for abuse in ways which were not feasible before. In particular, a lot more data about individuals and organizations flows outside their control, in ways which are often very subtle and unnoticed. This enables various forms of data collection and analysis, ostentatiously to useful ends, but with more problematic implications.

Individuals have dossiers collected, which can result in problems ranging from unwanted marketing, to denial of services or employment, based on an out-of-context analysis of a dossier.

Organizations face the problem that, if they want to take advantage of online collaboration opportunities, they apparently have to expose proprietary data which they wish to protect from outsiders. Even if they have contracts with their peer organizations which govern how data can be used, the peer companies' information security practises could prove inadequate, and malicious insiders at the peers may abuse data in spite of the contracts.

1.1 Individuals: dossier collection

A sinister aspect of mass computerization is that it enables a vastly more efficient dossier creation, than was possible with a paper-based bureaucracy. This means that an individual participating in this new economy may be wise to think twice about what information he is giving out to whom, in the context of using some online service, or participating in some collaboration mediated by computers.

As a simple example, a service provider may be (and most probably is) collecting data about his users. This may seem innocuous at first glance, but especially if aggregated across several sources, such data can enable very specific conclusions to be made about its subjects.

As Kafka shows us in his novels, conclusions about people based on data collected without proper oversight can be incorrect, and can lead to unexpected if not absurd outcomes. Such conclusions will probably not take relevant context into account. The availability of automated dossier systems encourages automated and thus simplistic analysis of the data they collect, which may make it difficult for subjects to make their case if they are for instance denied health insurance or employment based on credit and similar dossier data. If care is not taken to ensure transparency in our gigantic dossier-generating systems, a real version of *The Trial* may not be too far off. [50, 80].

A user Bob participating in transactions with non-corporations, say with peers in an online game, may have the impression that this activity cannot lead to dossier creation. However, a closer examination of the situation may reveal that the activity is not as free of institutional access as Bob had hoped:

- His software, which he almost certainly did not write himself, may be collecting some information, in order to (for example) generate embedded advertising in the game. One could object that such activity by the game software should be flagged as spyware by monitoring software, so at least making Bob aware that he is providing information to others. But what if some "innocuous" form of information gathering (like information used for targeted advertising) becomes so ingrained that users do not think of it as spyware?

- Bob's program may be clean, and not engage in information gathering, but what if his peers are not as careful? In the absence of explicit countermeasures, the information Bob's peers

can see may be just as good for dossier collection as that available at Bob's computer.

Many people would argue that the Internet provides de-facto anonymity, certainly good enough for an average user with "nothing to hide". (Here we note that K in Kafka's *The Trial* had nothing to hide too.) One problem with this argument is that faced with persistent information storage, all it takes is one slip of anonymity to link an anonymous dossier with a specific person. If just one service receives a real name (or a ZIP code, birth date and gender [42, 85]), as part of a login or other requested information, it could enable a "dossier consortium" to put a name on a dossier which had until then not been associated with a specific person.

The point of all this is to claim that a contemporary user of the Internet should be concerned about data collection, and interested in ways to reduce the amount of information he leaks. Further information on how data on individuals is collected, and what some of the ensuing threats are, can be found in *Database Nation* [39] and *No Place to Hide* [68].

1.2 Organizations

Organizations and companies have a more immediate problem with data protection, in that data exposed to competitors or other adversaries can often be quickly used to their detriment. For example, if a competitor of Bosoft Corp. learns some information about Bosoft's short or long-term business plans, they can quickly leverage that information for competitive advantage. This is a somewhat different situation than the more long-term problem of dossier collection and misuse.

Traditional security, certainly the kind used in practise, deals only with protecting the confidentiality and integrity of data which is stored, or being communicated (in transit)[1]. Thus, data can be stored at adversarial locations, and communicated over adversarial channels. The actual use of the data is for the most not secured. It is assumed to take place in a non-adversarial situation, for example on a company's own computers. However, what if organizations want to perform a computation for which a non-adversarial setting, agreeable to all participants, cannot be found? Several examples of when this may arise:

[1] In this work we do not consider the third security property, *availability*, beyond a brief examination of how our system may be extended to provide availability assurances.

- The increasing worry over insider attacks may result in the scenario that a company's own computers are no longer considered a non-adversarial environment.
- Competing companies may be able to profit by engaging in a collaborative computation: "coopetition". In this case, none of their computers provide an environment which is considered non-adversarial by everyone involved.

We will be focusing on the second issue: useful computations with Bosoft's confidential data may need to take place on other computers. However it is worth keeping in mind that the security techniques developed also apply to the threat of adversarial insiders.

There are two problems with the industry status quo on securing data during a computation. Firstly, many potential collaborations are probably ruled out because of worry about data security outside the company's perimeter. We list some scenarios like this in the next section.

Secondly, when organizations do engage in online collaboration, security for each participant is usually provided by means of "gentlemen's agreements", or contracts which specify acceptable use of shared data, and prescribe penalties for misuse. Such arrangements are not robust in the face of a changing security environment, for example with rising insider attacks, which means that collaboration may tend to be reduced, which would be ironic given improvements in useful functionality.

1.3 Example scenarios

1.3.1 Log analysis

Computer system administrators could benefit by sharing log files in order to gain a global view of an attack in progress. However, log files can clearly reveal a lot of information about an individual system. The administrator could very well want to control how much information is revealed to anyone external, and in the absence of ways to achieve this, he would not want to share his logs.

Analysis of log information from multiple organizations may enable better detection and better forensics than if each participant had only their own logs to work with. However if the different

parties do not trust each other, they will not want to reveal their logs externally. How can they do joint log analysis without the danger that external parties will obtain their log data?

1.3.2 Operations

Maintaining the operation of distributed systems often introduces tension between the better decisions enabled with information sharing among the parties involved, and the parties' concerns with revealing their operational data externally.

Power grid

The electric power grid is a huge distributed system, not only of electricity, but also of information. Keeping it running efficiently on a large scale, for example for most of the US Midwestern states, requires the cooperation of the many entities involved: generators, transmission providers, independent system operators and consumers. Efficiency and reliability gains can be obtained through fine-grained data sharing. A recent example of such developments is the deployment of *phasor measuring units* (PMUs), to improve system stability[2].

If the involved parties cannot agree on data security controls for such collaborative projects, the projects may not be deployed, and the benefits would be forfeited. If the projects do progress, but with inadequate security provisions, the system may become more open to attack and losses which outweigh the operational efficiency gains. For example, a main tension in the power grid setting is between parties sharing data for operational purposes, and hiding data from competitors to ensure an effective competitive market.

Simulating a distributed system

We may want to see ahead of time how a distributed system may behave in different circumstances. But the participants may not want to share data with external entities. If some of the simulation can be done securely, such that proprietary data is exposed only as the required outputs of the simulation,

[2]PMUs are discussed in many documents of the North American Electric Reliability Corporation http://www.nerc.com/

participants may be willing to provide the inputs needed to make the simulation realistic.

Such a capability may have been useful in a financial services system, which failed due to an unusual event—a (scheduled) early shutdown of some of the participants, which led to failures in other system participants, and necessitated a lot of recovery work. If the event could have been modelled in advance, perhaps the failure could have been foreseen and prevented.

1.3.3 Markets and auctions

Various kinds of auctions are ubiquitous, but they require that the participant trust the auctioneer (and his computing environment) to preserve the security properties of bid secrecy and accuracy. How can an auction system be set up so that even the auctioneer, who computes and announces the results, cannot learn and divulge bid information?

Auctions are a favorite subject of works on secure computation. Market participants may wish to keep their bids private, as the bids may reflect some internal conditions they would rather not broadcast, or the bids may be computed in a way which they believe confers competitive advantage. Thus, the participants want the market/auction to be carried out with confidentiality for their bids. They may prefer if even the auctioneer organization did not have access to the bids, but the result just materialized.

What makes this area even more interesting is that in some settings deciding the auction can be a very complex computation, not just a matter of finding the maximum of a list of bids. Electrical power auctions are an example of this, and we go into that topic in Chapter 7.

1.4 Reasoning about data leakage during computing

1.4.1 Formalizing who learns what

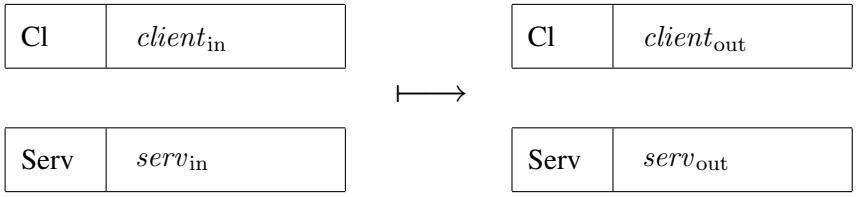
In settings where we compute things in collaboration with other parties—all online computing, and a lot of offline computing too[3]—we have gotten used to not thinking about what extra data is revealed to others during the process of computing some desired result. In this section we will begin

[3]For non-programmers, all software use involves collaboration with the software developers.

to develop some formalism for specifying fully what information flows where.

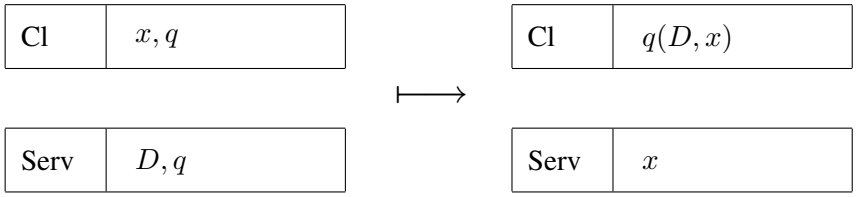
For example, in the very common situation of a client wanting to retrieve some data from a server, there is usually no question that the server will learn the client's request, and also the returned data. This does not need to be the case—the question of who gets what data during the computation can be specified just as the desired client output can be specified.

Consider such a client-server interaction, where the client wants to obtain some data from the server. Let's formalize the requirements of the interaction as a two-party function, with separate outputs for each party. We'll place the client first, then the server, in this format:



This notation means that the client provided $client_{\text{in}}$ and the server provided $serv_{\text{in}}$ to the computation, and they respectively received $client_{\text{out}}$ and $serv_{\text{out}}$ as results of the computation.

Let's look at a more concrete example. Say that the server has a dataset D, which has an associated query function q. The client knows what this function is[4]. The dataset could contain textual information for objects with names, in which case q would be a name look-up. It could be a graph, and q would then be some kind of path search (eg. for cycling directions). The client has some query data x, appropriate to D and q, and wants to learn the value of q applied to D and his query. Then, a first approximation of our two party function, in all current settings, looks like:



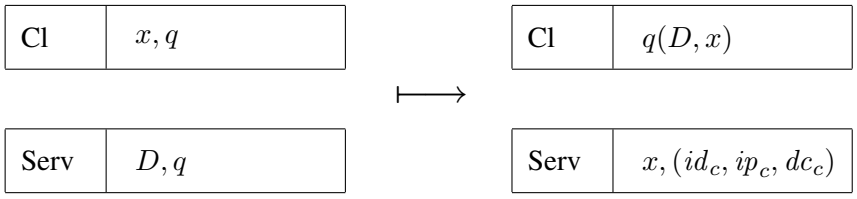
Ie. the client learns the result of his query, and the server learns the query (which for the server implies the result of the query, so we do not include $q(D, x)$ in the server output.)

[4]For now we will avoid the issue of what "knowing" a function means, and just say the client knows how to specify the function to the server.

In fact, the situation is usually worse for the privacy of the client. The dominant network application, HTML and friends over HTTP, now comes with extensive systems to collect and analyze user data. Companies like DoubleClick and Phorm are working with many online services to collect and correlate data on the services' users. In this environment, our stereotypical server probably learns much more from his client than just the query. For example:

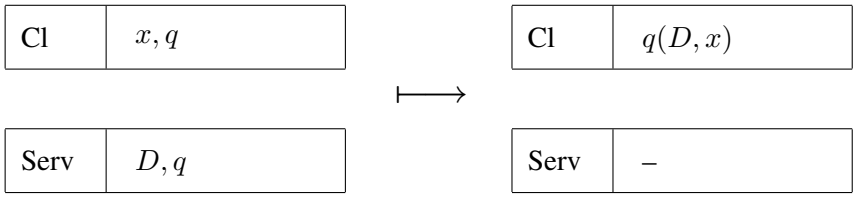
- If the server required the client to create an account and log in with that account, then the server learns the client's unique identifier id_c.
- The network infrastructure provides the client's network address ip_c.
- If the server participates in the extensive web tracking system run by DoubleClick and similar organizations, the server learns the client's DoubleClick identifier dc_c.

Thus, our two-party function probably looks more like this:



Additionally, the server can use these data to learn even more information about the client, for example by linking to the activity of the same IP address in other services.

This function is probably not what the client really has in mind for his interaction with the server. In terms of functionality, all the client wants is:

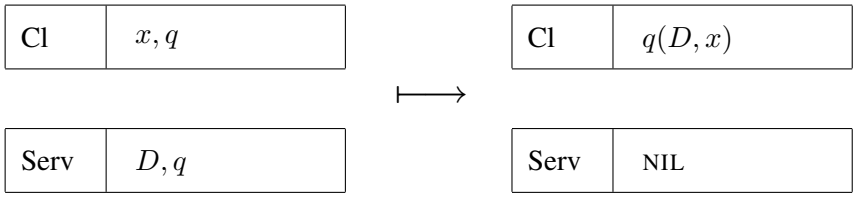


where _ is a don't-care value, ie. the spec does not say what the server's output is.

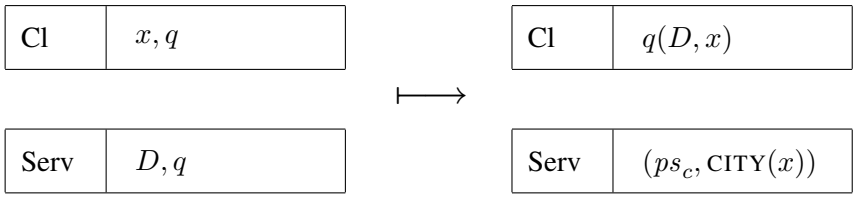
1.4.2 Specifying arbitrary limits

Here we begin to see some options opening up. The server's output is left undefined by the client's functional requirements: disregarding security concerns, the client wants to learn $q(D, x)$, and is not thinking about anything else. Currently in the wild, this opening has been filled by servers stuffing as much information there as they could. After all, the information is easy to store and analyse, it might yield better marketing results, and clients don't seem to object.

One definition of a secure system is a system which does what it is specified to do, and no more. Applying this definition to our client-server example, the server should not receive any output:



The server will probably be shocked to learn how much he was relying on a lapse in securely implementing the system specification, and will want to pencil in at least some output for himself. This can be accommodated in arbitrary ways, at least in theory. For example, say the server is offering a map service which gives cycling directions, the client wants to query this service, and he has a pseudonym ps_c. The client and server could agree that the two-party function is:



where x is a start-destination pair, and q is a suitable path search. With this spec, the server learns the client's pseudonym, and what the city in the query is. Thus, the server could offer city-specific advertising when sending the response back, and keep a simple profile of which (pseudonymous) clients are interested in which cities.

1.4.3 Realizing the specifications

The fact that we can specify completely who should learn what in a multi-party computation may seem inconsequential—a strong intuition is that since the server carries out the client's request, obviously he can learn that request, and any attempt to specify something else is little more than a game. Our response to this intuition is in two parts. Firstly, in this section we have only been specifying requirements, through the two-party function, without any implementation distractions. Thus, even if implementing such requirements is impossible, they are still valid requirements, and useful in that they provide a more complete understanding of data protection (or lack thereof) in multi-party computations. Secondly, implementation tools are in fact available to satisfy all the requirements we have shown, and more. One way to approach this problem is the topic of this thesis.

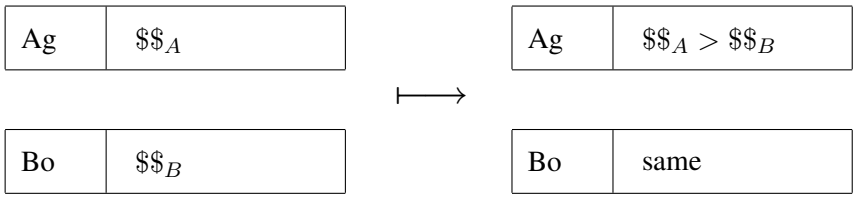
1.5 Secure computation

We have introduced how one might frame the requirement of minimizing or controlling disclosure of information during a computation. Now we will take a few steps in introducing how such requirements can be implemented, and then point ahead to Chapter 6, where we develop our angle on secure computation.

As a working example, say that Agnes and Boris are dining, and come to the tricky part of the evening—deciding who pays. They settle on a simple fair approach to the problem—the richer of them pays the whole sum. However, they are not quite far enough in their relationship to just tell each other what their net worth is. They are in an exemplary instance of the situation we have broadly described: they want to learn something about both their data—which one is larger—but want to avoid the disclosure that comes with the obvious way to do this—tell each other their values and so see which is larger. Solutions to such problems fall under the umbrella of *secure multi-party computation* (SMC), which the theoretical computer science community has worked on for almost three decades.

Illustrating this with our two-party functionality notation, for Agnes and Boris rather than client

and server:



This example, usually referred to as *the millionaires' problem*, has been the introductory example of SMC work for decades. However, it should not imply that SMC is a toy tool—the idea and some of the approaches scale to computing arbitrary functions, with limitations of a similar nature as those on any "normal" computing platform, like a RAM machine—computability, tractability, efficiency.

1.5.1 Existing solutions

Much work has been done on SMC, with various settings, requirements, and adversarial models. One common thread in these works is that the computation participants have to somehow carry out the computation themselves—they cannot pass it off to some (trusted) third party (TTP). This certainly adds spice to the problem, and motivated many deep results, utilizing a broad set of techniques both in constructing solutions, and in setting up the necessary models and definitions to enable reasoning and proofs about the solutions. Currently there are multiple protocols which allow arbitrary functions to be computed securely and without using a trusted third party.

There are however costs to "going it alone"—achieving SMC without trusted third parties. The most obvious one is efficiency—all the protocols are quite expensive. The computational costs of SMC protocols are certainly within a polynomial, mostly linear, and often constant factor of those in a normal setting, but the constants tend to be large enough to render all but the smallest or simplest functions infeasible to compute under SMC. Many of the schemes are also highly interactive, requiring a large number of rounds of interaction among the participants.

Another sacrifice made is that of flexibility—once a secure protocol is constructed with suitable proofs, modifying it to, for example, allow collection of statistics, is likely to be a large undertaking.

1.5.2 Introducing a TTP

Our angle on the SMC problem is that the "fundamental assumption" of SMC—that no actual TTPs exist which satisfy the security requirements—may be relaxed, for two major reasons:

- As we have mentioned, this fundamental assumption forces costs which have a qualitative impact on the scalability and applicability of resulting schemes, and
- Computational devices with very strong security properties have emerged, which provide a plausible way to introduce a TTP and maintain the strong security properties required by SMC.

The main thrust of our study has been to examine how useful such a TTP-assisted approach to SMC can be, and what we can learn from building TTP-based SMC systems. In this quest, we have built two systems, which solve different problems under the SMC rubric.

Private information retrieval

First we built a system to provide *private information retrieval* (PIR). PIR asks how a client can retrieve information from a database, without the database operator learning the identity of the retrieved datum. This is a specific multiparty function, but an important one. It has also received much attention, mostly in the theory community. As usual, the schemes which do not use a TTP have considerable performance problems, which we were able to improve drastically by employing a TTP in the form of a small trusted device, which we refer to as a *tiny TTP*.

Arbitrary secure multiparty computation

Then we generalized, producing *Faerieplay*, a system which uses a tiny TTP to provide SMC of arbitrary functions, specified in a high-level language. Faerieplay improves considerably on the performance of SMC systems which do not use a TTP.

1.6 Contributions

The main contributions of this work are:

- Arguments for flexibility in considering TTPs for implementing secure protocols.
- Formal model of TTP-based secure computation, and first steps in proofs with that formalism.
- A design and implementation[5] of a TTP-based SMC system which is more scalable and practical than existing options. Faerieplay:
  - is much more scalable than was possible with existing SMC systems. We achieve this by combining the standard circuit-based approach to secure computation with the blinded RAM access techniques developed for Oblivious RAMs [41].
  - is usable by non-specialist developers. Previous work had made a lot of progress in this direction [60], but does not scale to complex problems which require use of software engineering techniques like program verification and debugging.
  - runs on current secure devices, specifically the commercially available IBM 4758 secure coprocessor (see Section 4.2.1).
- Discussion and investigation of the usability of SMC on large/complex problems, where the performance overhead of existing techniques becomes prohibitive.
- An exploration of practical circuit programming. In particular we found that the slightly non-standard language used by Fairplay made development of larger programs difficult. We added support for writing Faerieplay programs in a subset of C++, and conclude that using a non-standard language was a mistake.

1.7 Road map

We start with two of our preliminary projects which dealt with *private information retrieval* (PIR). These projects, which we present in **Chapter 2** and **Chapter 3**, introduce in a somewhat informal

[5] All code is available at http://www.cs.dartmouth.edu/~trust/faerieplay/.

manner the ideas behind secure computation based on a tiny TTP, and present our work on a practical PIR system using the IBM 4758 secure coprocessor. In **Chapter 4** we describe tools and techniques which we use in this thesis. Hopefully this chapter will be interesting reading in itself, due to the significant nature of the works it addresses. In **Chapter 5** we outline other related works, which we did not use but which are significant in contextualizing our work. We describe how they relate to the work in this thesis, and how they differ. In **Chapter 6** we expand our presentation of the field of secure multiparty computation (SMC), and examine how it has been approached thus far, what this approach leaves open, and what our take on filling the gaps is. In particular we introduce the idea of using an actual trusted third party (TTP), which provides strong assurances as to its trustworthiness, in protocols.

In **Chapter 7** we begin the presentation of the main piece of this thesis: Faerieplay, a general secure multiparty computation system based on a tiny TTP, again in the form of a 4758. Chapter 7 is an introduction to Faerieplay, with some detailed examples of problems which would benefit from this approach vs. the established SMC techniques, and an outline of the system. With this introduction in hand we move on to a security model and security argument for Faerieplay in **Chapter 8**. In **Chapter 9** we present specifications of the high-level languages supported by Faerieplay for specifying functions, and of the circuit virtual machine used to execute the functions securely using the tiny TTP. Then in **Chapter 10** is our implementation of the Faerieplay compiler and circuit virtual machine, an implementation of *oblivious RAM* which provides another way to achieve tiny TTP-based SMC, and our results in using these systems and comparing them.

Finally in **Chapter 11** we summarize our work and present final thoughts and future work suggestions.

Chapter 2

Privacy-Enhanced Credential Servers

2.1 Introduction

This chapter describes our first project which touched the theme of unwanted information leakage, and how it can be practically addressed using secure hardware. Specifically, we identify privacy risks in centralized *public key infrastructure* (PKI) and authorization systems, and present a design and prototype of a practical solution.

This chapter is based on our earlier publication in the 2003 PKI workshop [46].

Hippocrates advised physicians to "first, do no harm." We would also like to apply this dictum to the design and deployment of new security infrastructure. While examining whether new technology solves existing security problems, we should also ask: does it create new ones?

Identity authentication and access control to resources are essential tools for solving the security problems of interacting within and across organizational boundaries. Advanced cryptography and protocols can provide security and privacy as an organization's members interact with each other and the world. But making this all work typically requires the organization to maintain a centralized credential server, that offers the right tokens and certificates to the participants when they need them. A typical example are the identity certificate directories which form an essential part of X.509 PKI. Here, and in other identity and access control systems, ease of management and implementation is

largely provided by *centralization* of data and control.

Although the functionality they enable can solve security and privacy problems, the existence of centralized servers creates new ones: because many interactions by a user require queries to the credential server, it is possible for the credential server to learn a great deal about these interactions, by monitoring who is asking for which credentials. Securing interaction against outside adversaries thus has the unwanted side-effect of enabling attacks on privacy by organizational insiders.

Section 2.2 will introduce the deployed systems which exhibit privacy problems we wish to address. In Section 2.3 we elaborate on the privacy problems introduced by the centralized components of these systems, and the trade-offs between convenience of implementation and excessive exposure of data to the system operators. Section 2.4 outlines the technology that we and others have helped produce, that may address these issues. Section 2.5 presents our solution plan for the privacy problem. Section 2.6 and Section 2.7 then present fully our design and prototype that applies the technology to solve the privacy problem. Section 2.8 and Section 2.9 present the experimental results and analysis from our prototype.

2.2 Our setting

The problems we wish to address are present in currently deployed systems, which we introduce in this section.

2.2.1 X.509 PKI certificate directories

In the standard approach to PKI, hierarchies of *certificate authorities* (CAs) and users emerge that mirror organizational hierarchies.

Using public-key cryptography requires knowing the public key of the other party, and being able to bind this key-holder to some relevant real-world property (such as identity or role). Within a user population served by a single CA, a *public key certificate* provides this information; in more complex hierarchies, a multi-step *certificate path* may be necessary. To provide these certificates, organizations set up directories, which are usually implemented via the *lightweight directory access*

protocol (LDAP).

Within a population, if Alice wants to send a secret message to Bob, she needs to obtain Bob's public key. Typically, she asks a directory for this. If Bob receives a message from Alice and wants to verify a signature, he needs her certificate. If Alice did not provide this with the message, then Bob needs to ask the directory; even if Alice did provide it, Bob may wish to check if it is still valid, which can require asking the directory.

Across different populations, parties may need to ask directories for additional certificates to construct trust paths. In more general settings, such as trust decisions based on attribute certificates as well as identity certificates, additional directory queries may be involved.

2.2.2 Shibboleth federated authorization

The Shibboleth project is a developing *federation* system for authorization. It seeks to enable different information providers to serve different clients, such that each provider maintains only one access control policy, each client only one set of credentials, and everyone uses the same enforcement mechanism [34, 76]. In the absence of federation, each provider-client pair may have to develop and deploy individual access control arrangements, which is not scalable.

For Shibboleth, a user is assumed to have a *home site*, which can provide information about her. The resources are located at the *target site*. The steps taken when the target site receives a request from a user are as follows (This is also illustrated in Figure 2.1).

1. The *Shibboleth Indexical Reference Establisher* (SHIRE) at the target site establishes an opaque *handle* for a user. The first time a user makes a request, the SHIRE contacts the home site *Handle Service* (HS) (via an http redirect), to ask it for a user handle. The HS would prompt the user to log in, or re-use any available *Single sign-on* (SSO) credentials without needing another login. The SHIRE arranges to store the user handle in a context of some kind, eg. using HTTP cookies, so that in subsequent requests the protocol with the HS can be avoided.

2. The SHIRE passes the user handle to the target site *Shibboleth Attribute Requester* (SHAR), which contacts the home site *Attribute Authority* (AA), with the user handle, to obtain the

user attributes required for the target site to check if the user satisfies the resource's access control policy.

3. The AA decodes the user handle to obtain a user identifier which it uses to look up the requested attributes and return them to the SHAR. The AA also checks the user's attribute release policy before releasing attribute values to the target site.

4. The target site can then use the user attributes to decide whether to allow the user access.

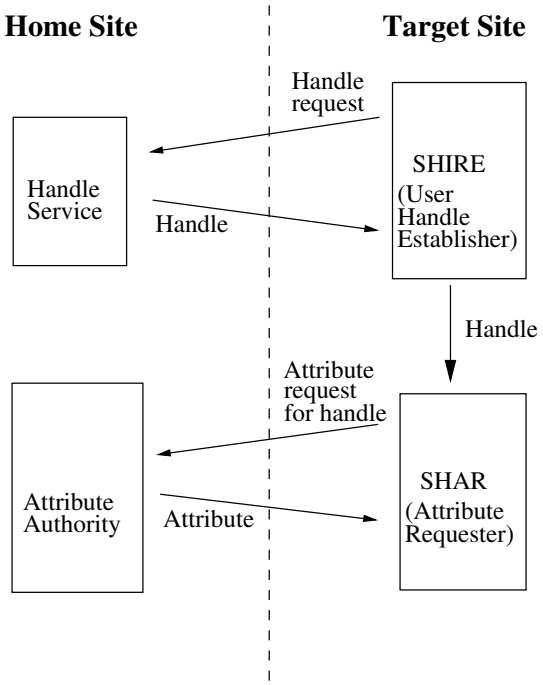


Figure 2.1: A simplified view of the Shibboleth procedure. On receiving a request, the target site establishes a handle for the user, then requests attributes for that handle, and makes an authorization decision based on the attributes.

Shibboleth privacy provisions

Shibboleth claims two main user privacy features:

- The target site never sees an actual user identifier, and must base its access control on non-identifying user attributes[1].

[1]The problem of ensuring that attributes are actually non-identifying is a difficult one, and not addressed by Shibbo-

- The user has the ability to control which attributes are disclosed to which target sites. Thus, he can for example avoid disclosing attributes for which he has unusual values and which may thus help identify him.

2.3 The privacy problems

Both X.509-style PKI and Shibboleth's centralized home-site structure allow considerable disclosure of user information to the respective authorities. The reason this is not usually considered a problem is that these authorities are "on the same side" as users at that site. However, insider attacks are widely recognized as a major problem, and this also applies to attacks on users using the information thus disclosed.

2.3.1 PKI certificate directory

Consider the privacy implications if the adversary Mallory operates the PKI certificate directory at Alice and Bob's organization. Alice may be using encryption on her message because she wants to keep the contents secret. But because she needs Bob's certificate, Mallory knows that Alice is sending a message to Bob. If Bob needs to obtain Alice's certificate (or check whether it's valid), he must ask the directory, so Mallory knows that too.

Thus, if Alice and Bob take the precautions of using protected channels for their message, Mallory still learns of the interaction via the PKI at the end points. Even if Alice and Bob did not take precautions, the use of PKI lowers the work required for Mallory—rather than monitoring network traffic, she can just log queries to a directory.

2.3.2 Shibboleth attribute authority

As described before, Shibboleth goes to some lengths to limit disclosure of user information to service providers, and to allow users to control what is disclosed about them. What is not covered

leth [42, 85]. The provision for user-specified attribute-release policies helps somewhat, by allowing the user to block release of attributes he believes may identify him. This mechanism is fairly limited though. A more comprehensive mechanism would allow blocking of a *set* of attributes which, taken together, could be identifying. Going even further, such dangerous sets could be dynamically computed using statistical techniques.

though, is that as a consequence of these features, and the convenience of centralizing identity and attribute management at the home site, the home site can learn a lot about their users' online activities—which target sites they visit, and in some cases even the exact URLs.

Shibboleth effectively trades information disclosure to target sites, for disclosure to the home site. Clearly, keeping information from the target site is beneficial to the user, but the benefit of the trade off is less clear. Pre-Shibboleth, provided that target sites do not share information, then each site can only learn about a user's activities on that site. Now, the AA receives information on *all* of a user's activities, across all target sites. Additionally, the home site is likely to have a much more complete base of information on a user than target sites have. This further enhances the ability of the home site to build a comprehensive user dossier.

As a concrete example, say John from Dartmoor College occasionally needs access to `http://webofscience.com/LegalizeIt`, `http://salon.com/archives/palestine/`, and `http://pop-music-journal.com/sexpistols/`, and these sites require Shibboleth authorization for users of subscribing institutions[2]. The SHAR at each web site will ask the AA of Dartmoor for some attributes of John, by passing John's opaque session handle (opaque to the sites, not to the AA), and the URL being requested. The URL is passed to the AA so it can decide which attributes to release, based on Attribute Release Policies. The attributes will likely be non-identifying, like confirmation of institution membership, or age, so the sites do not receive any personal information about John. The AA at Dartmoor however does see which sites (including URLs) are asking for attributes for John, and if Mallory at the AA wished to do so, she could log this information.

2.4 Background

In this section, we examine some relevant technology which can enable a solution to the server privacy problem as described above.

[2]Perhaps the sites also offer pricey personal subscriptions.

2.4.1 Secure coprocessors

A secure coprocessor is a small general purpose computer armored to be secure against physical attack, such that code running on it has some assurance of running unmolested and unobserved [95]. It also includes mechanisms to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [77]. The coprocessor is attached to a *host* computer. Since the secure coprocessor we use is implemented as a PCI card, we sometimes refer to a secure coprocessor as a *card*.

We describe secure coprocessors in more detail in Section 4.2.

2.4.2 Private information retrieval

The problem of *private information retrieval* considers how a user can obtain a particular record from a large data set that a server offers, without the server learning anything about which record was requested. We will illustrate the problem with reference to one of our motivating examples: a certificate directory. In this case, the directory holds a database of public key certificates. The actual certificates are not private, and this do not need to be encrypted. A user, say Agnes, wants to obtain the certificate of Boris from the directory. A private retrieval, in the PIR sense, would guarantee that the directory *cannot* learn anything about which certificate Agnes retrieved. If it holds N certificates, all it knows is that Agnes obtained one of those N.

More specifically, when we talk about hiding information from "the directory", we mean that no human or external process somehow related to the directory can learn the protected information. For example, administrator or root users on the machines running the directory should not be able to learn any protected information, or be able to pass such information to others.

What is not quite PIR

Several ideas may come to mind about how to achieve PIR. One could be that if the records were somehow encrypted, the server could be kept in the dark about the retrieved records. To examine this further, note that for this scenario to make sense, the data records must have been generated

by another party, who then provides them to the data server in encrypted form. The setting could be that the data server is a mirror of the data, and just serves to distribute it, ie. it is acting as a distribution server, which we will call DS. Let's call the data provider DP.

A simple approach is that the DP keeps the record names in cleartext, and just encrypts the contents, before handing off to DS. Then, when Agnes makes a request for the record named "Boris", DS can look up the name, obtain the encrypted contents, and send them to Agnes. This will keep DS in the dark about the record contents, but it has done nothing to hide the identity of Agnes's retrieval—DS has learned that she wants Boris's record, and no other.

A more sophisticated approach would be that DP encrypts both the record contents and the record name, before sending to DS. The users of the data would be somehow privy to the decryption functions of both record names and contents—this is plausible, as it is required for any form of distributed hosting of encrypted data. Then, Agnes would obtain the ciphertext of 'Boris', $E('\text{Boris}')$, before requesting that record from DS. DS would not know Agnes wants Boris's record, but can still use standard data lookup techniques to obtain the desired record and send it back.

However, this approach does not fully achieve the desired properties of PIR. In particular, it does not hide from DS the relationships between retrievals: DS can always learn whether a request is for the same record as some previous request, and how popular records are. Moreover, if DS colludes with any user of the service, or manages to impersonate a legitimate user, it can obtain the decryption functions which would allow it to learn record names and perhaps contents.

Theoretical PIR

Theoretical computer scientists developed many algorithms (e.g., [25, 23]) through which users, servers, and sometimes other parties could carry out computation and achieve PIR.

Two threads of theoretical PIR work exist. The first assumes that the distribution server consists of k non-colluding servers, with replicated copies of the database. The security of these schemes relies on the servers not sharing data related to client queries. This is a non-trivial assumption, but nonetheless underlies many proposed PIR schemes. For a database of N bits, two illustrative schemes in this space provide:

- sub-linear communication between the client and servers [12], eg. for $k = 3$ non-colluding servers, communication is $\mathrm{O}(N^{1/5.25})$; for $k = 4$, $\mathrm{O}(N^{1/7.87})$. The servers need to do $\Omega(N)$ online computation.
- $\omega(N)$ communication, $\omega(N)$ online computation, and poly(N) pre-processing [13].

The other approach to theoretical PIR makes the more standard assumption of a single server which is computationally bound in the usual sense, eg. finds exponential algorithms infeasible. In this case, a best-of-breed scheme has poly-logarithmic communication ($\mathrm{O}(\log^4 N)$), but linear work for the server, with a large constant [23]. Concretely, the server's work is one mod-exp per bit in the dataset—too expensive for large N. For example, if $N = 10,000$ items of $8,000$ bits each, retrieving one item would need 80 million mod-exp operations, which take about 24 hours on a 2GHz AMD Opteron-280 64-bit server using openssl (`openssl speed rsa1024` reports 900 private key RSA operations per second).

Practical PIR

Smith and Safford [79] then proposed the problem of *practical* PIR: using existing systems, can we provide PIR along the lines of a Web model: the user establishes an SSL session, issues a request, waits a short while, then receives the response?

Their solution used *Commercial off the Shelf* (COTS) secure coprocessors (SCOP) and assumed that a coprocessor can only hold a fixed small number of records internally at one time. Their scheme consists of handling a query to a PIR server by having a secure coprocessor read sequentially through *all* the records in the database (which is kept on the host), keep the correct record internally and return it to the user. The running time of a query is linear in the database size. Careful data structures permit the query processing work to be divided evenly across several SCOP devices, but the linear time bound per query is still problematic.

System setup for practical PIR

In the Smith/Safford scheme, and in subsequent works including ours, an important assumption is that the SCOP has only a small internal memory, and can only accommodate pieces of the dataset

at a time.

Concretely, the setup for practical PIR is that the database consists of N size-normalized records, numbered from 0 to $N - 1$. They may or may not be originally encrypted, depending on whether the contents need to be kept private from the PIR server. If the contents do not need to be hidden, then the input to the server can be plaintext records.

The records are stored on the host, and accessed from the SCOP via a simple API to the host:

- `read_record : position → Record-text` and
- `write_record : (position, Record-text) → unit`.

The Record-text will be encrypted for confidentiality and will have a *message authentication code* (MAC) applied for integrity. An assumption is that at least two records can be stored on the SCOP at a time.

Size normalization We assume that the records do not range greatly in size, and can all be padded to the same size. If they do vary in size, they could be split such that each piece is within a suitable size range. Let $s_{\max}$ be the number of pieces of the largest record. To assure retrieval privacy, each retrieval would need to look-up and return $s_{\max}$ pieces, even if some of them are dummies.

Asonov's scheme

Asonov et al. [7] improved the Smith-Safford scheme by decreasing the processing time for a query at the expense of a periodic preprocessing step.

We note that Asonov's algorithm is essentially the same as the "square root" algorithm developed earlier as one solution to the *Oblivious RAM* (ORAM) problem [41]. We elaborate more on the relationship between ORAM and PIR in Section 2.10.

The scheme is divided into two parts:

1. Preprocessing, where the database is randomly permuted, such that the host has no information about the positions of records in the permuted version. This means not only that the

host should not know which input record corresponds to which permuted record, but also should not know any relationships among the permuted records, eg. if permuted record r_1 corresponds to plaintext record i, then permuted record r_2 corresponds to $2i$.

2. Retrieval, where the SCOP fetches records from the permuted database.

Preprocessing Asonov's scheme permutes an N-item database in $\mathrm{O}(N^2)$ time, as follows. The SCOP generates a uniform random permutation vector `V` such that record number `V[i]` will be sent to position `i` of the permuted database (call it D_π). Then, for each position `i`, the SCOP sequentially reads *every* record from the host, keeps record number `V[i]` internally, and writes it out to position `i` of the permuted database. The host does not know anything about `V[i]`, so does not learn anything about the record going into position `i` of D_π. This process is equivalent to applying the Smith-Safford scheme N times, each time populating one element of the permuted database D_π.

Retrieval For the first record retrieved after a permutation, the SCOP simply gets the record's permuted position from its permutation vector `V`, and retrieves that element from D_π.

For the r^{th} retrieval (call it record R_r) after a permutation, the SCOP re-fetches from D_π *all* $r-1$ records previously retrieved, then fetches and returns R_r. If R_r was in the $r-1$ already retrieved records, it is kept internally to be returned, and the r^{th} record fetched is a random one not previously touched. This re-fetching is required to hide the relationships among retrievals. If it was not done, the adversary (on the host) could learn whether any retrieval is the same or different from any other retrieval.

The retrieval algorithm has a running time which increases with each retrieval, and after N retrievals it degenerates into $\mathrm{O}(N)$ time. To restore fast retrieval times, the scheme requires that the SCOP periodically generates a new permuted database. Given a new permuted database, using a fresh permutation, the retrieval algorithm resets the counter r to zero, and starts a new retrieval *session*.

2.5 Solving the credential server privacy problem

The rest of the chapter presents our practical solution for the credential server[3] privacy problem by implementing the server using PIR. There was no other PIR implementation at the time of this work. PIR provides the assurance that, upon receiving and servicing a request from a target site, the AA would only learn that that site wants the attributes of *some* user[4]. Users could then have assurance that the system operator is not observing their queries.

The question, then, is can we build a credential server using PPIR technology and COTS hardware that performs reasonably well?

We decided that the outside interface should be *Lightweight Directory Access Protocol* (LDAP), currently the most popular directory access protocol; we will make the limiting assumption that the querier can only specify one fully named record. This limitation is not unreasonable—asking a directory for the certificates of all Bobs does not seem like an indispensable operation.

We then consider the issues: Section 2.6.1 discusses extending PPIR to deal with *named* records; Section 2.6.2 presents a new approach to the permutation step that decreases the time from $O(N^2)$ to $O(N \log N)$; Section 2.7.1 discusses our PPIR setup; Section 2.7.2 discusses our prototype implementation; and Section 2.7.3 discusses the overall credential server architecture.

2.6 Extensions

We developed several extensions to the then-current PPIR work, to solve problems which arise when applying the technology to credential servers.

[3] Credential server examples include the Shibboleth *Attribute Authority* (AA), and a PKI X.509 certificate directory.

[4] We note that for Shibboleth, a more comprehensive strategy for protecting the user's identity from his home site would require making the *Handle Service* (HS) privacy-protected too, or the AA could make a reasonable guess at the identity of a request for attributes which comes soon after a login for a known user at the HS. A simple improvement would be to enable the *Single sign-on* (SSO) mechanism at the home site to let the user (or his browser) hold a long-lived Shibboleth opaque identity handle, and only require contact with the HS when this handle expires. Then the requests to the HS could be infrequent enough that they do not provide much information about subsequent requests to the AA.

2.6.1 Named records via hashing

Real database records are usually named as opposed to numbered. The approach we took to dealing with names in this prototype was to implement the database using hashing with chaining. Thus, hashing a record name yields a bucket number, and inside this bucket is the needed record. We effectively ran the Asonov scheme with numbered buckets. Within a given bucket, a record was retrieved by reading in all the records sequentially and keeping the right one.

The hash function we used is a simple and standard one, and does a good job of distributing the names. It is the same as the hash function used for strings in the Java language library[5]:

```
hash(str[0..i]) = hash(str[0..i-1]) * 31 ^ str[i].
```

Another more complicated name resolution option we considered was a *perfect hash function* [49]. Such a function needs to be constructed especially for the current set of keys, but then hashes each name into a unique bucket, without any collisions. Perfect hash functions are not very compact—they require an index whose size is comparable to the key set's size. However since this index consists of integers smaller than the key set size[6], it would certainly be a lot smaller than storing a full name table. We did not implement perfect hashing in our prototype though.

2.6.2 Private permuting with Beneš permutation networks

We have identified and implemented an alternative and faster permuting algorithm which we shall describe here. It is based on a *Beneš Permutation network* - a network of *switches* wired together in a fixed manner and able to perform any permutation of its inputs, just by setting the switches [89].

A switch has two inputs and two outputs, and it may cross the inputs, or pass them on straight, determined by a switch bit. By propagating values along the wires and through the appropriately set switches, any permutation of the input can be produced at the output. An example 4-input network is shown in Figure 2.2. A permutation network for N inputs can be built recursively as shown in Figure 2.3. [27] This network clearly consists of $\Theta(N \log N)$ switches. Setting all the switches to

[5] See http://java.sun.com/javase/6/docs/api/java/lang/String.html#hashCode().

[6] At the time (early 2003), the Dartmouth PKI held about 5,000 certificates, for which a 2-byte word would suffice. Since then the PKI has grown to about 30,000 certificates, which can still be covered by a 2-byte word.

achieve a given permutation is possible with a $\Theta(N \log N)$ algorithm [88, 1].

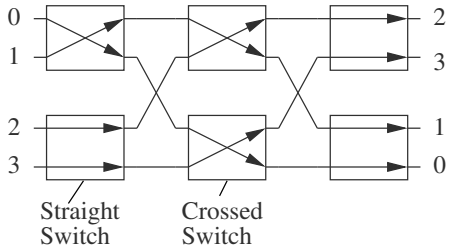


Figure 2.2: A Permutation network with 4 inputs, performing the permutation $\langle 2, 3, 1, 0 \rangle$.

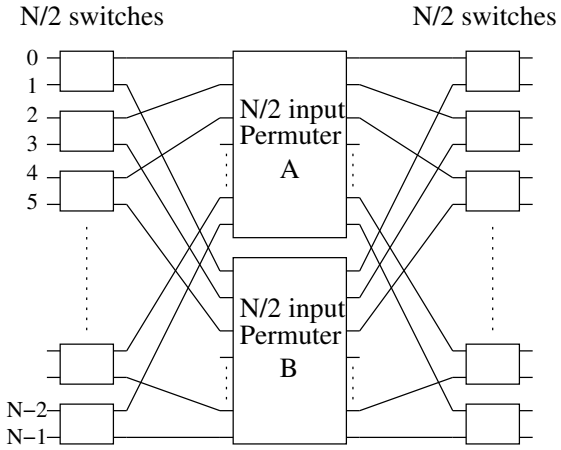


Figure 2.3: A Beneš network on N inputs built from N switches and 2 networks on N/2 inputs each. Note that the Beneš network is a particular instantiation of the more general Clos network. Each of the switches on the left have one output wired to permuter A, and one output to B. The switches on the right have their inputs similarly connected. This construction is straightforward but not entirely minimal—N/2 switches can be removed while still enabling any permutation to be executed [89].

Applying to SCOP-based private permuting

A permutation of a dataset using a Beneš network would consist of the SCOP internally generating a random permutation vector (as before), generating a network for its chosen permutation, and then executing all the switches in topologically sorted order (eg. column-major). Generating the network (ie. the switch settings) for a given permutation takes $\Theta(N \log N)$ time, as does executing the switches.

A single switch can be interpreted for our permuting algorithm as follows. The coprocessor reads in two records (the inputs), switches their places if its switch-bit is set, and writes them out to the same two positions. The host should be unable to tell if the two records were switched or not. This can be achieved by re-encrypting the records for example, assuming semantically-secure encryption is used. Concretely this can be realized by using a symmetric block cipher in *Cipher Block Chaining* (CBC) mode, and using a fresh random *Initialization Vector* (IV) for a re-permutation.

2.7 Prototype

2.7.1 System setup

Our prototype runs in the IBM 4758 model 2 secure coprocessor with Linux [78]. The 4758 is a commercially available device, validated to the highest level of software and physical security scrutiny currently offered—FIPS 140-1 level 4 [82]. It has an Intel 486 processor, 4MB of RAM and 4MB of FLASH memory. It also has cryptographic acceleration hardware. It connects to its host via PCI. Our host runs Debian Linux, with kernel version 2.4.2-2 from Redhat 7.1 as needed by the 4758/Linux device driver.

Linux is an experimental operating system for the 4758, which runs IBM's proprietary embedded operating system CP/Q++ in production, but Linux has considerable advantages in terms of code portability and ease of development—our prototype is written in C++, making extensive use of its language features and the Standard Template Library, and it runs fine on the 4758 with Linux.

Another major advantage of using Linux on the 4758 is that almost the same code can be compiled to run on a normal Unix host. We have abstracted the SCOP-host communication, and the cryptographic functionality, and implemented those differently for running on the 4758, and for running on a Unix host. On the 4758, we use SCC (Secure Cryptographic Coprocessor) sockets for communication, and a device-file-based interface to the 4758's cryptographic hardware for crypto. When running the SCOP code on a Unix host, we use UDP for communication[7], and OpenSSL for

[7]We have only used processes co-located on one machine so far. In this setting, the lack of a reliability guarantee with UDP has not been a problem. If the SCOP process and host process were on separate machines, UDP may not suffice.

crypto.

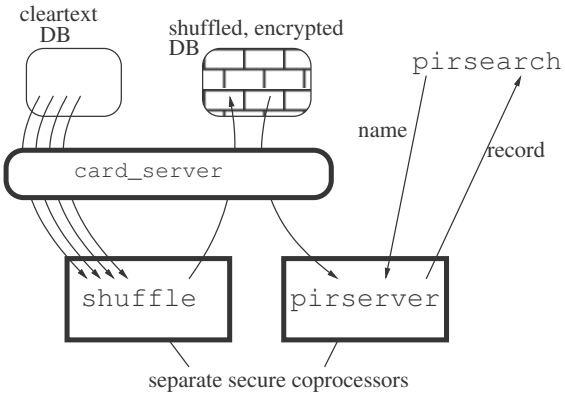


Figure 2.4: An overview of our PIR prototype. The SCOP programs are `shuffle` which does the permutation, and `pirserver` which handles retrievals. On the host, a small client program `pirsearch` performs a search by passing the name to `pirserver`, and `card_server` handles DB access requests from the SCOP programs. Communication between the SCOP and `card_server` is over SCC sockets, one of the mechanisms provided for 4758 Linux to talk to the outside. We serialize data using the External Data Representation (XDR) library in the Sun RPC package. XDR is specified in RFC 1832 (See eg. http://www.faqs.org/rfcs/rfc1832.html. XDR data types and functions are declared in `rpc/xdr.h`, and the implementations are included in current `libc` versions.

2.7.2 PPIR implementation

An overview of the components of our PPIR prototype is shown in Figure 2.4. Our implementation of retrieval with hashing is illustrated in Figure 2.5. We implemented the permutation step using both the naive algorithm described in Section 2.4.2, and using a Beneš network. We make our code available at http://www.cs.dartmouth.edu/~sasho/privdir/.

2.7.3 System architecture

An overview of the whole system is shown in Figure 2.6. It consists of the PPIR system described in Section 2.7.2 connected to an OpenLDAP server by means of a *shell backend*. The OpenLDAP server has a variety of ways to access the actual data it provides LDAP access to. One of them is to run a shell command to retrieve or update records.[8] We wrote a Perl script to allow the OpenLDAP

[8]There are more operations besides query and update which are idiosyncratic to the LDAP protocol.

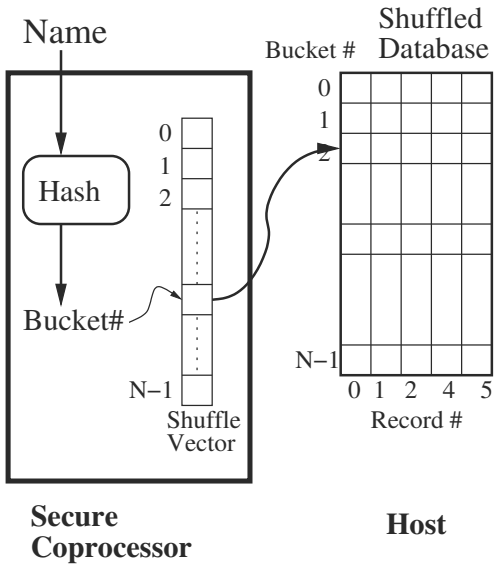


Figure 2.5: The private retrieval procedure. Once the SCOP has identified the correct bucket number, it retrieves all the records in that bucket, and keeps the correct one.

server to use our `pirsearch` program (see Figure 2.4). We tested this whole setup by sending LDAP queries with the `ldapsearch` LDAP client to our PIR prototype.

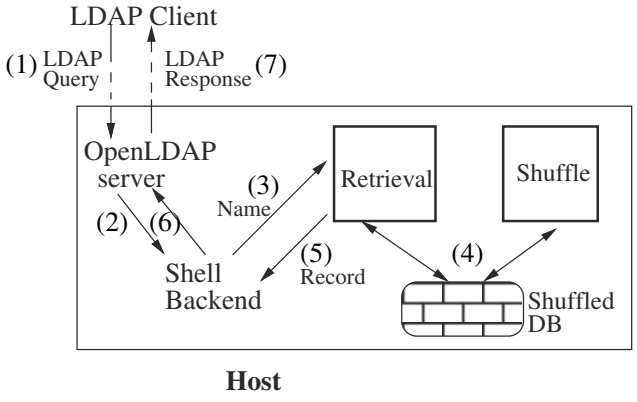


Figure 2.6: System Architecture. OpenLDAP is used to provide the gateway between our PPIR server and LDAP clients.

This setup is a temporary way to achieve the connection to LDAP, and it is clearly not *root-secure*—secure even against an adversary running as root on the host—as queries are in the clear on the host before being handed to `pirsearch`. Our plan for a secured connection all the way from

the client to the retrieval coprocessor is shown in Figure 2.7. It will make use of LDAP over SSL, and use OpenLDAP libraries for parsing of LDAP queries, and construction of LDAP responses.

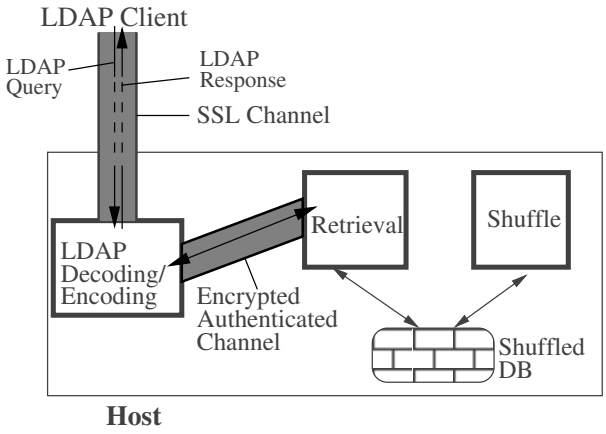


Figure 2.7: Final System Architecture, using a third coprocessor to handle LDAP and SSL operations. A separate coprocessor will likely be needed for these tasks because of space restrictions inside the coprocessors.

2.8 Experimental results

2.8.1 Performance measurements

Initially, we implemented the naive $O(N^2)$ permutation algorithm, with the results shown in Table 2.1. The database size N was 1000, and hashing had resulted in every bucket holding 5 records. The times shown are for just one pass of the permutation algorithm, which consists of the SCOP reading each of the buckets once.

Record size (bytes)	**Time to Read $N = 1000$ Records (sec)**
115	18
530	24
1345	34

Table 2.1: One pass (out of $N = 1000$) of the permutation step, vs. Record size.

After implementing the permutation using a Beneš network, we performed testing of the proto-

type running continuously for some time, by having distributed LDAP clients[9] send queries periodically, asking for randomly picked legal names. The query processing times of one of the clients are shown in Figure 2.8.

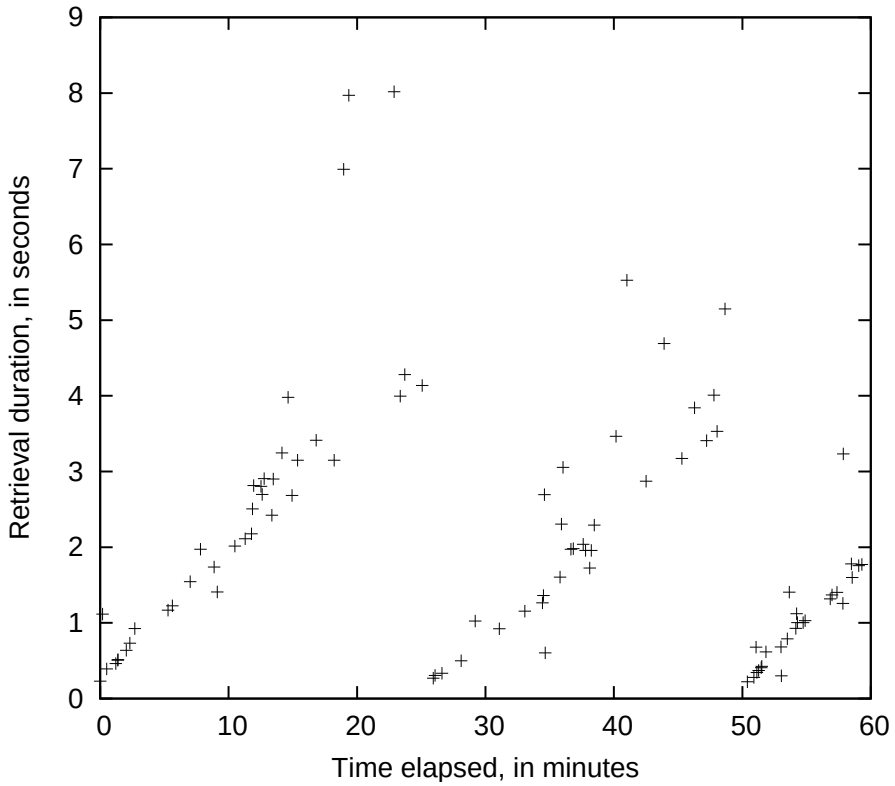


Figure 2.8: Retrieval times of an LDAP client. It was one of 5 independent clients, each sending randomly spaced queries, on average one every 40 seconds. The database size was 1024 records, each of about 1500 bytes[10]. The server was set up with one SCOP continuously producing permuted databases, and one handling queries. Each point corresponds to a single query. The increase in response time as a retrieval session progresses is clear, as well as the transition into fresh sessions. At the beginning of a session, the queries are being serviced quickly, and the different query processes are not contending much for the single SCOP. Towards the end however, as queries take longer to process, there is some contention, and an increased variance in the latency.

Since database permutation is the bottleneck of this system, in Figure 2.9 we show the running times of a single permutation, using a Beneš network, against two parameters: database size and record size.

[9] `ldapsearch` from the OpenLDAP package.

[10] A Base64-encoded user certificate in the Dartmouth pilot CA was 1250 bytes in size.

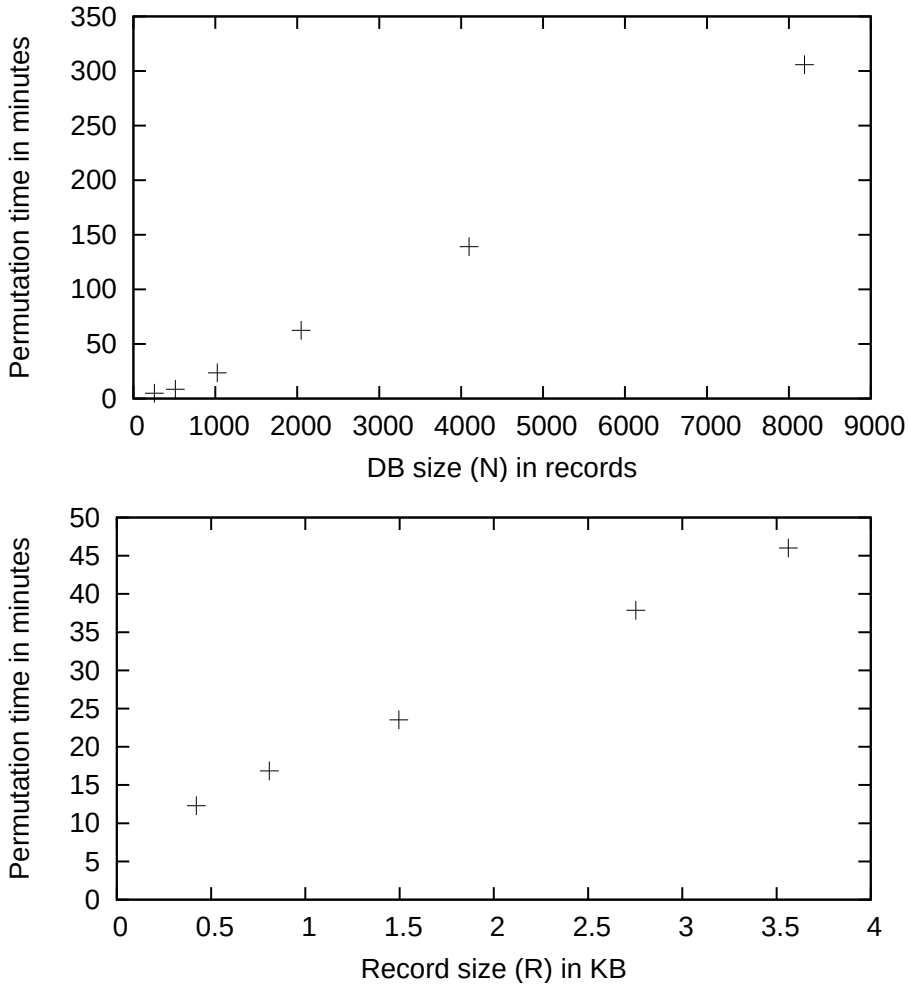


Figure 2.9: Running times of a single database permutation, using a Beneš network, against (1) database size N, with record size fixed at 1500 bytes; (2) record size M, with database size fixed at 1024 records.

2.8.2 Hashing overhead

The main price of using a fixed hash function is that collisions inevitably occur, and in this case they are particularly damaging—since all buckets need to look the same to the host, they must all hold the same number of records. In our experiments the largest bucket received 5 records from the hash function, so we had to pad *all* the buckets to 5 records, thus having 4N dummy records. This has the effect of increasing run times for the whole system by a factor of 5, compared to what it would be without the name hashing overhead.

2.9 Analysis

2.9.1 Permutation

Several observations arise from the naive ($O(N^2)$) permutation measurements in Table 2.1. Firstly, a linear relationship between the record size (m) and read time (seconds) for 1000 records (t) is $t \approx 16 + m/75$. This confirms that there are considerable overhead costs to the host-SCOP communication, and that maximizing the amounts of data transferred in a single host-SCOP I/O burst is desirable.

Secondly, the whole permutation, which consists of N scans through the whole database, would in this case take $18,000$ seconds, or 5 hours, for the smallest record size. This brings into question the real usability of the scheme with naive permutation for larger but quite realistic database sizes like $10,000$ records—the prediction in that case is 500 hours, or almost 3 weeks—permute that! This observation led to the (predictable) conclusion that permuting with a $O(N^2)$ algorithm was a dead end, and prompted us to seek more efficient means, which led to the Beneš network.

Using the Beneš network, times improved as expected. Time measurements against two variables: database size N and record size M, are shown in Figure 2.9. A permutation of a $8,200$ item database took a much more reasonable 5 hours. As we shall see below, this result appears to provide a basis for a usable system, especially if the name hashing overhead is removed.

2.9.2 Name resolution

As we wrote above, hashing required us to introduce 4N dummy records into the hashed database. This brings about a factor of 5 increase in the running time of most procedures in the system. If we use a perfect hash function, each record name would hash to a unique record number, and there would be no need to carry the dead weight of dummy records. Thus our current running times for shuffling *and* retrieval could be reduced by up to a factor of 5. The cost would be the complexity of computing the perfect hash function when the name set changes (which should be infrequent).

2.9.3 Consolidation and feasibility

When thinking of how to compose an actual directory system using the algorithms we have described and prototyped, it will be essential to run the permutation process on multiple SCOPs, in order to increase the rate at which fresh permuted datasets become available for the query process.

We will now derive two quantities of interest:

- How many permuter SCOPs are needed to supply enough permuted databases to ensure that queries can be serviced at some specified latency $T_Q^{\max}$?
- How small can $T_Q^{\max}$ be if we have a given number of SCOPs for permuting?

The following table summarizes the quantities used in this analysis:

Symbol	**Meaning**
r	Retrieval number in session
$T_Q(r)$	Duration of query r
$T_*(r)$	Total duration of retrieval session with r queries
$T_Q^{\max}$	Maximum query duration; a property of a particular deployment.
M	Abbreviation for $T_Q^{\max}$
$k(M)$	$\max r$ s.t. $T_Q(r) < M$, ie. k is the number of queries in a session where all queries are handled in the time bound M.
k	$k(M)$, with parameter M assumed implicitly
$T_*(M)$	$T_*(k)$ where $T_Q(k) < M$, ie. time of a longest session where all queries are handled in the time bound M.
$T_\pi(N)$	Duration of permutation for DB of N elements.
T	$T_\pi(N)$, with parameter N assumed implicitly.
S	Number of SCOPs dedicated to producing permuted DBs

From the query processing algorithm, and confirmed by the data shown in Figure 2.8, we can conclude that $T_Q(r)$ is linear in r. Let the coefficients be a and b, ie. let

$$T_Q(r) = a + br$$

The total time for r adjacent queries is the integral of the above linear:

$$T_*(r) = ar + \frac{br^2}{2}$$

If we are given a maximum query latency M, the largest value of r such that query number r is processed quickly enough is:

$$k(M) = \frac{M-a}{b} \tag{2.1}$$

Using the value for $k(M)$ from Equation 2.1 above, in terms of M the duration of a longest acceptable session is

$$\begin{aligned} T_*(M) &= \frac{a(M-a)}{b} + \frac{b\left(\frac{M-a}{b}\right)^2}{2} \\ &= \frac{1}{2b}(M^2 - a^2) \end{aligned} \tag{2.2}$$

Given S permuting SCOPs, it takes time $T_\pi(N)/S$ to produce a single permuted DB, which is used for one session. For maximum utilization, we want that the permuting SCOPs produce DBs as quickly as the query SCOP consumes them to answer $k(M)$ (abbreviated to k) queries:

$$\frac{T_\pi(N)}{S} = T_*(k) \tag{2.3}$$

Given these equations, we can easily derive our two relationships of interest, as follows.

S in terms of M How many permuters do we need to provide a maximum latency of M? Using Equations 2.3 and 2.2, we have that

$$\begin{aligned} S &= \frac{T_\pi(N)}{T_*(k)} \\ &= T_\pi(N)\left(\frac{2b}{M^2 - a^2}\right) \end{aligned} \tag{2.4}$$

M in terms of S What best (lowest) latency can we provide given S permuters? From Equation 2.4, and abbreviating $T = T_\pi(N)$, we have that

$$\begin{aligned} S &= \frac{2bT}{M^2 - a^2} \\ M^2 - a^2 - \frac{2bT}{S} &= 0 \\ M &= 1/2\left(1 \pm \sqrt{4a^2 + \frac{8Tb}{S}}\right) \\ M &= 1/2\left(1 + \sqrt{4a^2 + \frac{8Tb}{S}}\right) \end{aligned}$$

The selection of the positive square root is clear, as the negative square root would yield negative values for M (for this to be false, we would need $4a^2 + \frac{8Tb}{S} < 1$, which is not the case for any measured value of T).

Estimates for various feasible combinations Using the equations derived above, and the values for a, b and $T_\pi()$ from the experiments, we can project the following parameter combinations.

DB size N	Permuters S	$T_Q^{\max}$	$T_Q^{\max}$ without hashing (see below)
8192	3	37.2	12.8
8192	7	24.5	8.5
4096	3	25.3	8.8
4096	7	16.7	5.9

If we employ a record name-to-index mapping using a low overhead method, the measured times given here should improve. Assume in particular that we can eliminate the record-size overhead of name hashing, which would reduce the record size by a factor of 5. Using the results shown in Table 2.1 and Figure 2.9, we can estimate that this record size reduction would reduce the permutation time and the query handling time by about a factor of 3. Note that we do not get a perfect speedup of a factor of 5, as the fixed costs like communication are still fairly prevalent at the record sizes of interest, eg. 1500-byte records, inflated to 7500 bytes because of chaining.

2.10 Related work

Work on anonymous credentials, for example by Stefan Brands [19, 20], and IBM's IDEMIX system [24] offers a more general solution to privacy concerns in current PKI. However, these approaches require a complete replacement of the structure as well as the implementation of currently deployed PKI. Our proposals here are more incremental and have the potential to be deployed until such time as a more general overhaul of PKI is in place. Additionally, accountability is difficult to provide with anonymous credential systems—disclosing an identity under certain conditions, while preserving good privacy properties in normal operation is a difficult problem. Progress has been made in supporting revocation of access for misbehaving users, without use of a TTP [22, 87]. This does not allow de-anonymization though, so limits the consequences of bad behavior to denying future access. De-anonymizing bad users would probably impose a higher burden of proof than blocking access, and a programmable TTP seems a good environment to check such a proof.

The Oblivious RAM problem—a small trusted CPU running a program on a large untrusted RAM without revealing anything about the program's behavior—is almost isomorphic to the hardware-assisted PIR problem addressed in this chapter [41]. The "square root" algorithm presented there is structurally the same as what we have implemented here. The contribution here is the use of Beneš networks to perform the permutation of the database, as opposed to bitonic sorting networks in ORAM, which are a factor of $O(\lg N)$ larger, but require less memory to perform. The ORAM authors also present an asymptotically superior algorithm–the polylog solution. We have not implemented this algorithm, but have compared it to the square root algorithm we used in terms of concrete operations: reads and writes to the RAM. Our analysis reveals that polylog should overtake the square root algorithm only for $N > 2^{20}$. This database size is infeasible on the 4758 using these algorithms, so the polylog algorithm is not of interest in our setting. The new extension of the ORAM polylog algorithm by Williams and Sion [91], which we examine in Section 5.1.2 on page 81, may provide a much more efficient alternative here.

Asonov, concurrently to our work, proposed another approach to shuffling the dataset [8], which aims to reduce the number of IO operations between the host and the SCOP. It consists of creating "database slices", each of which has one piece of a record. These slices are then permuted under

the same permutation, and reassembled. This approach can reduce the number of communications between the host and SCOP from $O(N^2)$ to $O(N\sqrt{N})$, and since communications incur a high cost, this considerably improves the time needed by the original $O(N^2)$ algorithm.

Chapter 3

Private Information Retrieval and Update: PIR/W

3.1 Introduction

In this chapter we describe some follow-up work from the private-directory system in the last chapter. Here, we have broadened the scope of possible applications of Private Information Retrieval (PIR), and want to consider how the initial prototype from the last chapter can be extended to other uses.

This chapter is based on our earlier publication in the *3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, 2004 [47].

Examples of where PIR can be useful abound, usually where privacy through data encryption does not suffice, because traffic analysis of the encrypted data can yield useful information. A medical doctor retrieving *medical records* (even if encrypted) from a database may reveal that the owner of the record has a disease in which the doctor specializes. A company retrieving a patent from a *patent database* may reveal that they are pursuing a similar idea. A person or company querying the registration status of Internet domain names indicates that they may be interested in those names. An unscrupulous insider could use this information to buy the names before the legitimate user does, and then demand a higher payment to release them. Clients of all these databases would

benefit from the ability to retrieve their data without the database being able to know what they are interested in.

In the previous chapter we described our initial *Practical Private Information Retrieval* (PPIR) prototype running on the IBM 4758 secure coprocessor with Linux, and offering an LDAP interface to the outside. Here, we start by elaborating on the model of secure coprocessor (SCOP)-assisted PIR.

3.1.1 PIR using secure coprocessors

As introduced in the previous chapter, PIR seeks to allow a client to retrieve an element of a dataset held at a server, such that the server cannot learn the identity of the element.

The model which we follow is that we have available a physically protected computing space at the server, which can give assurance that logical or physical attacks which try to view or tamper it would fail. If this space was large enough to hold the whole dataset, the problem would be solved, as clients could negotiate a secure session with it, and then retrieve their data. Since it is physically protected, no one should be able to observe what item the client obtained. Unfortunately practical considerations result in real protected environments being quite small, much too small to hold the entire dataset.

Thus, the problem becomes that we want to provide private access to a large dataset while using only a small amount of protected space. This is almost isomorphic to the Oblivious RAM problem [41], which we discuss further in Section 3.2.

Model In Figure 3.1 we show the more concrete setup: we have a dataset of N items each of size M. The items may be visible to the host; they may also be encrypted (for the SCOP's private key), though why and how they may be encrypted ahead of time is orthogonal to our topic here. A client connects to the SCOP (tunneling via the host) and delivers a request for one of the items. The SCOP is very limited in memory—it is allowed $\Omega(M + \lg N)$ bits memory, which is the minimum needed to store pointers into the dataset, as well as a constant number of actual data items. Any larger storage, like the actual dataset or pre-processed versions of it, is provided by the host. Thus

the SCOP has to make I/O requests to the host in order to service a client request. To be a correct PIR scheme, it must be the case that the host cannot learn anything[1] about client requests from observing the I/O from the SCOP.

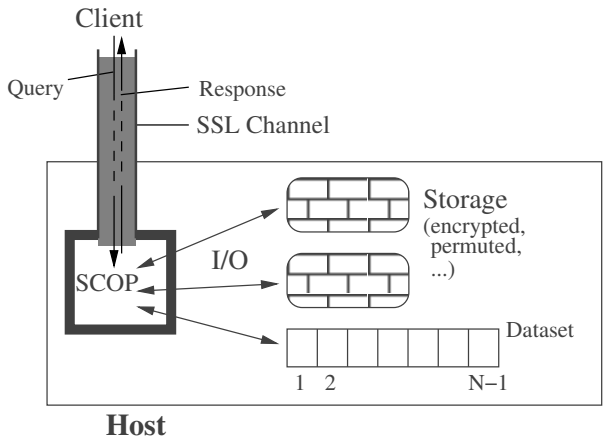


Figure 3.1: The setup of hardware assisted PIR

Simply encrypting the records does not solve the problem; the server can still learn the *identity* of requested items, and (if the server colludes with a user) can learn what any given record decrypts to. It is also insufficient to only hide the identity of *single* retrievals, as then an attacker could learn the popularity of individual items, and correspondence between requests, eg. "Agnes and Boris both retrieved the same data item today".

Latest PIR algorithm

The structure of the algorithm we use was originally developed by Goldreich and Ostrovsky for the Oblivious RAM problem [41]. We note first that it relies on having a dataset of contiguously numbered items, from 1 to N. It proceeds in retrieval *sessions*, where a session S consists of:

Randomly permuting the contents of records 1 **through** N. First, the SCOP encrypts each record in the dataset. Then, the SCOP (pseudo)randomly selects a permutation π on the index range $\{1..N\}$, which we also write as $[N]$. Finally, the SCOP relocates the contents of each record r, $1 \leq$

[1] We are assuming that cryptography works; this scheme is not secure in the information-theoretic sense, since the host can still see ciphertext. Security relies on the adversary being unable to break the underlying primitives like symmetric ciphers.

$r \leq N$, to location $\pi(r)$, changing the encryption along the way. This produces the permuted dataset of encrypted items D_π. The relocations must be done so that the host cannot learn which permuted record corresponds to which input record, after having observed the pattern of record accesses during the permutation. Using the terminology of Goldreich et al., the permutation algorithm must be *oblivious*: have the same I/O access pattern regardless of the input (ie. the permutation)[2]. [41].

Servicing $k \ll N$ retrievals. By now, the permuted dataset D_π is available on the host, and the SCOP knows π. The SCOP uses this knowledge to hide the identities of retrieved records. In order to retrieve record r, the SCOP reads in $\pi(r)$ from D_π, and the host does not learn what r can be.

What is left is to hide the relationships between retrieved items, so the host (for example) cannot tell how many times a given item was retrieved. The approach is to copy records which have been accessed into a *working pool* P_S of maximum size k, which is scanned in its entirety for every retrieval. On each retrieval for record r, one record from D_π is added to P_S: either r if it is not already there, or a random untouched record if it is. Thus, records in D_π are accessed at most once.

The implementer can set a a maximum value of k, to put a maximum value on the response time for any given query. However, the permuting step needs to be fast enough to have a new permuted dataset ready when P_S reaches that maximum k.

3.1.2 Using oblivious networks for private operations on data

The private permutation implementation has varied in the literature, and in our previous prototype we had added a new approach: using Beneš *permutation networks* [89]. A Beneš network can perform any permutation π of N input items by passing them through $\mathrm{O}(N \lg N)$ crossbar switches which operate on two items, either crossing them or passing them straight. The connections between the switches are fixed for a given N, only the cross-bar settings differ for different π.

This network is useful for our problem because (1) the SCOP can use cryptography to simulate a cross-bar switch on two items resident on the host without the host learning which way the switch went, and (2) by doing this for all the switches in a Beneš network, the SCOP can permute the

[2]The access pattern, ie. the sequence and values of I/O operations, will not be identical for all π, but must look identical to a computationally bound observer.

whole dataset without the host learning anything about the permutation, even though he observes all the record I/O. More specifically, to execute a switch the SCOP reads in the two records involved, internally crosses them or not, and writes them out encrypted under a new key so the host cannot tell if it was a cross or not. Since the network consists of $2 \lg N$ columns of switches with $N/2$ switches each, and the SCOP can execute the switches column by column, he can use one key per column, thus never needing to store more than two keys at a time during the operation.

Networks similar to the Beneš are capable of performing other tasks obliviously, again making use of the fact that the SCOP can hide which way a unit operation (on two inputs) went, and by virtue of the fixed structure of the network, the ability to hide the setting of each unit extends to being able to hide the setting of the whole network. We later make use of *sorting* and *merging* networks in this manner.

3.1.3 Improvements to the prototype

There are two areas where we saw the potential to improve our prototype: memory usage inside the SCOP, and the ability to update items privately.

Memory usage

Our prototype used two techniques which required $\mathrm{O}(N \lg N)$ bits of storage inside the SCOP[3]. One was the storage of a permutation π selected uniformly at random from the set of all $N!$ permutations. The other was the execution of a Beneš network on the data items—computing the switch settings of the network required $\mathrm{O}(N \lg N)$ bits[4].

These "memory-hungry" techniques were not a problem in practice for the kind of datasets we were treating, with $N < 2^{13}$ or so, and the memory available in the 4758. However even for $N = 2^{18}$, two objects of $N \lg N$ bits each would need more than 1MB, which begins to strain the 4758's memory. In any case, the memory requirements were, strictly speaking, inconsistent with the desire to have a small protected space.

[3]Note that this is less than the $\mathrm{O}(NM)$ storage which would be needed to hold the whole database: the size of data items we were working with was at least 1KB.

[4]It is not useful to store the settings on the host, as they are computed in an order dependent on the input, so an adversary could learn about π by observing this order.

Updates

Our prototype was really a Private Information *Retrieval* server, and did not have the option for clients to update the contents of data items. This ability could be of interest though, in more interactive applications of the PIR technique, for example if one wanted to build a private filesystem, which could be housed in a remote location but assure a user that nothing about his activities on the filesystem could be gleaned by the remote site.

A database of medical records could also benefit from a private updates feature—if the database is updated without privacy protection, the server operator could learn that, for example, an AIDS specialist is updating Boris's record, and infer that Boris may be at risk of AIDS. If private update techniques were available, doctors could update patient records without any possibility of information leaking to the database operators.

In general, such strong privacy provisions could allow more flexible arrangements for storage of data, with assurance that only allowed information flows will take place, even in the presence of traffic analysis by a strong adversary.

3.2 Related work

Throughout this chapter one notices references to Oblivious RAM (ORAM) [41]. This is because that problem has a very similar structure to hardware-assisted PIR, and the mechanisms developed there are for the most applicable here too. The ORAM problem is for a physically shielded but space-limited CPU to execute an (encrypted) program such that untrusted external RAM cannot learn anything about the program by observing the memory access pattern. The CPU corresponds to the SCOP (acting on behalf of clients), and untrusted RAM corresponds to the host. The asymptotically slower solution presented there (square-root algorithm) is what we base our algorithm on.

The asymptotically superior ORAM solution (polylog algorithm), has a $\mathrm{O}(\lg^4 N)$ per memory access overhead. An actual operation count reveals that it has a larger actual overhead that the square-root algorithm for about $N < 2^{20}$. Such large dataset sizes are practically infeasible for both algorithms on the hardware we used (the IBM 4758), so we have not experimented with the more

complicated polylog algorithm. The new extension of the ORAM polylog algorithm by Williams and Sion [91], which we examine in Section 5.1.2 on page 81, may provide a much more efficient alternative here.

The ORAM work has covered some of the aims we address in this chapter, namely private reading and writing of memory words of size $\lg N$ bits, using a protected CPU with $\Theta(\lg N)$ bits memory size (in this case the "record" size which we call M is the word size, $\lg N$ bits).

The new contributions over ORAM in this chapter are:

- an asymptotically and practically more efficient method of re-permuting the dataset between sessions (Section 3.3.2),
- a practically efficient session-transition scheme (Section 3.4.2),
- permutation using the Luby-Rackoff scheme (which has advantages, for example enabling us to compose and invert pseudo-random permutations) (Section 3.3.1),
- an actual implementation on commodity secure hardware.

Ostrovsky and Shoup introduced communication-efficient *private information storage*, the computationally secure version of which is based on the Oblivious RAM algorithm [69]. Their scheme for computationally-secure private information storage is to simulate an ORAM secure CPU–insecure RAM protocol by running the CPU algorithm on two non-colluding servers, and the RAM on a third server which must not collude with both CPU servers (but can with one of them). The two CPU servers use a secure two-party protocol to carry out the CPU algorithm such that neither of them in separation knows the secret material (key to a pseudo-random function) which can reveal the request pattern coming from the user.

This scheme would incur considerable costs on account of the secure two-party computation used by the CPU servers, and is not practically competitive with our TTP-based approach.

3.3 Solution for high memory usage

In this section we present solutions to the high memory needs of the previous prototype. As mentioned before, we had two distinct sources of super-logarithmic memory usage, both of which are addressed.

3.3.1 Permutation

We need a way to generate, store and compute a permutation π on the set of integers $\{1, \ldots, N\}$, abbreviated to $[N]$.

Functional requirements π should be storable in $\mathrm{O}(\lg N)$ space, which rules out the use of a truly random permutation: that would require $\mathrm{O}(N \lg N)$ bits of storage.

It should also be efficiently invertible, which is required by our re-permuting algorithm (Section 3.3.2). This means the SCOP should be able to efficiently compute $\pi^{-1}(i)$ as well as $\pi(i)$.

Security requirements π should be difficult to guess, for an adversary of our PIR/W protocols. In particular, the adversary should not be able to guess images of π for inputs which he has not already observed. He can observe some images, as he may be posing as a client, and query the system with known inputs i. Then he can observe the accesses that the SCOP makes to the host structures, and thus learn the image $\pi(i)$.

Such an attack: observing the mapping of the secret function on inputs of the adversary's choosing, is called a *chosen-plaintext attack* (CPA). One important parameter of a CPA is the number of input-output pairs the adversary can observe. In our case, it could be a maximum of k pairs, where k is the number of accesses in a session.

The requirements on π point to the use of a *pseudorandom* permutation, and the one we chose is the Luby-Rackoff-style cipher (L-R cipher) on $\lg N$-bit blocks, with 7 rounds (LR_n^7) [57].

An L-R cipher on $2n$-bit blocks is a *Feistel network* with independent pseudo-random *round functions*. A Feistel network (illustrated in Figure 3.2) consists of r iterated rounds, each of which

computes values $L, R \in \{0,1\}^n$:

- L_0 and R_0 are respectively the first and second halves of the plaintext x,
- $(L_i, R_i) = (R_{i-1}, L_{i-1} \oplus g_i(R_{i-1}))$,
- g_i are the *round functions*, $f_i \in \{0,1\}^n \to \{0,1\}^n$, which just have to be pseudo-random and independent from each other. Note that they do not have to be permutations for the whole network to be a permutation—this is part of the point of a Feistel network, that non-invertible functions are used to produce a permutation.
- $\oplus$ is the bitwise XOR operation,
- The ciphertext is the concatenation of L_r and R_r.

Luby and Rackoff initially proved chosen-plaintext security with 3 rounds, and chosen-ciphertext security with 4 rounds, in both cases with only a limited number of queries against the cipher oracle.

Recent results have improved the security bounds for higher-round L-R ciphers to state that LR_{2n}^7 is indistinguishable from a truly random permutation by a computationally unbounded adversary given m chosen-plaintext queries, where $m \ll 2^{n(1-\varepsilon)}$ [70].

A variation on the basic L-R scheme has been conjectured to give a much higher resistance to CPA—unbalanced Feistel schemes which have round functions $g_i \in \{0,1\}^r \to \{0,1\}^{2n-r}$, where $r \neq n$. Such a scheme is described by Naor and Reingold [66, Sect. 6], among others. We show an illustration in Figure 3.3. Unbalanced L-R schemes have previously been shown to improve block cipher resistance to cryptanalysis [75], and to enable faster ciphers [58]. More recently their improved resistance to CPA has been shown [66, Sect. 6]. Furthermore, Patarin conjectures that an unbalanced L-R scheme on $2n$ bits, using round functions $g_i \in \{0,1\}^{2n-1} \to \{0,1\}$ (ie. boolean functions on $2n-1$ bits), is secure against CPA with m chosen-plaintext queries, where $m \ll 2^{2n(1-\varepsilon)}$ [70].

For the pseudo-random functions inside the cipher, we use TDES (which is hardware accelerated on the 4758) with expansion and compression to give a function on the required domain, as illustrated in Figure 3.4.

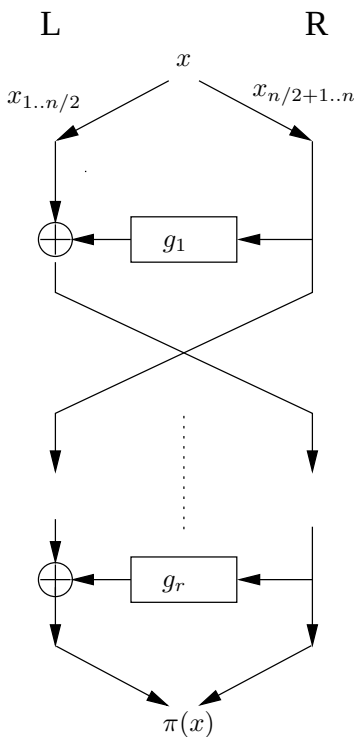


Figure 3.2: A snapshot of an r-round Feistel network for a permutation π, on n-bit strings, with r round functions $g_i : \{0,1\}^{n/2} \rightarrow \{0,1\}^{n/2}$.

Permutation on arbitrary-sized sets

Since Luby-Rackoff style permutations are only defined on sets of a size which is an even power of 2, and we need a pseudorandom permutation for the set of integers $[N] \stackrel{\text{def}}{=} \{1..N\}$, where N may not be a power of 2, we need to be able to adapt a permutation on $2n$-bit integers to arbitrary-sized sets. A technique for this, among other things, is described by Black and Rogaway [17]. The technique is to iterate a base permutation, which is on $2n$-bit integers, for the lowest n s.t. $2^{2n} \geq N$, until a result in range (ie. $\leq N$) is obtained. The expected number of iterations is less than 2 (since 2^{2n} may be only a little larger than N, in which case the base permutation will produce images $\leq N$ with high probability), and the authors prove that this produces a pseudorandom permutation as strong as the base permutation. It is also invertible using the same technique: apply the base inverse permutation π^{-1} until an in-range pre-image is obtained.

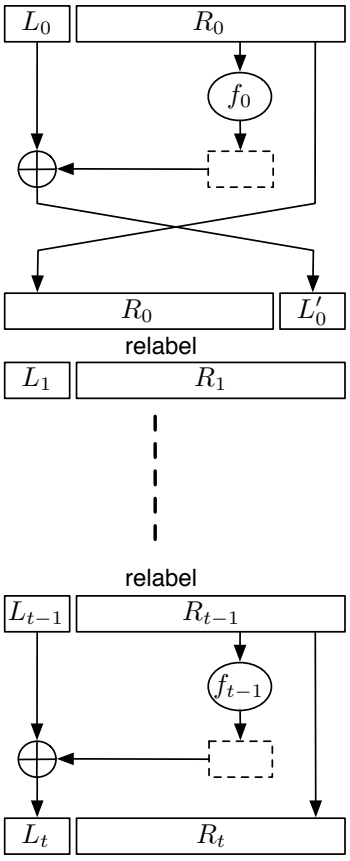


Figure 3.3: An unbalanced t-round Luby-Rackoff network on $2n$ bits, with t round functions $f_i : \{0,1\}^{|R|} \to \{0,1\}^{|L|}$, where $|R| + |L| = 2n$.

3.3.2 Permuting the dataset

Once we have established a random or pseudo-random permutation, we need to actually permute the records such that the server cannot learn anything about the permutation. As mentioned before, the Beneš network is not applicable if we are to use only logarithmic space. The algorithm to set its switches for a given permutation has resisted many attempts to reduce its complexity.

Take 1: sorting networks

The straightforward solution here is to use a sorting network, which is structurally very similar to the Beneš network, to perform the permutation. In particular, the Batcher bitonic sorting network

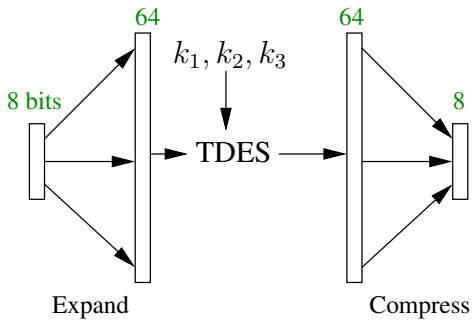


Figure 3.4: Pseudo-random function g on 8 bits, to be used inside a Feistel network operating on 16-bit blocks. The TDES function inside is just one possibility—any good block cipher should be usable there. The expansion step is deterministic and just needs to preserve all the information in the input x—appending 0-bits suffices. The compression is likewise deterministic and should make use of each bit in the ciphertext, for example by XOR-summing the respective 8-bit sub-blocks of the ciphertext.

can sort N items by passing them through a fixed network of $\frac{N \lg^2 N}{4}$ *comparators*, which are 2-input 2-output units that sort the two inputs. It is very structurally similar to the Beneš network, but is self-routing, as the comparators do not need any external information (like the switch settings in the Beneš network) to decide their action [11].

In order to perform a permutation π using a bitonic sorting network, the SCOP would first tag each data element i with its target index $\pi(i)$, and then run the sorting network using the $\pi(i)$ values as keys for comparison of elements.

Unfortunately, while the bitonic sorting network is ideal for our task in terms of its simplicity and low space requirements, it has an appreciably larger number of comparators ($\frac{N \lg^2 N}{4}$) than the Beneš network has switches ($N \lg N$). Thus, performing a permutation using a sorting network would take $1/4 \lg N$ times longer than a Beneš permutation.

Permuting an arbitrary number of elements In its natural form, the bitonic sorting network works on a number of elements N which is a power of 2. However it is trivial to accommodate an arbitrary N—simply ignore any comparators which have at least one input from a wire greater than N.

Re-permuting the dataset

The solution which we came up with takes advantage of the fact that only a small fraction of the dataset is touched during a query session. The untouched items are completely unknown to the adversary, and thus do not need to be re-permuted. Only the touched items have to be re-permuted, to invalidate the information the adversary may have learned about them during the session. Informally, the procedure for re-permuting is as follows.

Let the current permutation be π_1. Let T be the touched items at the end of a session, and $\overline{T}$ be the remaining items, untouched. Let the size of T be k (which is constant in our prototype, so as to limit the query response time). For the next session we generate a new permutation π_2. Also we assume that the indices of the items in T are available in a list L_T in the SCOP. Then we follow the following algorithm:

Re-permuting algorithm

(i). Re-order the items in $\overline{T}$ so they are sorted by $\pi_2(i)$. We do not need to do this obliviously. We just need to hide what are the indices of T under π_2 (but not under π_1—this is already known).

(ii). Obliviously re-order the items in T so they are sorted by $\pi_2(i)$.

(iii). Obliviously merge the re-ordered T and $\overline{T}$, to give a dataset permuted under π_2.

This yields savings both in time and space over using a Beneš network to do a full re-permutation. We will first describe in more detail the algorithms used, and then present the resources needed. We assume that we can compute inverse permutations, which is true with Luby-Rackoff style permutations. **Step (i)** is shown in Algorithm 1. **Step (ii)** can be directly performed using a *sorting network*, eg. one of Batcher's networks. However a more efficient approach is to use the Beneš network, after computing the permutation vector for the reordering needed. This can be accomplished using the list L_T, with one sorting step to obtain a sorted list of the indices in T under π_2[5]. **Step (iii)** can be performed using a merging network.

[5]Note that we had to perform this sorting at the start of Algorithm 1 too, so the output of that can be reused.

A good reference for sorting and merging networks is found in *Introduction to Algorithms* [28, chap. 27], and in Section 3.1.2 we explain how such networks can be used to perform operations on a large dataset obliviously.

Notes Step 6 in Algorithm 1 must take the same amount of time at every execution[6], but this is easy given the sorted array L_{T,π_2}, and takes constant time.

Algorithm 1 Step (i) of the re-permuting algorithm: Reordering the items in $\overline{T}$ from π_1 to π_2

Require: T is the set of touched records, $\overline{T}$ are the remaining records.
Require: D_{π_1}: the whole dataset under π_1, on the host.
Require: L_T: list of the indices of T, in the SCOP.
1: $L_{T,\pi_2} \leftarrow$ indices of T under π_2, sorted ▷ *Using* L_T
2: $\overline{T}_{\pi_2} \leftarrow \emptyset$ ▷ *The destination array (on the host) for the records in* $\overline{T}$
3: $j \leftarrow 0$ ▷ j *is an index under* π_2
4: **while** $j < N$ **do**
5: $j \leftarrow$ next index in $\overline{T}$ in order of π_2 ▷ *guided by* L_{T,π_2}
6: $r \leftarrow \pi_1(\pi_2^{-1}(j))$ ▷ r *is an index under* π_1
7: $R \leftarrow$ `read_from_host` $D_{\pi_1}[r]$
8: Tag R with destination j ▷ *But this tag is hidden from the host*
9: Append encrypted R to $\overline{T}_{\pi_2}$ ▷ *Recall* $\overline{T}_{\pi_2}$ *is on the host*
10: **end while**

Step	Time cost	Space cost (in bits)
(i)	$\mathrm{O}(k \lg k)$ for sorting, $\mathrm{O}(N-k)$ for the loop	$\mathrm{O}(k \lg N)$ for the indices of T
(ii)	$\mathrm{O}(k \lg k)$ for the Beneš network	$\mathrm{O}(k \lg N)$ for building and storing the permutation vector, $\mathrm{O}(k \lg k)$ for the Beneš network
(iii)	$\mathrm{O}(N \lg N)$ for the merging network	$\mathrm{O}(\lg N)$ for indices

Table 3.1: Cost of the re-permuting algorithm. k is the number of queries in a session, same as the size of a touched set. Note that the cost of the merging network in the last step is the dominant one, and that is half the cost of a Beneš network on the same input size. Also the storage needed is $\mathrm{O}(k \lg N)$, which is $\mathrm{O}(\lg N)$ for constant k (which is how we set k). Even if $k = \sqrt{N}$ as in the ORAM square-root algorithm, the storage required is considerably sub linear.

The initial permutation

For the initial permutation, which has to re-order all the items obliviously, we resort to the use of Batcher's bitonic sorting network. This method was used in the ORAM work for all permutations

[6] Or an adversary could use timing attacks to deduce information about the indices of $\overline{T}$ under π_2.

of memory. This takes $\mathrm{O}(Nlg^2N)$ time rather than the $\mathrm{O}(N \lg N)$ for the re-permutation algorithm above, but only needs to be done once when the server starts up.

Correctness and security

Since our initial paper with the material in this chapter [47], Wang et al. have published a similar PIR scheme with some improvements [90], and provided a proof of security, which applies for the re-permutation algorithm we present here as well.

3.4 Solution for updates

The problem of evolving our previous design to support private updates of data items reduced to two main tasks: ensuring integrity of data, even against replay attacks[7]; and dealing with the fact that incoming updates render the data in any long-lived preprocessing steps stale: for example a permuted dataset will be out-of-date by the time the permutation is done (assuming that permutations run in parallel to queries, which is necessary to avoid downtime between sessions).

The easy part was modifying the retrieval session to deal with (1) hiding whether a client request is an update or a retrieval, and (2) hiding which item in the working pool is being updated. The approach is to update *all* records in the working pool with every request. Note that this is a similar workload as we already have for reading—each record in the working pool must be read for each read request, but the working pool size is much smaller than N (eg. $\sqrt{N}$). In particular, for every record r in the pool, the SCOP writes either $\{r\}_K$, or $\{r_{new}\}_K$ if a new value r_{new} is provided by the client. The variable K is a new key generated for this encryption of the working pool only. Given this change of key, the host cannot tell if and where a new record was written. Note that the SCOP does not need to keep the keys for previous versions of the pool.

[7] Replay attacks are where the adversary replaces an item with another one which has a correct checksum/MAC, but comes from a previous execution of the algorithm.

3.4.1 Integrity

The integrity of any object stored on the host is assured by first tagging it with a value t which specifies both the *physical* and *temporal*[8] location of the object, and then applying a keyed message authentication code (MAC) to the object and the tag. The location code and MAC are stored with the object on the host. For example, during the last step of a re-permute operation (the merging network) we have $t = \langle s, d, i \rangle$, where s is the current permutation number, d is the depth within the network[9] (both temporal), and i is the item's current actual location in the dataset (physical). Thus, an adversary obviously cannot modify the item's contents undetected, but it also cannot substitute an item from an earlier time (ie. cannot perform a replay attack).

Of more interest is how to compute the temporal location of an object updated during a query session. Within the s^{th} query session, at the end of the i^{th} client request, the query SCOP has built up a working pool of touched records $P_s = \langle r_1, r_1, \ldots, r_i \rangle$. The temporal tag for each record in P_s would then be $t = \langle s, i \rangle$. The notable aspect here is that the SCOP can compute the temporal tag for each object which needs it while maintaining only a fixed small amount of state—s and i in this case. This temporal tagging with small state is the same notion as the "time-labeled" property expounded for some of the Oblivious RAM simulations, and also used to protect against tampering and replay attacks [41].

3.4.2 Session continuity

The problem of transitioning between query sessions is trivial in the case of read-only PIR: since the database contents are assumed static, several permuted copies can be produced in advance and used immediately whenever needed—the permuted dataset does not go stale. If updates are supported though, pre-permuting is not an option as the permuted datasets *will* be stale soon. Even worse, updates will occur between the start and end of a permutation, requiring them to be incorporated into the output of that permutation before it can be used. Here we describe our scheme for transitioning between sessions.

[8] "Temporal" in the sense of where in the timeline of the algorithm the object is located.

[9] The merging network has $\frac{1}{2} \lg N$ levels of $N/2$ independent comparators each, and the depth is the current level number during an execution of the network.

Given that we run a permutation concurrently with a query session, the output of the permutation will not contain the updates received during that session. We deal with this problem by incorporating the records T_i touched during session i into the working pool of session $i+1$ from the beginning. This means that session $i+1$ will touch each record in T_i at every operation, in addition to its own accumulating T_{i+1}. At the end of session $i+1$, its working pool will contain both T_i and T_{i+1}.

Overview of the algorithm

In Figure 3.5 we show a diagrammatic representation of the actions of all the components during one full session.

We first note that the working pool of the query session consists of two parts—the set of records touched during that session (which is empty at first), and the ones touched during the last session.

At the end of session $i-1$, the query SCOP has produced a new touched set T_{i-1}, and the permuter has produced a new permuted dataset D_{π_i}. The new session i starts by writing the items in T_{i-1} into D_{π_i} (directly, one by one), and also adding them to its working pool.

The permuter begins to re-permute the dataset D_{π_i}, with touched set T_{i-1}, for use in session $i+1$ (recall from Section 3.3.2 that we only need to obliviously reorder the items in a permuted dataset which have been touched since the last permutation—these items are now T_{i-1}).

3.5 Our PIR/W implementation

Here we outline the concrete algorithm we use in this prototype for reads and updates within an access session. It is slightly generalized from the case where a SCOP provides access to a single dataset—here, the SCOP may control multiple datasets.

As before (see Section 2.4.2), the SCOP's host provides a simple interface to create, read and write contiguously indexed *containers*, each consisting of a fixed number of fixed-size elements.

Each dataset D is stored in two containers on the host:

- A_D—the N items in the dataset at the beginning of the current access session.

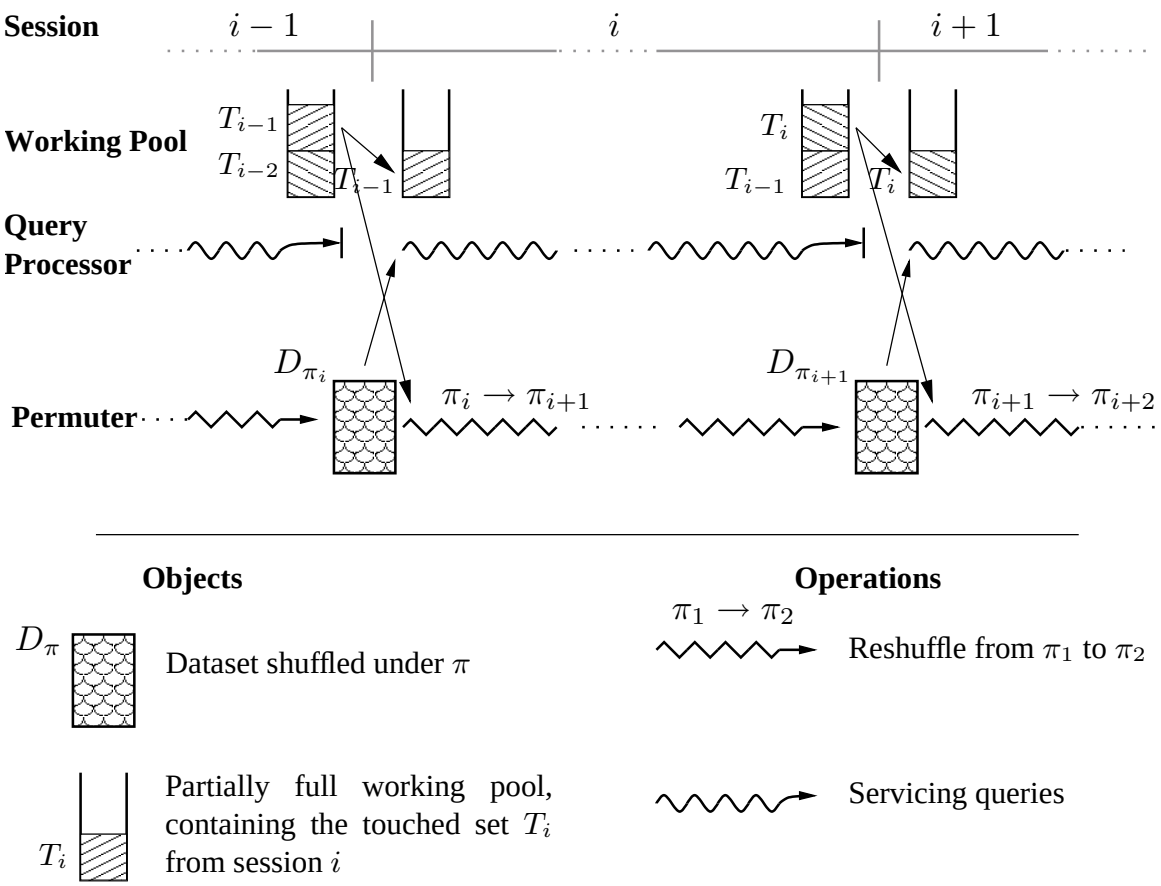


Figure 3.5: A snapshot of the overall algorithm across one session. At the end of session $i-1$ the permuter has prepared D_{π_i}, and the query SCOP has a working pool containing the touched sets T_{i-1} and T_{i-2} (see Section 3.3.2). Then, the permuter begins a re-permutation with permutation π_{i+1} and with touched set T_{i-1}. The query SCOP begins a new session i with T_{i-1} in its working pool, and filling in its own T_i.

- T_D—k entries representing the working area of this access session. Each entry contains a whole item, and its physical (unpermuted) index.

All elements in the containers are encrypted and MACed. All updates within a session are done to T_D, while A_D stays static.

To describe how reads and writes are implemented, we first describe several subroutines.

`append-to-T (`D`,` i_π`)` An item and its index from A_D are appended to T_D, preserving the invariant that all items in T_D are unique. The index will be i_π, unless that is already present in T_D,

in which case the index must be a random index $r \in [N_D]$ (ie. an r in D's index range) not already in T_D. This step takes $\mathrm{O}(|T_D|)$ time, to check whether i_π is in T_D, and to find a suitable r if needed.

`scan-T (`D`, `i_π`, should-write, new-val)` All the elements of T_D are processed, one by one:

1. read in $\rightarrow X$,
2. if index of X is i_π and `should-write` is set, update value of X with `new-val`,
3. write X out re-encrypted.

The element with index i_π (which, as a precondition, has to be in T_D) is kept internally and returned at the end of the scan. If this scan is part of a write operation, the result is not needed, but it is used for a read.

Now we are ready for the brief definitions of reading from and writing to a dataset:

read A read on index i of dataset D is done by the SCOP as follows:

READ(D, i)
1 $i_\pi \leftarrow \pi(i)$
2 APPEND-TO-T(D, i_π)
3 $X \leftarrow$ SCAN-T$(D, i_\pi,$ FALSE, NIL$)$
4 **return** X

write A write of value X into index i of dataset D is very similar:

WRITE(D, i, X)
1 $i_\pi \leftarrow \pi(i)$
2 APPEND-TO-T(D, i_π)
3 SCAN-T$(D, i_\pi,$ TRUE, $X)$

3.5.1 Cost Analysis

Define $\mathrm{PIRO}(N)$ to be the amortized cost (or overhead) of a read or write to a private database of N items. The total cost consists of the online cost of scanning the working area, and the (potentially offline) periodic cost of permuting and re-permuting the database. In terms of k, the size of the working area:

- The online cost for each retrieval is $\mathrm{O}(k)$, and
- The periodic re-permutation cost of $\mathrm{O}(N \log N)$ is incurred every k accesses.

Thus, the amortized cost per access is

$$\mathrm{PIRO}_k(N) = \mathrm{O}(k + \tfrac{N \log N}{k}) \tag{3.1}$$

We have set k in different ways depending on the setting:

- For an interactive PIR/W service, we set k it to a constant value, to limit the latency of the service, which determines how long users have to wait for an answer.
- For non-interactive usage of the algorithm, like in the Faerieplay system presented from Chapter 7 and on, we set k to minimize the amortized cost. This setting is $k = \Theta(\log N\sqrt{N})$, which yields an amortized cost of

$$\mathrm{PIRO}(N) = \mathrm{O}(\log N\sqrt{N}) \tag{3.2}$$

3.6 Experimental results

Here we present some performance results from our prototype. We had configured the session size k to be 128. As in our previous PIR prototype, a constant session size gives an upper bound on the online latency of query processing.

In Figure 3.6 is shown the running time for the re-permutation operation described in Section 3.3.2, for different database sizes N. In Figure 3.7 we show how long it takes the query SCOP

to process queries. Note that these times are independent of N, as long as permuted datasets are available when needed.

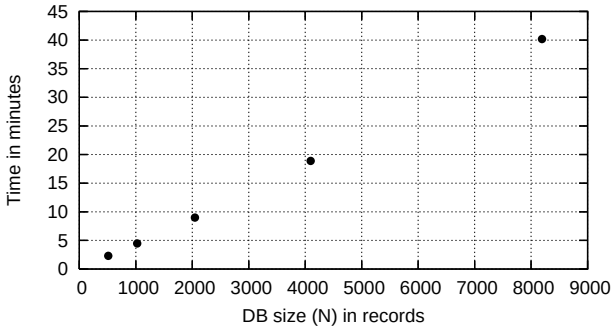


Figure 3.6: Duration of re-permutation, for varying dataset sizes. The record size was 850 bytes in all cases. The dominant operation is the oblivious merge, all the others take much less time.

Putting these two measurements together gives an idea of what kind of service this prototype can offer. In Table 3.2 we show the query processing time possible for different N, with two limiting factors: the query processor speed, and the re-permutation speed (keeping in mind that the query processor can only service $k = 128$ queries before needing a new permuted dataset from the permuter).

N	Query processor limit	Permuter limit	Response Time
1024	5.5	2.0	5.5
2048	5.5	4.1	5.5
4096	5.5	8.7	8.7
8192	5.5	18.5	18.5

Table 3.2: Query response times (in seconds) attainable with different sizes of datasets. The two limit columns show how the respective operations limit the response—the query processor with its average latency (from Figure 3.7), and the permuter by virtue of having to complete a whole re-permute before the next session can begin. In the $N \geq 4096$ cases, the query SCOP could handle more hits, but a single permuter is not producing permuted datasets quickly enough. An easy way out here is to do the merge step of the re-permutation in parallel, using two or more SCOPs, and gaining linear speedup with the number of SCOPs, as the merging network is actually intended for parallel use. For $N = 1024$, the query SCOP can be always busy and the permuter will keep up.

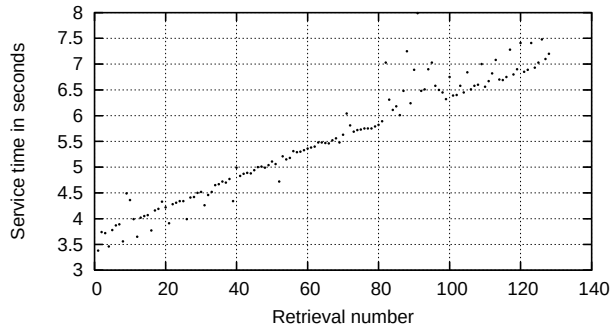


Figure 3.7: How long does the query SCOP take to service a request? Here are the times during one query session. The record size was 850 bytes, and the database size N does not affect the query times. Recall that because of the session-continuity algorithm (Section 3.4.2) the working pool starts with all k items touched during the previous session, in this case 128. The pool accumulates k more during the session.

3.7 Conclusion

We have presented the evolution of our previous work on a hardware-assisted private information retrieval prototype—improved performance in terms of both running time and space, and the ability to update items privately. The prototype gives reasonable performance on dataset sizes up to about $10,000$, and can benefit easily from parallelism via extra hardware units.

Chapter 4

Setting the Stage: Building Blocks

Having presented our preliminary projects in the last two chapters, in this chapter we summarize some previous works which we use extensively in the rest of the thesis. Many of these we have already touched on: secure coprocessors, oblivious RAM. The application of the remaining topics introduced here will become apparent in the subsequent chapters.

4.1 Trusted computing base

The notion of a *trusted computing base* (TCB) is important in most security-sensitive endeavors, as it frames the set of computational entities which do not need to be explicitly secured, but are assumed to be secure.

The classical treatment of the TCB comes from the Orange Book [31]. A TCB is defined as the total set of components in a computer system, which are required to enforce the system's security policy; any components not in the TCB should not have to be trusted to cooperate in maintaining security, but should instead be seen to be under the adversary's control.

By definition, no provisions are made in a protocol or security subsystem to verify or enforce the trust properties of a TCB. Usually some external considerations apply to warrant the trust, eg. the reputation of a hardware manufacturer, or "many eyes" scrutiny of open-source software. Additionally, steps may be taken outside of a protocol to ensure the trustworthiness of the protocol's

TCB. For example, the software implementing the protocol may be in the TCB, but it has been model-checked to verify the required trust properties, and its integrity is tracked subsequently.

For example, if Bob uses a web browser to generate a secure connection to a remote service, he is assuming that his browser is carrying out the SSL protocol as specified, and is not for example sending the negotiated secret key to some third party, saving it for later use in decrypting the communication stream, or using a weak algorithm for generating any random strings needed by the protocol. Thus the browser is part of Bob's TCB.

In general, a TCB exists any time a protocol is actually implemented and used on a computer, whether the TCB is explicitly acknowledged and managed, or just exists under the covers.

It is a well-accepted principle that a system's TCB should be as small as possible. For example, if Bob's TCB in SSL transactions includes his whole browser (not to mention operating system), it may be difficult to provide assurance of that TCB's trustworthiness. If the TCB can be shrunk to just the SSL module in the browser, for example by separating it from the other modules like the Javascript interpreter, then a stronger argument could be made that that TCB is actually trustworthy. The Javascript interpreter can do whatever it (or rather the adversary) wants, but it should not be able to violate the security properties promised by SSL.

For multiparty settings, there is a distinction between self and others. A TCB is necessarily defined from a particular participant's perspective, and each participant's TCB is likely to be different. The notion of a single system-wide TCB does not make sense. Works on SMC usually postulate that others (second and third parties) are not in a user's TCB, while the totality of the user's hardware and software is.

4.2 Secure coprocessors

An important reason to pursue a direction of work or study is "because we can". The practical aspects of this thesis have been enabled by the recent development, building, and commercial deployment of devices which can serve as TTPs in multi-party computations, in the form of secure coprocessors.

A secure coprocessor is a small general purpose computer armored to be secure against physical attack, such that code running on it has some assurance of running unmolested and unobserved [95]. It also includes mechanisms, called *outbound authentication* (OA)[1] to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [77]. The coprocessor is attached to a *host* computer. The SCOP is assumed to be trusted by clients (by virtue of all the above provisions), but the host is not trusted (not even its root user). The strongest adversary against the schemes presented here is the superuser on the host, who may also be equipped with a drill.

We elaborate on the adversary, his abilities and how he can be modeled in Chapter 8.

4.2.1 IBM 4758 secure coprocessor

The 4758, on display in Figure 4.1, is a commercially available device, validated to the highest level of software and physical security scrutiny then offered—FIPS 140-1 level 4 [82][2]. It connects to its host via PCI. It has an Intel 486 processor at 99 MHz, 4MB of RAM and 4MB of FLASH memory. It also has cryptographic acceleration hardware, and notably a "fast path" DES and TDES mode of operation, where data can be streamed from the host through the device's DES engine without touching the device's internal RAM. The maximum throughput of the TDES engine is about 20MB/s, which is certainly much faster than contemporary (ie. about 1998) CPUs could manage, but is slower than what a current (2007) fast CPU and RAM can do. For example, on a Dell server with an AMD Opteron 280 2.4GHz CPU, released in October 2005, the `openssl speed` command reports speeds of about 21.5 MB/s for TDES, and 46.5 MB/s for 128 bit AES.

In production, the 4758 runs IBM's CP/Q++ embedded OS; however, IBM also provides an experimental configuration with Linux (the follow-on product from IBM, the PCIXCC [6] runs Linux normally). Linux has considerable advantages over CP/Q++ in terms of code portability and ease of development, so we have used only the Linux configuration for our implementations.

[1] OA is more recently referred to as *remote attestation* in the context of the Trusted Computing Group.

[2] FIPS 140-1 has been superseded by 140-2 since 2002, but the new standard does not provide any higher assurance levels [83].

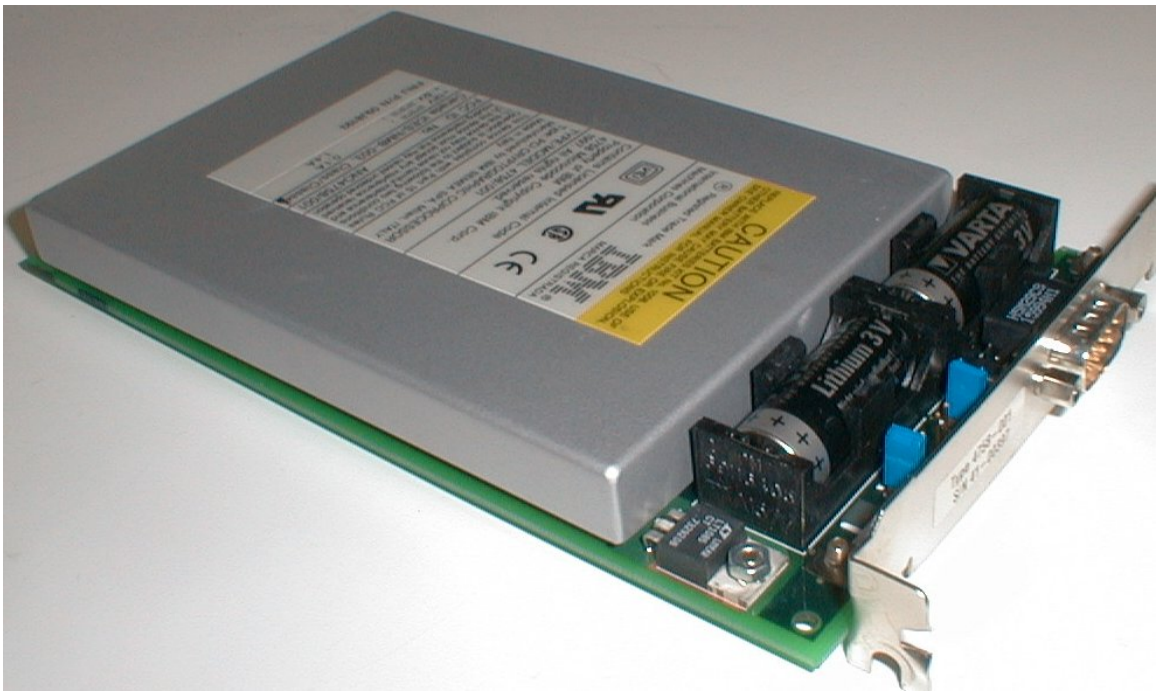


Figure 4.1: An IBM 4758 secure coprocessor. The PCI connector is on the right hand side, not visible here.

Logical security and trust

The 4758 presents a general purpose computing environment, but it is geared towards providing high assurance for a *single* application, rather than supporting convenient multitasking. Thus, the software configuration at any one time consists of a single application. In terms of trust assurances provided by the 4758, the operating system and the user application can be considered as one: the 4758 assures that they will run as programmed, with secure access to their private key material. The device cannot assure that the application and OS are themselves correct—they could leak secrets or fail to preserve data integrity.

Outbound authentication

As introduced above, Outbound Authentication (OA) is the means by which a secure coprocessor can prove to relying parties that it is an untampered device running a known instance of a program, and also allow a user to encrypt data such that it is only accessible to a particular SCOP (in a known

software configuration).

The 4758 Outbound Authentication mechanism is based on a chain of key-pairs and certificates, starting with the device key-pair which is generated and signed at the factory. The mechanism allows software on the 4758 to prove its identity (program hash), the identity of all system software, and the device attributes to remote parties. The basic capability that OA provides is for the device to sign data with a key which is certified by the certificate chain leading up to the (fixed and trusted) IBM 4758 foundry key. A recipient of such a signature can conclude, for example, that the signature was generated by "version X.Y.Z of Acme policy enforcer, running on CP/Q++ on a genuine and untampered 4758". A protocol may use such a signature in different ways:

- While a user and a 4758 application are setting up a secure channel, the 4758 will send the user a signed session key, or its part of a session key. If this is an OA signature with a supporting certificate chain, the user has assurance about what software and hardware entity he is talking to over the secure channel.
- The signature could be on some piece of data which is the output of a computation done by the 4758, in which case the authenticity of the data can be proved at a later time as well.
- It could be a signature on a public encryption key held in the SCOP, which allows Agnes to encrypt data with that key knowing that the ciphertext can only be accessed by that SCOP, running the software indicated in the signature and chain. .

The 4758's OA mechanism also allows upgrading application and system software, as well as part of the device's firmware. Such upgrades are constrained to delete secrets/keys if the upgrade may result in uncertainty over what software versions have had access to the keys.

The direct software identity authentication deployed in the 4758 could in theory be expanded along the lines developed recently for consumer-level trusted hardware like the Trusted Platform Module (TPM) [86]—eg. property-based attestation [73], and direct anonymous attestation [21], but this is orthogonal to our work here.

Physical security

The 4758's casing includes a tamper-detecting mesh. If it detects an attempt to open the casing, it triggers zeroization of the DRAM (computation state), and battery-backed RAM (long-lived secret keys). Thus, a physically present adversary cannot open the device and learn anything useful about its computation. Additionally, the device will preemptively zeroize itself if it detects too low temperature or excessive radiation, both of which may prevent effective and fast zeroization of the device's RAM.

4.3 Oblivious RAM

Motivated by the problem of protecting software from copying and unauthorized use, Goldreich and Ostrovsky developed techniques for a trusted CPU to execute a program using an untrusted RAM, such that an adversary controlling the RAM cannot learn anything about the program [41].

The major challenge in foiling such an adversary is to hide the access pattern to the RAM, ie. the sequence of addresses issued to the RAM. This includes hiding the actual addresses, as well as relationships among them.

More concretely, the access pattern that an adversary controlling the RAM observes should look the same to him, regardless of the program that the CPU is executing, and its inputs. We will develop the security model for this setting in Chapters 6 and 8.

Two main algorithms emerge from the ORAM work—the square root algorithm and the polylog algorithm, named by their amortized overhead. Namely, in order to carry out one access to a RAM of N words, the square root algorithm takes $\mathrm{O}(\sqrt{N} \lg N)$ time, and the polylog algorithm takes $\mathrm{O}(\lg^4 N)$. Both times are amortized—the algorithms periodically engage in long pre-processing operations. Also, the big O is much bigger for the polylog algorithm—concrete operation counting shows it overtaking the square root algorithm only at about $N = 2^{20}$.

We have described the square root algorithm in Section 3.1.1, as we used it in our preliminary private information retrieval (PIR) projects. We do not discuss the polylog algorithm any further, as it is more expensive for the dataset sizes which are feasible to run on the 4758 as a TTP, and is

also more complicated than the square root algorithm (the polylog algorithm can be considered a generalization of the square root algorithm).

4.4 Secure multiparty computation

Secure (multiparty) computation (SMC) aims to enable multiple parties to engage in a joint computation on their private or proprietary data. They would like to learn something which is a function of all their data, but none of them want to reveal their actual data to any of the others.

For example, a group could want to learn which of the members has the highest income, but no member wants to disclose their actual income to any of the others. We introduced this *millionaires' problem* in Section 1.5 on page 10. It is the "hello world" example of SMC.

We expand on SMC in Chapter 6.

4.5 Trusted third parties

4.5.1 Parties?

In the setting of distributed computing, and more generally in collaborative ventures, the notion of various "parties" is used. One entity (person or organization or more generally a user) provides the vantage point for any discussion, and is called a first party. In a computational setting, the first party will have some TCB, which will behave in accordance with their interests. (This may be a simplistic assumption, as we elaborate in Section 6.3.2.) This TCB could be the user's PC, or in the case where the user is an organization, it could be multiple hosts within the organization's network.

The other entities directly involved in the collaboration, as specified by the functionality, are called second parties. They can also be seen as the first party's peers.

Finally, there can be other entities involved in implementing the functionality, and they are called third parties. A (computational) third party could perform many tasks in a computation—provide the current time, provide a common random string for the participants, maintain an audit of the computation, etc.

We note that one does not usually hear much about first or second parties, as they are always part of a multiparty computation, and do not need to be named. We are pointing out this classification mainly to frame the notion of a *third party*.

4.5.2 Trusted

Usually, using a computational third party has security implications. For the examples above, the respective TTP should provide the correct time, ensure absence of coercion in the generation of random strings, and ensure integrity of an audit log. Thus, in some way, most third parties are actually *trusted third parties* (TTPs)—they are trusted to perform some task correctly, and if they do not, the multiparty computation could diverge from its requirements.

4.6 Introduction to adversary models

In analyzing secure protocols, one of the first steps is to specify what kind of adversary the protocol should defend against.

There are two broad kinds of adversaries one almost always encounters in works on secure protocols: *passive* or *semi-honest* adversaries, and *active* adversaries.

Adversaries are also classified on an axis of efficiency—a protocol can be secure against an efficient (polynomial time) adversary, but fail against an adversary without time constraints.

4.6.1 Semi-honest adversary

A *semi-honest* adversary, also called *passive* and *honest but curious*, is one who follows the prescribed protocol, but tries to analyze the information he receives (as part of the protocol) in order to learn more than he should. In general, designing a protocol secure against a semi-honest adversary is easier, and often done on the way to dealing with an active adversary.

4.6.2 Active adversary

An *active* adversary can deviate from the protocol in arbitrary ways, in order to violate the security requirements. Dealing with such an adversary is usually complex and expensive, requiring techniques like zero-knowledge proofs in many or all stages of a protocol.

4.6.3 Computationally feasible adversary

Mostly, protocols' security is based on some intractability assumption (usually unproven but very strongly thought to be true). Such protocols are secure if the adversary cannot solve instances of the underlying intractable problem (and often *only* if he cannot solve the underlying problem—a solution for that problem would often lead to a direct attack on the secure protocol). Thus, the adversaries under consideration are limited to be computationally feasible on a realistic computing model, and on problem sizes induced by the key sizes recommended for the protocol. For example, we do not worry about an adversary who can perform a brute-force search on a 128-bit symmetric key space, as such a search would take on average 10^{21} years on hardware which can examine 5 billion (thousand million) keys per second.

4.6.4 Unbounded adversary

Sometimes, protocols are designed and analyzed to be secure against an adversary without time constraints. A protocol like this is also referred to as *information-theoretically secure*, as (for example) it has no information with which to violate confidentiality properties, no matter how much computation it carries out.

4.6.5 Adversary against hardware-TTP-assisted protocols

From our perspective of security through use of secure hardware, an important aspect of adversarial behavior is how deeply the adversary would tamper with the tools used to execute the protocol. In increasing order of intrusion:

1. No tampering at all takes place. The adversary (who can be a protocol participant, or the oper-

ator of a TTP device) uses the legitimate software and hardware, but tries to learn something from the outputs of that toolset. One example could be reading logs which contain sensitive information.

2. Software-level tampering—the adversary will try to replace software with his own doctored version, which will provide him with more information than specified. He can also try to interfere with the correct software using means enabled by the OS or machine, for example by abusing debugging interfaces. The capabilities of this adversary can be captured by assuming that he has *root-level remote access* to the target computer (which in our setting would be one of the participants, or the TTP) and can use that as he wishes.

3. Hardware tampering. This adversary has physical access to the target computer, and the skill and equipment to attack the computation through the hardware it runs on. Simple examples of his attacks in this case would be probing some system bus [81], or replacing the RAM with one which modifies its contents according to the adversary's strategy. Probing or tampering with the CPU are also options.

The following are platforms which provide the means to security against the respective level of attack:

1. A plain computer

2. A TCG or Intel LaGrande[3]-equipped computer. A computer with a TPM needs the OS to be trusted (ie. part of the TCB), while a LaGrande-based computer can have only a simpler OS in one of the virtual machines as part of the TCB.

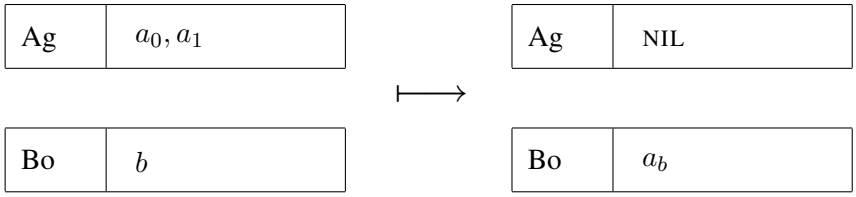
3. A secure coprocessor like the IBM 4758.

4.7 Oblivious transfer

Oblivious transfer (OT) is a fundamental two-party cryptographic protocol primitive, which is used in many higher-level secure protocols, including most protocols for SMC. OT can be framed in

[3]Intel CPUs and chipsets with LaGrande technology, now called Trusted Execution Technology or TXT, have recently been released (August 2007).

the usual way for secure protocols—by specifying the desired functionality[4]. In its simplest form, 1-out-of-2 OT, the functionality is



where all the variables are single bits. In words, Boris retrieves exactly one of two bits that Agnes possesses, without Agnes learning which bit he retrieved. In the OT setting, Agnes is called the *sender* and Boris is the *chooser*. OT in a slightly different form was introduced by Rabin in 1981 [72], and reformulated as stated here by Even et al. [35].

OT can be implemented in fairly simple and efficient ways. For example, the protocol by Bellare and Micali [14] has an overhead of two public key encryptions by the sender and one decryption by the chooser, and is non-interactive [65].

4.8 SFE via blinded circuits

In 1986 Yao devised the first protocol for two parties to securely compute a two-party function, such that each party only learns their own partial result [94]. Like most subsequent works in this area, his protocol works with the two-party function represented as a *boolean circuit*.

This protocol is attractive as it is reasonably simple, and is very efficient in terms of rounds—the participants only need one round of interaction, and the rest of the processing is done locally.

The basic idea of the protocol is that one of the parties, say Agnes, *blinds* or *scrambles* the circuit, to produce an object which can be evaluated similarly to a circuit, but using blinded values[5]. Agnes does not see any values besides her own input and output, and in particular she does not see any circuit internal values. Then the other party, Boris, evaluates the blinded circuit, during which he does know the blinded values being computed, but does not know what they mean, ie. what

[4] See Section 1.4 on page 6 for an explanation of the notation.

[5] The terminology for the transformed circuit has varied, the most popular terms being "scrambled" and "garbled". We believe that "blinded" is appropriate as it captures the aspect of doing specific things with objects/strings whose actual value one does not know.

boolean values they correspond to.

Yao's protocol is formalized and proved secure against a semi-honest adversary in [56].

In describing the protocol, we start with Agnes and Boris agreeing on the function they will compute, as a boolean circuit C. They also agree that Agnes will blind the circuit, and Boris will evaluate the blinded circuit.

4.8.1 Circuit blinding

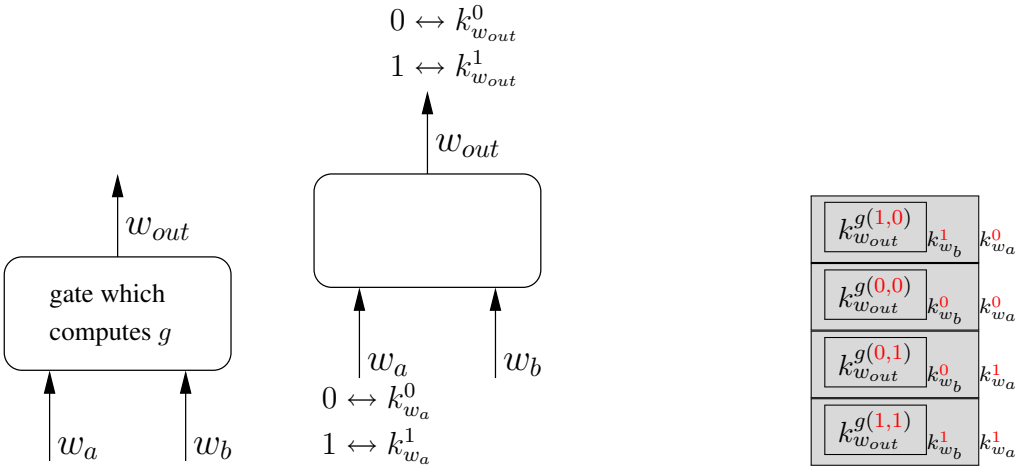
Agnes produces a blinded form of the circuit in this manner:

- For every wire w in C, she assigns a random secret k_w^0 to represent 0 and k_w^1 to represent 1. We call these *blinded bits*, keeping in mind that they differ for each wire w. They can in practice be random keys for a symmetric cipher like AES.
- For every two-input gate g in C, she builds a *blinded truth table* which will enable Boris to compute the output blinded bit given the two input blinded bits, without revealing any additional information. A simple way to do this is to represent each of the four entries in the truth table as double encryptions of the blinded output bit, using the two blinded inputs bits as keys. This is shown in Figure 4.2. The four entries must be in random order, otherwise Boris could learn something about the actual values of the inputs for a gate, based on which table entry he ends up using to evaluate the gate.
- The resulting set of blinded truth tables constitute the blinded circuit BLIND(C).

At this point, Agnes has generated BLIND(C), and sends it to Boris.

4.8.2 Getting blinded inputs to Boris

Now, Boris has BLIND(C), but does not know any of the blinding keys. He also does not know any blinded values. In order to start the blinded computation, he needs the blinded values of the input gates.



(a) The original boolean gate, with its wires labeled.

(b) Agnes has generated random blinded bits for each of the three wires, eg. on wire w_a, $k^0_{w_a}$ will represent 0. These blinded bits would typically be keys for a symmetric cipher.

(c) The final form of the Yao gate. See the main caption for details.

Figure 4.2: Generating a blinded Yao gate, for a 2-input boolean gate computing boolean function g. **Notation** for the blinded gate: a plaintext P encrypted under key K is shown as $\boxed{P}_K$. The Yao gate consists of four ciphertexts, where each ciphertext is for one specific combination of input bits x_a and x_b. Each ciphertext is a double encryption: the plaintext is a blinded bit value corresponding to actual (output) bit $g(x_a, x_b)$. The two keys are the blinded bits corresponding to x_a on input wire w_a and x_b on input wire w_b. When Boris needs to evaluate a Yao gate, he has two blinded bits (the inputs), with which he tries to decrypt each of the gate's rows/ciphertexts. Only one of those decryptions should work, and the cleartext is the output blinded bit for this gate. The other three decryption attempts should fail as at least one of the keys is incorrect.

Agnes's inputs Getting Agnes's blinded inputs is easy: she knows the blinding mapping, so she can just blind her input bits and send Boris the blinded values.

Boris's inputs Getting his own inputs blinded is more complicated. Only Agnes knows the blinding mapping. In order for Boris to obtain from her the blinded bits corresponding to his own inputs, two security concerns must be addressed:

- Agnes should clearly not learn about the values of Boris's inputs, and
- Boris should not learn anything about the blinded values which do not correspond to his actual inputs. Eg., if for a particular input gate g his input is 0, then for gate g he should only learn the blinded value of 0. If he did learn other blinded values for the input gates, he could

evaluate the blinded circuit with varying values for his inputs (recall that he now has blinded values for Agnes's inputs), and thus possibly infer something about Agnes's inputs.

These security requirements are exactly addressed by 1 out of 2 oblivious transfer (OT) (see Section 4.7). Thus, Agnes and Boris engage in an OT protocol for each bit in the blinded values of Boris's input, with Agnes acting as sender and Boris as chooser. OT guarantees that Boris learns exactly one value—the blinded value for his input, and no more, which is what we need here.

Now Boris is ready to evaluate the circuit on blinded values.

4.8.3 Blind circuit evaluation

Boris now evaluates the whole BLIND(C), starting with the input blinded bits and using the blinded truth tables to obtain, in the end, the output blinded bits.

4.8.4 Results distribution

- Boris sends Agnes's output blinded bits to her. She knows the correspondence of blinded bits to actual bits and can so interpret her result.
- Agnes sends Boris the blinding map for each of his output wires, so he can interpret his blinded results.

These two steps can be done in either order, and the order determines which player learns his or her result first.

4.9 Security problems of the simple SFE protocol

The protocol we have presented has two main shortcomings, in terms of what adversarial actions it protects against.

4.9.1 Fairness

In the results distribution phase, the player who first learns their result can then stop their participation in the protocol, so that the peer does not learn their result. Guaranteeing fairness—so that neither player can be caused to learn much less that their peer due to the peer aborting early—is not easy [71].

4.9.2 Active adversary

This protocol is secure only against a semi-honest Agnes. If Agnes was attacking the protocol actively, she could produce a blinded circuit which does not correspond to the original circuit, and in this way cause Boris to compute an incorrect result. Boris would not be able to detect such an attack.

There are multiple techniques to harden this scheme in order to make it secure against active adversaries. A simple one is implemented by Fairplay, which we describe next.

4.10 Fairplay: an SFE implementation

The Fairplay project built a system with which two players can actually carry out Yao's protocol and compute a function securely between themselves [60].

Fairplay specifies a two-party functionality using a high-level imperative language called *Secure Function Definition Language* (SFDL). SFDL looks like a normal imperative language, and draws on the syntax of C and Pascal. The main components of Fairplay are a compiler to translate SFDL to a boolean circuit, and a runtime system which Agnes and Boris can use to carry out Yao's protocol (see Section 4.8), namely, blind the circuit, transfer input values using oblivious transfer, evaluate the blinded circuit, and distribute outputs.

The authors' experiments with Fairplay focused on evaluating SFE implementation techniques, like the OT protocol used and the techniques employed against active adversaries. The evaluation was in terms of computation as well as communication between the parties.

4.10.1 Measures against active adversaries

Fairplay addresses the problem of an actively cheating Agnes (circuit blinder) by having Agnes produce and send to Boris m different versions of BLIND(C), with different randomly assigned blinded bit values. Boris picks one circuit to evaluate, and Agnes reveals the blinded wire values for all the others, so Boris can examine them and confirm they map to the correct circuit. This allows Agnes a $1/m$ cheating probability, while increasing time and communication complexity by a factor of m.

There are more sophisticated (more efficient and more complicated) solutions to actively cheating players, eg. see [40].

4.11 Cryptographic tools

We use only basic tools in our protocols—a secure symmetric encryption scheme and a secure keyed MAC scheme. We do also use asymmetric encryption, but only as an aid in managing symmetric keys, which is a very standard and well-established application of public key cryptography.

4.11.1 Secure encryption

Here we provide a (semi-)formal definition of symmetric encryption, and what security properties a secure scheme provides.

A symmetric encryption scheme $\mathcal{SE}$ operates on the space of keys K, plaintexts X and ciphertexts Y. We also explicitly specify the random bits R involved in the randomized algorithms. $\mathcal{SE}$ consists of three algorithms: $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ where

- $\mathcal{K}\colon R \to K$ is a randomized key-generation algorithm,
- $\mathcal{E}_K\colon X \times R \to Y$ is a randomized encryption algorithm using key K, and
- $\mathcal{D}_K\colon Y \to X$ is a deterministic decryption algorithm using key K.

Informally, the security property of a secure encryption scheme is that, having seen a ciphertext,

an adversary can only learn one thing about the source plaintext, which is its length. The formal statement of this security property allows the adversary to view encryptions of multiple plaintexts of his choice, and only requires that he has a negligible probability of inferring something (even one bit of information) about a "challenge" ciphertext. This security property is called *indistinguishability under chosen plaintext attack* (IND-CPA). Encrypting with an encryption scheme which provides IND-CPA is also referred to as *semantic encryption*.

Note that secure encryption does not only ensure that the adversary cannot reverse the encryption, it ensures that he cannot learn *anything* about the plaintext (apart from its length), including whether it is the same as the plaintext of some other ciphertext under the same key.

One immediate consequence of this security definition is that the encryption scheme cannot be deterministic, or the adversary could trivially learn if two ciphertexts correspond to the same plaintext. A typical way to achieve the required randomization is to use the CBC mode of encryption with a random IV [15, Chapter 4]; there are many other ways too, most involving different modes of operation with an underlying block encryption scheme like AES.

Chapter 5

Related Work

The related work falls into two categories:

- background material which is directly connected with our work, like Fairplay, Yao's circuit-based protocol, our previous PIR/W work and the Oblivious RAM work. These works are covered in Chapter 4.
- In this chapter we cover works which have some similarity or applicability to ours, but which we have not used in the analysis, design and implementation of this work.

5.1 TTP-based PIR

5.1.1 Wang et al.

An extension to our TTP-assisted PIR/W system presented in Chapter 3 is [90]. They present a modified version of our session-transition algorithm which is faster ($O(N)$) but requires more space in the TTP (to hold k entire records[1]). They also present a definition and proof of correctness of their entire protocol, which applies in the same way to ours, as they are very similar.

[1] recall k is the number of accesses within a session.

5.1.2 Williams/Sion

Hot off the press, Williams and Sion have improved the polylog algorithm for oblivious RAM to produce a new randomized algorithm with much lower overhead [91]. Their projected performance of a PIR implementation using this algorithm on an IBM 4764 secure coprocessor (the follow-on device from the 4758, with 64MB of RAM and a more powerful CPU) is less than 1 second to service a retrieval from a dataset of $N = 10^9$ records, which would obviously be much faster than what our implementation provides. It would be interesting to see how this projection plays out in practice, but even allowing for considerable reductions in performance due to unforeseen implementation overhead, this is a big improvement over the current algorithms.

A serious problem with using this scheme in Faerieplay is that it is only shown to be secure against a passive adversary, whereas a TTP for secure computation must assume an active adversary.

Additionally, the Williams/Sion algorithm requires $\Omega(\sqrt{N})$ protected RAM, which is greater than our (and ORAM's) limit for a tiny TTP (detailed in Section 8.5.2 on page 122). However, the efficiency gains promised by this technique support its use on high-end devices like the 4764 for very large problem sizes.

5.2 PIR for anonymous email

An interesting application of PIR is as a component of an anonymous email communication system, in particular to allow a response to an anonymous email to be sent [74]. The idea is to have the recipient of a response retrieve his message using PIR, thus only revealing to any adversaries that some response was retrieved. This technique improves upon the standard reply block techniques used in earlier anonymous email systems, eg. [30].

The security assumptions of anonymizer systems based on mix networks, and multi-server self-reliant PIR schemes are similar—several non-colluding servers. Thus, there appears to be no need for a TTP-based scheme in this application.

5.3 Implementations of SMC

The most relevant SMC implementation for this work is Fairplay, described in Section 4.10.

A system to generate code for cryptographic protocols for functionalities like signatures and encryption is given in [59]. This is a much lower-level system than ours or Fairplay, but shares the aim of building a shared infrastructure for implementing secure protocols from an abstract description, and thus avoiding all the potential pitfalls of doing this by hand for every different scheme.

5.3.1 High-level Languages for general SMC

Usability to programmers is an important property of a general-purpose computing system. Not much attention has been given to this in the (small) literature on actual implementations of general-purpose SMC. Even Fairplay, which is one of the more accessible systems, does not provide tools for developing complex programs, like debugging and validation tools.

A project aiming to produce a usable SMC system using the available self-reliant protocols is SIMAP[2]. They are developing a language for describing SMC functionalities [67], as well as a run-time system (called SMCR) to implement various self-reliant SMC protocols. They have discussed issues like how to specify the injection of users' inputs into the computation, and how to monitor information flow. However usability challenges like debugging ability are not yet addressed. The system is designed around self-reliant protocols, which would limit its scalability to larger problems with a large indirect array component.

5.3.2 Using Smart Cards

The TrustedPals project aims to build an SMC system in a setting where each user has a smart card which is trusted by all the other users [37] for most purposes, and whose only failure mode is *general omission*, where they stop participating in the execution. Given this trusted network of security modules, they reduce the SMC problem to a fault-tolerance problem on the network of secure modules, and show solutions and impossibility results inherited from the fault-tolerance

[2] See http://www.sikkerhed.alexandra.dk/uk/projects/simap.htm

literature.

The assumption of secure modules for every participant in the system is a stronger one than of a central TTP, as it requires more modules, which is either more expensive, or uses cheaper and less secure modules. If this trade-off is made however, it is possible that this approach provides a faster means to achieve SMC. The performance results in this paper do not include complex functions with a significant indirect array component, which we argue are a main weakness of existing SMC protocols (see Section 6.6.2 on page 94).

5.4 Trusted hardware

Hardware modules like the TPM are intended to protect data—keys and software measurements—as opposed to computation. Also, they do not protect against physical tampering attacks, which are an important part of our threat model.

Current research (e.g., [55, 84]) and product efforts (e.g., [29]) explore the notion of a secure computing environment built around a security-enhanced CPU, with security provisions extending to the whole system by means of partitioning, and memory encryption and checking. These systems (if implemented) offer varying degrees of protection against a physically present and dedicated adversary with a drill and bus and RAM probes, but none of them address this threat directly.

The XOM project [55] investigated how to design a secure desktop-oriented processor architecture and operating system such that only the processor needs to be trusted, and not the OS and the RAM. The adversary's goal is to copy software which is run on the machine. They leave open the implications of the adversary observing a program's memory access pattern.

The AEGIS project [84] investigates the use of an innovative way for a processor to wield a secret key, by using a *Physical Random Function* based on random delays in silicon gates. It also assumes an untrusted RAM, but leaves open the consequences of exposing the RAM access pattern.

Peter Winkler has a patent on an "Electronic trusted party"–a device resembling a hand calculator, which multiple people can use to privately enter data and observe the result of a function on all the data [92].

5.5 Hiding address sequence

HIDE [100]: protect memory-access pattern on a XOM-like machine. Targeted at a PC-like machine and applications. HIDE is much less strict about information leakage than the security requirements for SMC mandate.

Arc3D is a subsequent work in this field, which improves on some of the weaknesses of HIDE which make it easy for an attacker to learn the secret permutations used by HIDE to hide real addresses [43, 44]. .

5.5.1 Cryptographically weak devices

Remotely keyed encryption schemes seek to enable high-bandwidth encryption (on a host machine) using long-term keys held in low-bandwidth devices like smart cards [18]. This work shares the theme of enabling large computations using a small trusted space, but is otherwise quite different as it has no obliviousness requirements, and an adversary controlling the host can decrypt ciphertext until he is removed.

On a similar theme, Modadugu et al. have developed a prototype using an untrusted host to help a Palm Pilot with the computation of generating RSA keys [61].

5.6 Correctness of SMC programs

As the complexity of functions to be computed securely grows, so does the challenge of getting the implementation right. Here we are referring just to the implementation of the function, and not of the underlying SMC system. SMC functions are prone to the usual bugs which result in them not computing the correct result. Little work has been done to address the difficulty of debugging complex functions implemented for SMC, in any environment.

A related problem is the possibility that the function may reveal more than the desired amount of information, just through the outputs it produces. Stated another way, the outputs may have more dependencies on the inputs than desired, and thus provide more information about the inputs.

This situation can easily arise with complex functions. This problem is tackled (independently of SMC) in the field of *information flow control*, eg. [63]. Tools from that field provide ways to track information flow, and thus reduce unexpected flows from inputs to outputs.

5.7 Self-reliant SFE

The blinded-circuit solution to self-reliant SFE is not the only one proposed. Other approaches make use of different representations of the function f to achieve different properties like limiting the communication burden on protocols with large inputs but sub-linear communication in the non-private setting [64]. This work also provides a related functionality to indirect arrays—efficient lookup tables with dynamic contents. However the access pattern to the tables must be static—the technique does not provide an efficient RAM capability, which is left as an open problem.

5.8 Protecting control flow

An important part of Faerieplay is preventing an adversary from obtaining information by observing the sequence of operation addresses that the TTP executes. In general, the operations could be, for example, instructions or circuit gates. In Faerieplay we use the fixed nature of a circuit to render control-flow information useless to the adversary: it is the same for all inputs to the computation.

Another way to prevent information leakage through control flow was proposed by Zhang et al [99], in the setting of executing code sent by a server to a smart card, such that an adversary does not learn about the program by observing the communication. The program is divided into partitions, whose addresses do not reveal any control flow information, and then the partitions are sent by the server to the smart card as needed, and cached for efficiency. The adversary is apparently assumed not to know the program code, and is trying to learn something about it, but the adversary model is not clearly explained. An adaptation of their technique to assume that Mallory knows the program, and prevent him from learning about the execution, could be an alternative to using a circuit to ensure that the execution trace does not leak information. However, this possibility requires more investigation.

5.9 Specialized SFE protocols

A large body of work exists on designing specialized protocols for solving specific two-party or multi-party problems privately:

- Database operations, like `join` and `intersection`, across tables from two owners [4, 54];
- Privacy-preserving protocols for geometric problems [10, 38];
- Privacy-preserving data mining, eg. [36]
- Supply chain management techniques which enable collaboration among competitors without revealing proprietary information, eg. [9]

Although the majority of SFE work is theoretical, some of these works include concrete running time analysis [4] and even real prototypes [3].

Many of these protocols could be as efficient or more efficient than our general-purpose TTP-assisted technique, but they have the serious disadvantage that a whole new cryptographic protocol needs to be designed (and validated and securely implemented) for a new problem or class of problems. The difficulty of protocol design would seem to preclude the specialized secure protocols from scaling to the many different settings which can benefit from SMC.

Chapter 6

Secure Computation

In this chapter we will develop the brief introduction to Secure multiparty computation (SMC) given in Section 4.4, and we will present a critique of the main direction of the literature in solving SMC problems. This motivates our work on Faerieplay which starts with Chapter 7.

6.1 Introduction

Secure multiparty computation (SMC) aims to enable multiple parties to engage in a joint computation on their private or proprietary data. They would like to learn something which is a function of all their data, but none of them want to reveal their actual data to any of the others.

Many examples come to mind: Agnes wants to convince Boris that she satisfies his security policy, without revealing the values of her credentials to him; a group of sysadmins at different organizations want to analyze all their log files for signs of attacks, but none of them can reveal their own company's logs externally. We have elaborated on these and other scenarios in Section 1.3.

6.1.1 Intuition of the setting

Here we will describe the aspirations of SMC, focusing on the way they differ from many other security settings, like secure communication and access control.

Sensitive data

SMC only makes sense in the case where the data needed to produce a result in a multiparty computation is confidential and sensitive to some participants, such that the problem cannot be solved by having one party tell their input to the other. To illustrate this distinction, consider the case of a client Boris who wishes to buy a service from service provider Agnes. They want to figure out whether they can agree on the price.

- If either of them does not consider their price limit sensitive, that person can tell the other their limit, and the other just says yes or no.
- However if both consider their price limit sensitive, then we have a candidate problem for SMC—Agnes and Boris need to figure out whether their price limits overlap, but neither wants to reveal their limit.

If the computation participants only consider *some* of their data to be sensitive, they may be able to partition the computation such that no participant has to release their sensitive data outside her own domain. The Jif/split tool for distributed Java programs with data ownership labels implements such partitioning [98]. However, in case all the data is sensitive, or a partition which satisfies the users' constraints cannot be found, applying SMC techniques would be the next step.

A strong adversary

Each participant believes that the other participants will undertake powerful attacks in order to learn more than they should from an SMC interaction. We present adversary models, with their practical implications, in Section 4.6. One consequence of facing dedicated adversaries is that SMC cannot rely on "gentlemen's agreements" at any level. For example, we cannot assume that some component will protect a stakeholder's data based on an access-control decision, unless this component is in that stakeholder's *trusted computing base* (TCB), and the data never leaves that TCB without explicit means of ensuring its confidentiality.

6.2 SMC specifications

The specification of an SMC problem is usually split into two parts: a *functionality*, which specifies what result the parties want to compute, and a security requirement, which specifies limits on what data the parties can learn—usually that the parties should learn no more than what the result of the functionality tells them.

6.2.1 Specifying the functionality

The functionality can be specified in "normal" ways for specifying programs, and it does not include any aspects related to computing it securely. For example, one could define the function in an imperative programming language:

```
1  /* Millionaires' problem between Agnes and Boris: the functionality
2   * specified in C. */
3
4  /* Convention to indicate the relation between data and user: field X in the
5   * two_party_data structure is associated with user X */
6  struct two_party_data {
7      int agnes;
8      int boris;
9  };
10
11 two_party_data isLarger (two_party_data in) {
12     two_party_data result;
13
14     result.boris = result.agnes = in.agnes > in.boris;
15
16     return result;
17 }
```

6.2.2 Specifying security requirements

The security specification for an SMC problem sets limits on what information can flow where in a protocol. This is trickier than specifying the functionality, as it always specifies a negative property—eg. no more than x amount of information can flow outside Bob's TCB. The security model usually comprises the bulk of an SMC work.

The usual specification, which looks deceptively simple at first, is that every computation participant should learn no more than what the functionality specifies for them. Several factors conspire to make the completion of this task quite complicated:

- Formalizing and proving negative data flow properties like this is very difficult.
- Any formalization and proof is very dependent on the adversary against whom the protocol must defend. For example, it is much easier to construct and prove a protocol against a passive/semi-honest adversary than against an active one.
- The TCB assumptions one makes also influence the resulting security definitions and proofs a lot.

We spend some time setting up formal definitions for our system in Chapter 8.

Note that for complex functionalities, it may be difficult to show what their intrinsic data "leakage" properties are, ie. exactly what is revealed by the output of the functionality, even if implemented perfectly securely. Such questions are studied under the umbrella of *information flow control*, and addressing them could involve annotating the source code of the functionality in order to help automated checkers to establish the absence of undesired data flow [63].

6.3 Ideal and actual solutions to SMC

6.3.1 Ideal solution to a SMC problem

All secure protocol problems have a "trivial" ideal solution: instantiate a so-called ideal trusted third party (TTP) (see Section 4.5 for an introduction to TTPs) which is fully trustworthy, load a program

to compute the desired functionality (eg. check a list of credentials against a policy, or execute a general program), have each participant send inputs and receive outputs from the TTP over a private channel. Security requirements are met trivially in this ideal solution because the TTP is assumed to behave correctly.

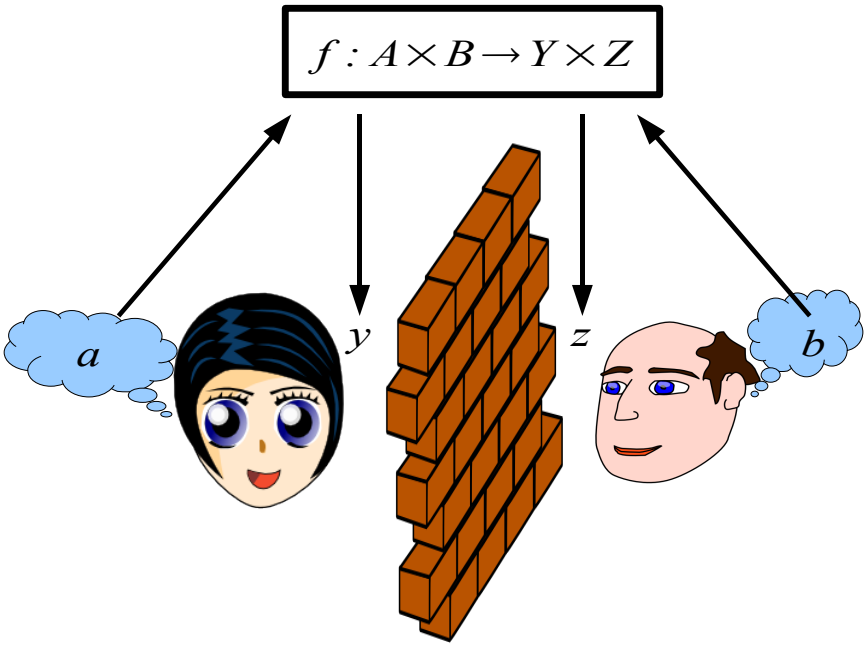


Figure 6.1: An ideal secure protocol for any 2-party functionality f.

6.3.2 Actual solutions to an SMC problem

Having thus defined the ideal solution, a typical SMC paper goes on to develop a protocol which does *not* use a TTP, but looks to a specified adversary the same way as if a TTP was actually used (formalizing and proving this property often takes the majority of space and ingenuity of an SMC work). Thus, the security properties of the real protocol are the same as those of the ideal TTP-based protocol, often modulo some probability of failure and dependence on intractability assumptions. (Many protocols do employ TTP's, but only in a way which does not require the TTP to see participants' data, for example as certificate authorities in a PKI, or as providers of a shared random string.)

6.4 Self-reliant protocols

We call the protocols which have been designed as actual solutions for SMC *self-reliant*, as they insist that each participant's TCB should be only "himself". A similar term has not been used in the literature previously, because it was always implicit: a self-reliant protocol was considered to be the only kind worth pursuing, and thus does not need to be defined as such.

In this work we examine several aspects of self-reliant protocols, which suggest that the costs of the self-reliant approach are high enough that investigating other approaches is worthwhile. Here we will claim that there is in practice no such thing as a self-reliant protocol, which reduces the distinction between self-reliant and TTP-assisted. Then, in Section 6.6 we present specific costs and weaknesses of current self-reliant techniques, like inefficiency, complexity, and inflexibility.

6.4.1 Implicit TTP dependencies of self-reliant protocols

If we admit some reasonable flexibility in the appearance of a TTP, we can spot TTPs lurking inside standard self-reliant protocols.

The protocols in question are usually very complex and subtle, and so must be implemented by specialists, ie. almost certainly not by the user. Thus, from the user's viewpoint the implementer becomes an external trusted party. This problem is especially acute as the complexity of the protocols implies that there are many possibilities for hiding code which compromises the confidentiality properties promised by the protocol; see for example work on malicious cryptography [96, 97].

Additionally, there has recently been increasing concern about the trustworthiness of hardware foundries, eg. [2, 51]. The concern is that hardware may be maliciously constructed to violate security and privacy properties of software executing on it, for example by storing data and transmitting it to an external adversary, or again engaging in malicious cryptography, for example embedding secret data in "randomly" generated keys. The untrusted foundry problem applies to all computation, including our TTP-assisted proposals, but it is more significant in the case of self-reliant secure protocols, as it reduces the difference in TCB requirements between self-reliant and TTP-assisted protocols.

6.5 Circuits as a computational model

Most of the protocols for SMC represent the function or program to be executed as a boolean or arithmetic circuit. The circuit is an executable format in a similar manner way as RAM executable, which is a series of instructions for some machine architecture. A circuit is usually thought of as a directed acyclic graph (DAG) with nodes representing gates and edges representing data flow, but it can also be seen as a flat list of gates, in topological order, in which case it begins to resemble a standard RAM executable quite clearly.

6.6 Shortcomings of the self-reliant circuit-based approach

Self-reliant circuit-based solutions for SFE incur a communication and computation cost at least linear in the circuit size, and thus the circuit size is the primary performance metric. In addition, however, each gate is usually executed using an expensive algorithm or protocol in order to achieve strong security properties, thus increasing the cost even more.

6.6.1 Large constant overhead for scalar code

It is possible to compile some functions into a boolean circuit which can be executed serially[1] with only a constant factor slowdown compared to a RAM machine representation of the same function. For example, a scalar addition on 32-bit integers can be compiled to 64 gates (with 3 inputs each); a RAM program would require one 32-bit addition instruction. Likewise, any sequence of scalar operations has the same asymptotic cost in boolean circuit form as in RAM program form.

In these cases, the overhead of executing the function using a boolean circuit-based SMC protocol over executing the same function "normally", ie. on a RAM machine, is also a constant factor. The contributors to this constant overhead are:

- Each gate (analogous to an instruction) works on single bits, and thus more gates are required than instructions which operate on words of 32 or 64 bits,

[1] meaning with a circuit simulator which executes the gates one by one

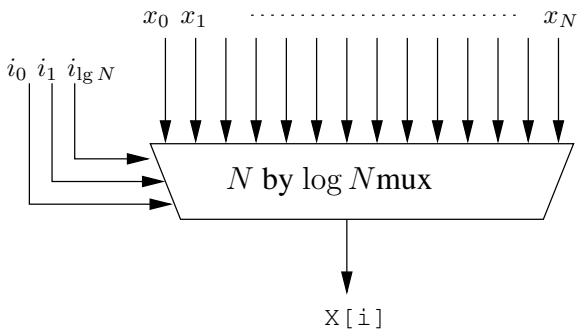


Figure 6.2: Indirect array lookup in boolean circuit, on an array with N elements.

- Each 2-input boolean gate requires several cryptographic operations for execution. For example, in Yao's two-party protocol, each gate requires:
  - Generation of two random symmetric keys, and encryption of two ciphertexts under each of the 4 permutations of the keys,
  - Random re-ordering of the 4 ciphertext pairs,
  - Decryption of (expected) 4 ciphertext blocks

6.6.2 Linear overhead for array code

In addition to scalar operations, high-level code can use *indirect arrays*. An array access is indirect if the index is non-constant, and thus on a RAM machine the array lookup requires two lookups—one to get the index from RAM, and the other to read or update the array value at that index. An indirect array access is cheap on a RAM machine (disregarding issues of cache efficiency)—it takes constant time.

In the traditional blinded-circuit approach to general SFE, indirectly-addressed arrays are a major source of inefficiency. Each bit in the array is represented by a wire in the circuit, and the array lookup (or update) is translated to a $\lg N$ by N multiplexer, as illustrated in Figure 6.2. There is no more efficient way to encode the array and lookups against it in a one-pass circuit—if the array lookup sub-circuit included any fewer than N gates (of small constant fan-in), it would not be able to produce the value of some array index.

Thus, code which uses many indirect array accesses will translate to a very large boolean circuit, and take correspondingly long to evaluate. This is a serious problem, as indirect array access is frequently an essential component of efficient algorithms, as we elaborate in Section 7.3.

The existing SMC literature leaves open the problem of indirect array access, eg. [64].

6.6.3 The SMC status quo warrants introducing a tiny TTP

Given this state of affairs with SMC solutions, we decided that this was a good problem for secure hardware:

- Self-reliant protocols have been extensively studied but still have a significant shortcoming which renders them very inefficient for many real-world problems,
- Using a tiny TTP allows a considerably more efficient solution with similar security properties as self-reliant protocols.

6.7 Conclusion

As traditionally pursued, self-reliant solutions for SMC have enough weaknesses to warrant serious investigation of other approaches, which we begin in the next chapter.

Chapter 7

Faerieplay

7.1 Introduction

At this point we have covered all the relevant background work, the necessary tools and concepts, and our own preliminary projects, and we are ready to start with the main part of this thesis. We have designed and built a scalable secure computation system utilizing standard secure hardware. We introduce: *Faerieplay*[1].

Our preliminary projects, in Chapters 2 and 3, considered a specific secure computation problem, namely retrieving data from a remote server while hiding the contents *and* identity of that data from the server: private information retrieval (PIR).

Then, in Chapter 6, we described secure multiparty computation (SMC), and some of the SMC work thus far. A major critique of the existing work on SMC is that the insistence on *self-reliant* protocols results in multiple performance and scalability problems. A fundamental tool of efficient algorithms, *indirect addressing*, which we discussed in Section 6.6.2, is very inefficient in all self-reliant SMC protocols, when compared to a standard RAM computation model. Towards improving this inefficiency, we proposed using a *tiny TTP* as part of the computation protocol. A tiny TTP has two main properties: it is *trusted* by the computation participants to enforce their security requirements, and it is very small in terms of memory size. We develop Faerieplay's security model

[1]The name Faerieplay combines *Fairplay*, which provided much inspiration for us, and the idea from the theoretical SMC community that a TTP is just an idealized notion, like a *fairy*.

in Chapter 8, and in particular the details of our tiny TTP in Section 8.5.

7.1.1 Faerieplay chapters road map

In the next four chapters we report on our design, security model, implementation and evaluation of the Faerieplay SMC system. In this chapter we start with a quick description of Faerieplay which helps to frame the subsequent discussion. Then we describe some important problems for which Faerieplay provides fundamentally more efficient solutions than existing SMC solutions. Then we outline the structure of a Faerieplay system. Finally we present a comparison with oblivious RAM, which provides another means of TTP-based SMC.

The next three chapters present details of Faerieplay: in **Chapter 8** we develop a model of the system and specify the security assurances provided; in **Chapter 9** we give specifications for the Faerieplay source languages and virtual circuit machine; and in **Chapter 10** we present our implementation and experiments.

7.2 Faerieplay in two minutes

Faerieplay provides a secure multiparty computation service, whose users have to trust a *tiny TTP* to get similar security assurances as with traditional SMC protocols, but with much improved efficiency and scalability.

7.2.1 Program as a circuit

The main role of Faerieplay is to compute functions. What should be the form of these functions? We have considered two options. The first is as a *circuit*, which has been the standard in SMC work. Secondly, as a *RAM program*—secure computation of RAM programs, in a similar security setting as ours, was investigated in the oblivious RAM project [41]. We decided on an approach which could be seen as a *hybrid*: a circuit which is augmented with special gates (*array gates*) to abstract indirect array access. We implement array gates using an algorithm which we initially applied to our private information retrieval (PIR) project (Chapters 2 and 3), and which is based on algorithms

from oblivious RAM. This approach to SMC is more efficient than either pure approach:

- Compared to a pure circuit implementation of algorithms with a substantial indirect indexing component, our optimized evaluation of indirect indexing provides a large improvement. See Section 6.6.2 for an outline of the difference, and Section 7.3 for concrete case studies.
- Compared to a pure oblivious RAM approach, we again find that our approach offers efficiency benefits, as we only apply the expensive algorithm for secure indirect indexing where needed. See Section 7.7 for a theoretical comparison, and Section 10.5 for what we observed in practice.

7.2.2 Tiny TTP

Since the TTP should be trustworthy in an environment where it is exposed to physical attack, it needs heavy armoring against such attacks. This makes it difficult to provide lots of RAM, never mind secondary storage, in the armored perimeter. The particular device we use, the IBM 4758 which we introduced in Section 4.2.1, has 4MB of RAM, and even with just that, it costs about US$2,500.

In Faerieplay, we explore the hypothesis that the way to make trustworthy TTPs more accessible is to make their protected area as small as possible, and hence as cheap as possible. Accordingly we base our designs on the assumption that the protected space in the TTP is *tiny*. More concretely, we specify that the TTP has only the following amount of protected storage:

- $\mathrm{O}(\log N)$, where N is the size of the largest array in the current program. This is to allow the TTP to manipulate a constant number of pointers into any of the datasets it is working on.
- $\mathrm{O}(B)$, where B is the block size of the symmetric cipher used to encrypt sensitive data stored on the host. Larger data items are processed in pieces; for example if the tiny TTP needs to re-encrypt a data item which is larger than B under a new key, it does this block by block.

We would like to keep the TTP's algorithms' space requirements independent of M, which is the maximum size of a single element in the working datasets/arrays.

4758 as a tiny TTP If you are wondering how the 4MB secure RAM available on the IBM 4758 corresponds to $B + \log N$—the relationship between the abstract limits and the 4758's total RAM is not direct and not very important either:

- Much of the RAM in the 4758 is taken up by the code of the *Circuit Virtual Machine* (CVM), and relatively little of it is left for the computation's data.
- The 4758 is just our current prototyping environment. We have been designing a specialized hardware device which can implement the Faerieplay CVM much more efficiently. It would provide much less secure data space than the 4758, but have optimized hardware implementations not only of the cryptographic primitives, but also of the other primitives used in Faerieplay, like the comparator which comprises the sorting networks used in our PIR/W algorithm (see Section 3.3.2).

7.3 Case studies for SMC

In the next two sections we seek to demonstrate that efficient optimization algorithms use indirect addressing extensively, and thus implementations in an SMC setting would benefit from our approach which improves the efficiency of indirect addressing relative to pure circuit-based techniques. We analyze two particular optimization problems, and we examine several algorithms to solve them and the algorithms' interaction with SMC. The problems we consider are:

- finding a shortest path between two vertexes v_1 and v_2 in a directed weighted graph.
- Electricity auctions: scheduling electricity suppliers with different cost curves to meet a given demand at lowest cost—an instance of combinatorial optimization, with extensive real-world use.

7.4 Graph shortest paths via Dijkstra's algorithm with heaps

The single-source shortest path problem, in a multiparty setting is:

- Boris has a directed weighted graph $G = (V, E)$,
- Agnes has two vertexes $v_1, v_2 \in V$

Agnes should learn a shortest (minimum weight) path from v_1 to v_2 and the cost of such a path.

It seems quite plausible that Boris wishes to keep his graph private, as a commercial asset for example, and Agnes wishes to keep her start and end-points secret, as she does not want the graph provider to learn where she wants to go. Thus, the security requirements for this setting are that Agnes learns *only* her path, and Boris learns nothing.

Apart from direct applications to routing and similar problems, shortest path searches are a very common subroutine in various larger optimization algorithms, in which case the parameters to the graph search need to be kept private as part of ensuring the privacy of the larger computation. In fact, our electricity auctions case study uses a shortest path search as a subroutine.

Time complexity of SP related to complexity of indirect indexing

The overhead of circuit-based computation relative to RAM computation varies across different operations: scalar arithmetic has similar complexity in both cases, whereas indirect indexing has much higher complexity in the circuit model. Here we examine how this affects the relative complexity of two algorithms for the graph shortest path problem.

Dijkstra's algorithm using a heap priority queue takes $\mathrm{O}((E + V) \log V)$ time[2], which is $\mathrm{O}(E \log V)$ in the usual case of a connected graph [28], provided that indirect indexing into the heap is a constant-time operation. To examine the setting where indirect indexing is non-constant, let h be the complexity of indirect indexing:

Definition 1

$$h(N) \stackrel{\text{def}}{=} \textit{time complexity of an indirect lookup from an array of N elements}$$

[2] As usual, we use V to indicate both the set of vertexes and the size of that set. Likewise E is the set and the number of edges.

The algorithm makes $O(V \log V)$ accesses into the heap, which is of size V. In our Dijkstra implementation targeted at the Faerieplay circuit machine, the Faerieplay program maintains an array of all E edges (see Section 10.6.1 on page 185 for details). The implementation accesses this array $O(E)$ times. Thus, in total the cost is

$$\begin{aligned} \text{DIJKSTRA\_SMC}(E, V) &= O(V \log V\, h(V) + E\, h(E)) \\ &= O(E \log V\, h(E) + E\, h(E)) \quad [\text{if } E \geq V] \\ &= O(E\, h(E) \log V) \end{aligned}$$

Specific costs in different settings In a standard RAM computation, the cost is

$$\text{DIJKSTRA\_RAM}(E, V) = O(E \log V)$$

In the SMC scenario, if a direct circuit implementation is used for the heap and edges array, $h(N) = N$, so the running time degenerates to

$$\text{DIJKSTRA\_CIRCUIT}(E, V) = O(E^2 \log V)$$

If a TTP running the square-root PIR/W algorithm for secure indirect indexing is used, $h(N) = \sqrt{N} \log N$, yielding a total running time of

$$\begin{aligned} \text{DIJKSTRA\_SMC\_TTP}(E, V) &= O(E\sqrt{E} \log V \log E) \\ &= O(E\sqrt{E} \log^2 E) \end{aligned}$$

Bellman-Ford As an additional example of analyzing the impact of non-constant indirect indexing, we look at the Bellman-Ford graph single-source shortest path algorithm. This algorithm solves the shortest path problem in $O(VE)$ time on a RAM machine. It is more regular than Dijkstra's algorithm—it processes the graph edges in a fixed sequence, regardless of the graph or the vertexes in the search. However, it still relies on indirect indexing: for each edge operation (a *relaxation*),

it needs to look up and update parameters (distance estimate and predecessor) of the vertexes incident on that edge. Those vertexes do depend on the concrete graph, and thus cannot be directly accessed. Thus, with the cost of indirect indexing of the vertex array, the cost of Bellman-Ford is $\mathrm{O}(VE\,h(V))$. Implemented as a circuit, this is $\mathrm{O}(EV^2)$.

In Table 7.1 we summarize the different running times of both algorithms with the two SMC models—scrambled circuit, and Faerieplay.

Algorithm	Non-private	SMC circuit: Fairplay	circuit/TTP: Faerieplay
Dijkstra	$\mathrm{O}(E \log V)$	$\mathrm{O}(E^2 \log V)$	$\mathrm{O}(E\sqrt{E} \log^2 E)$
Bellman-Ford	$\mathrm{O}(EV)$	$\mathrm{O}(EV^2)$	$\mathrm{O}(EV \log V\sqrt{V})$

Table 7.1: Running times for Dijkstra's algorithm with heaps using various strategies: without privacy provisions; using only a blinded circuit; and using a circuit and TTP for faster indirect addressing. For comparison we show the Bellman-Ford algorithm, which has a smaller but still substantial indirect-indexing overhead.

7.5 Electricity auctions

Electricity auctions give rise to a complex optimization problem which is solved daily on a very large scale. The setting is that some number of electricity generators are competing to supply the demand in a given market. The demand is represented simply as an expected amount of electricity which will be used by consumers in the market, as well as a peak demand, which the generators should be able to meet at short notice, but do not normally have to provide. The current direction in electricity markets is to allow for individual consumers to place bids for electricity too, as opposed to being aggregated into a single total expected demand quantity.

The optimization problem for *electricity supply scheduling* asks: how much electricity should each generator produce so that the system can provide for the average and peak demand at lowest cost? In particular, each generator places a bid, which is a discrete upward-sloping supply curve of price vs. power supplied. The problem is to allocate how much electricity each generator should supply (and be paid for) in order to meet the average and peak demands, and minimize the cost. This is an instance of *combinatorial optimization*.

7.5.1 Importance of data privacy

Electricity auctions have real privacy requirements, as the efficient operation of a market depends on the secrecy of the operational information of each participant. Knowing a competitor's bids would allow an adversary to shape their own bids, or their operational plans accordingly, thus potentially gaining an unfair advantage over other market participants.

In most US regions, electricity scheduling is carried out by the regional *Independent System Operator* (ISO). For example, the Midwest ISO, MISO (see http://www.midwestiso.org/) coordinates the market for 9 Midwestern states, and parts of 4 more. The ISO collects the bids from each generating company, and carries out the computation to schedule the generators. If any ISO insiders are motivated to do this, they could abuse their knowledge of the generator's bids for fun or profit. Also, any software vulnerabilities in the scheduling software could allow bid information to reach undesirable recipients.

Thus, we have a multiparty computation, where each participant's data are sensitive, and there are incentives to cheat by trying to learn competitors' data—this is exactly the setting for SMC.

7.5.2 Detecting bad behavior

One reason for the ISO to have direct access to bids is to be able to detect collusion among the generators, which could reduce the effectiveness of the competitive market. It would be possible to add extra checks to the computation, which try to detect bad behavior and alert the ISO if any is suspected. There is ongoing research on reduction of privacy provisions in response to bad behavior, e.g. [87]. Similar ideas could apply here, to decide if, when and how much bid information would be divulged to the ISO or to a fourth party if the computation detects suspicious behavior.

7.5.3 Details and algorithms

Algorithms for electricity scheduling have been developed for decades, and they all face a similar challenge—the problem is an instance of integer programming, possibly non-linear. Integer programming is NP-complete, and thus must be approximated if dealing with non-trivial problem

sizes [93].

In our investigation, we have focused on the system presented in Sherri Ann Koenig's 1975 masters thesis on electricity scheduling [53, 62]. It gives an approximation algorithm for optimally scheduling a set of generators, for a number of time periods. Each generator has a cost schedule (upward-sloping supply curve). Scheduling is subject to a *demand constraint* (expected demand) and a *reserve constraint* (peak demand). This formulation of the problem is simpler than the contemporary power scheduling setting, as it does not have competitive buyers (no demand curve), and does not account for transmission constraints. We chose it as a (relatively) simple starting point in this area, though already quite complex.

The data in electricity scheduling

The parties involved are N generators (the bidders) and the ISO (the auctioneer).

The scheduling takes place for each of T *time periods*, of potentially differing lengths. 24 periods of one hour each is the normal situation, for the *day-ahead market*, where scheduling is done once per day for the coming day. Thus, for each hour of the next day, each supplier is allocated a fixed amount to supply.

The *public* parameters (ie. known to all participants) of the computation are:

- An expected demand D_t in megawatt hours (MWh) for each time period, which must be met.
- A reserve capacity $R_t \geq D_t$ which must be available at short notice. In normal conditions, where D_t needs to be supplied, some of the spinning generators must be supplying an amount lower than their capacity. If required by an unusual surge of demand, they can quickly increase their supplied amount (at a higher price), to increase the total supplied up to R_t.

The *private* inputs from each generator/bidder i are:

- A discrete supply curve with K_i points. Each point indicates the price ($/MWh) that the bidder will charge for a given power capacity (MWh). As usual the supply curve slopes upwards: generators charge a higher price per unit for more units (power) supplied.

- A fixed startup and shutdown cost, incurred in addition to the operating cost if the generator must stop or start at any time period.

The result of the computation is, for each generator i, and each time period t:

- $x_{i,t}$: a boolean which indicated whether generator i is spinning at time t.
- $y_{i,t,k}$ (for k in [1 .. K_i]): a fraction which indicates how much of the capacity in the k-th segment of generator i's supply curve is to be supplied at time t. For the integer values of $k \in 1..K_i$, the function $y_{i,t,k}$ looks like $[1, \ldots, 1, p, 0, \ldots, 0]$, for $0 \leq p \leq 1$, ie. the generator supplies from the start up to a point in their supply curve.

The output in summary: during time period t, is generator i spinning, and how much power is it producing.

Algorithm outline

The optimization algorithm is a branch-and-bound (BB) search: fix some of the variables and minimize cost by varying the remaining free variables; re-choose the free variables and iterate.

For each BB node (ie. for each binding of variables), the algorithm computes a cost minimization by solving a Lagrangian relaxation of the main problem. The Lagrangian relaxation is easier than the main problem—it is an instance of DAG shortest path. The algorithm finds optimal Lagrange multipliers by iterative sub-gradient optimization.

7.6 Faerieplay outline

Faerieplay consists of two major components: a compiler which translates high-level programs into a circuit, and a *Circuit Virtual Machine* (CVM), which evaluates a circuit on given inputs. The CVM runs inside a *Secure coprocessor* (SCop) like the IBM 4758, and uses data storage services provided by the SCop's host.

7.6.1 Compiler

The Faerieplay compiler translates several high-level languages into an arithmetic circuit augmented with *array gates*, which abstract secure and efficient indirect array access. The compiler formats the circuit as a sequence of gates in topological order.

7.6.2 Circuit virtual machine

The CVM runs inside an IBM 4758 *Secure coprocessor* (SCop), and evaluates a circuit generated by our compiler: it receives inputs from the computation participants (including array inputs), runs the circuit on them, and then sends the results to their respective recipients. The CVM implements the array gates by using the PIR/W algorithm presented in Chapter 3.

7.6.3 SCOP-host API

The host of the SCop running the CVM provides data storage services to the CVM. The host's data API is in terms of *containers*—a container is a numbered sequence of data items. The API is:

- **create** a container, with a given length and element size. Return a reference to the container—the remaining API calls use that reference to identify the container. The CVM also specifies a hierarchical name for the container–this is not essential but does help with understanding a computation's use of containers.
- **read an element** from a container.
- **read multiple elements** from a container; used to amortize the high fixed cost of communicating from the SCOP to the host.
- **write an element** to a container.
- **write multiple elements** to a container.
- **read and write the header** of a container. The CVM can use this eg. to store (encrypted) symmetric keys.

- **dispose** of a container, when done with it.

The host is not trusted (see Section 8.2 for the adversary model), and the CVM is responsible for providing the security properties specified in Chapter 8.

See Section 10.3.2 for more details on host containers.

7.6.4 Architecture and work flow

Faerieplay's system architecture is outlined in Figure 7.1. It gives a high-level view of what Faerieplay provides. With reference to that figure, here are the main steps involved in a Faerieplay computation:

1. Users compile the source code for the function f using the compiler, to produce a circuit. This circuit is analogous to a standard machine-executable file, except that instead of machine instructions it contains a sequence of gates.

2. Users provide the circuit and their respective inputs to the CVM.

3. The CVM evaluates the circuit on the inputs:

   (a) It populates the values of the *input gates* with the users' inputs.

   (b) It evaluates the gates one by one, in topological order. This evaluation is specified in detail in Section 9.5.

   (c) It fills in a value for each gate after it evaluates the gate.

   - It evaluates array gates using the PIR/W algorithm (Chapter 3). It maintains several containers on the host for each array.

   (d) When it has computed all the gates, the results of the computation are available at the *output gates*, and the system sends them to their respective owners.

A more detailed look at the components of the Faerieplay CVM is shown in Figure 7.2.

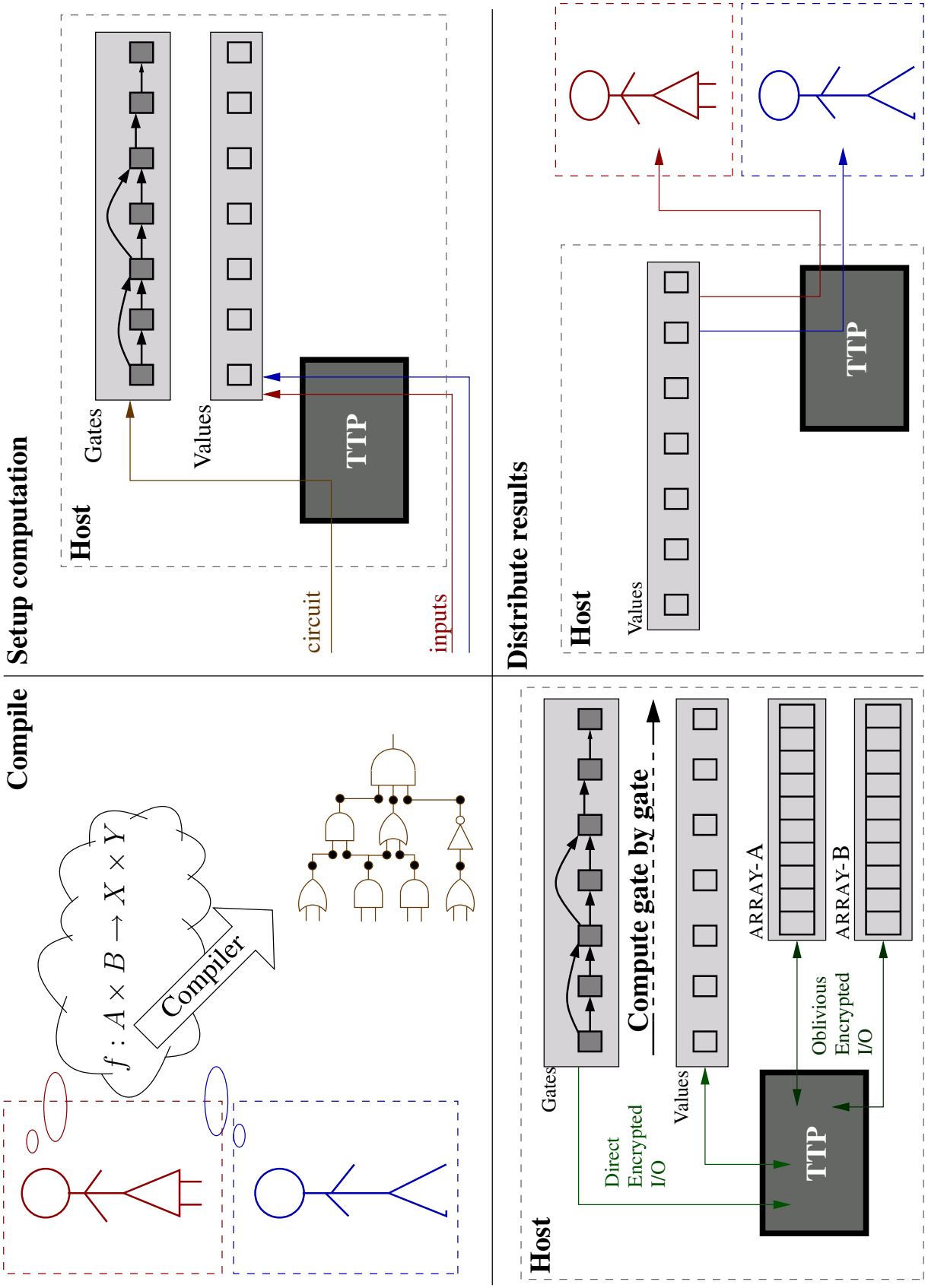


Figure 7.1: Outline of a Faerieplay computation, as described in Section 7.6.4.

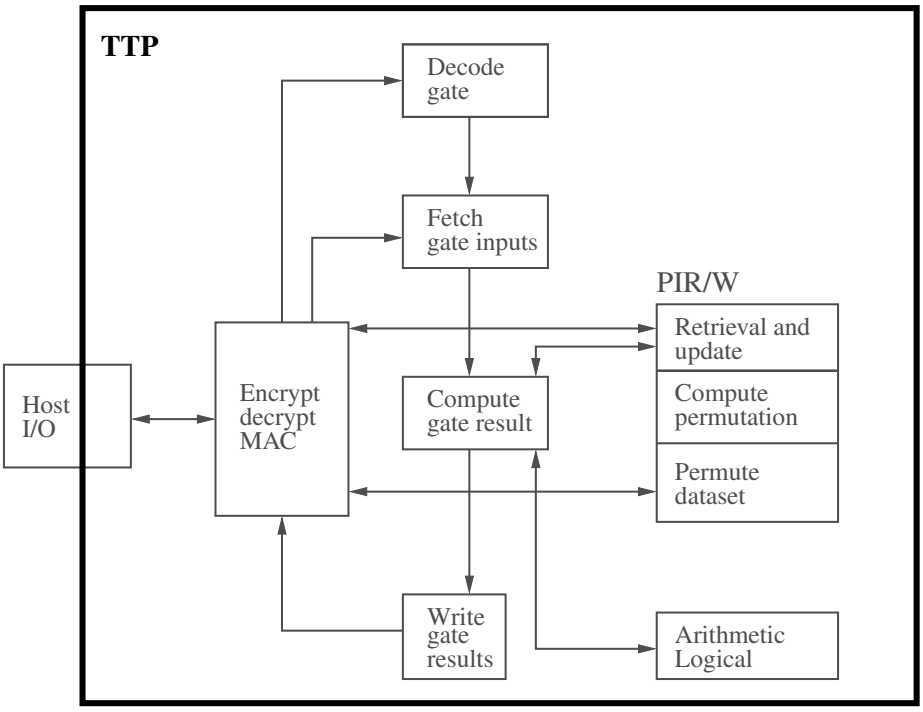


Figure 7.2: Components of the Faerieplay CVM.

7.7 Baseline solution: Oblivious RAM

Oblivious RAM (ORAM) [41] provides an existing technique to achieve general SMC on a tiny TTP, as it enables the TTP to run a program, using a large untrusted RAM, while keeping the view of the RAM (and the resident adversary) oblivious. We have described ORAM in Section 4.3. Here we describe how our approach reduces the overhead imposed by ORAM.

Both approaches use the oblivious indexing algorithm which was introduced with ORAM, and which we developed for our private information retrieval system (Chapter 3). For an N-element array, it imposes an overhead of a factor of $\mathrm{PIRO}(N)$ per access (see Section 3.5.1). The main difference between ORAM and Faerieplay is that we limit oblivious indexing to just where it is needed—indirectly-indexed arrays, while ORAM uses oblivious indexing for every instruction in the program. This is illustrated in Figure 7.3. The remaining elements—code, scalar data and direct arrays—all reside in the circuit and are accessed in the same pattern for all inputs, so the pattern does not need to be hidden. The actual values are of course hidden by encryption.

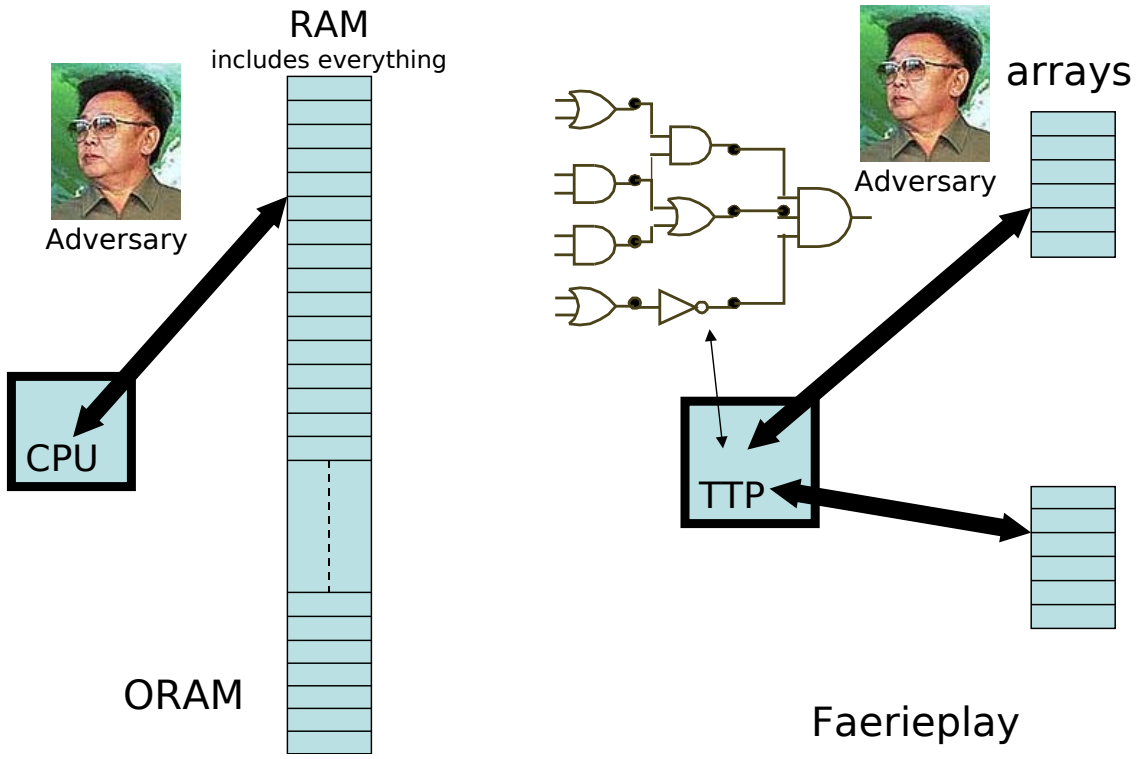


Figure 7.3: Comparison of memory accesses in ORAM and Faerieplay. Thick arrows indicate expensive oblivious access and thin arrows indicate cheap direct access. Faerieplay needs fewer indirect accesses to fewer elements than does ORAM.

The gain in our approach stems from two sources:

- We have fewer expensive memory accesses—only those referring to indirect arrays; and
- Our expensive memory is smaller, hence the overhead for each access is smaller.

More concretely, let N_T be the total number of RAM accesses, N_I be the total number of accesses to indirect arrays. (We observe in our results, Section 10.6, that the actual numbers for ORAM and Faerieplay, in seemingly quite different implementations, are quite close). Then, ORAM incurs a $\mathrm{PIRO}(N)$ overhead (see Section 3.5.1) on $N = N_T$, whereas Faerieplay incurs $\mathrm{PIRO}(N)$ overhead only on $N = N_I$, and $\mathrm{O}(1)$ overhead on $N = N_T - N_I$.

How do N_T and N_I compare? $N_I < N_T/2$, as even if all instructions access indirect arrays, the instructions which cause the accesses still have to be loaded. In Section 10.6 we provide concrete numbers on the Dijkstra example, for graph sizes well beyond those feasible on the 4758 TTP.

Additionally, ORAM cannot naively use a load-store instruction set, as then the intervals between TTP accesses to the host depend on the mix of CPU vs. memory instructions, which may differ in different conditional branches of the program. Observing the gaps from outside the TTP, the adversary could learn information on taken branches. A simple solution is to have all instructions access memory, which carries a considerable efficiency penalty compared to a load-store approach—it introduces many more memory accesses, which are the expensive operation for ORAM. More complicated but potentially more efficient would be to ensure that both branches of a conditional have an identical pattern of CPU and memory instructions.

On the negative side, Faerieplay, as well as other circuit-based approaches, present a somewhat more constrained programming environment than traditional environments like an imperative language compiled to its native RAM machine, as detailed in Appendix A.

Secondly, Faerieplay has to spend time executing both branches of a conditional, even though one will be thrown out. As noted above though, ORAM has complications stemming from needing to hide taken branches too.

We are now finished with the introduction and overview of Faerieplay, and will move on to different details in the next three chapters: a security model and definitions, specifications for the system components, and implementation details and evaluation.

Chapter 8

Faerieplay Security Model

Having introduced Faerieplay in Chapter 7, and before we move on to system specifications (Chapter 9) and our implementation (Chapter 10), in this chapter we develop a security model for Faerieplay. This model is an abstract view of the Faerieplay system, focusing on security properties. The aim of the chapter is to provide a precise statement of the security properties that Faerieplay provides. This security statement will obviously be used by the system's users to understand what assurances they have. Additionally the security model will provide a set of specifications for the implementation which will ensure that the overall Faerieplay security guarantee is met.

For the system users we state the user-level guarantees that Faerieplay provides, mainly to do with hiding user data from the adversary. We also develop a model for the system, which we can use to state and prove more detailed security properties, and thus build up an argument for why the high-level guarantees hold.

For the implementers of the system, we derive specifications which they need to meet in order for the abstract security properties to be satisfiable.

The assurance of the security properties underlying the high-level security guarantee comes in two main ways. Firstly we can infer security properties from the structure of the implementation, eg. the use of arithmetic circuits. Secondly, we can state specifications for the implementation, eg. executing "similar" operations in a fixed amount of time regardless of the inputs. These specifications then provide axioms to build up the high-level security argument.

In this chapter, we start by showing the high-level user view of the Faerieplay system. This requires introducing the *Secure Computation Server* (SCS) which users interact with. Then, we give a preview of the structure of the SCS, which sets up the presentation of our adversary model. We then detail the main underlying mechanisms of Faerieplay: the tiny TTP, and representing the function as a circuit. After this setup, we begin the detailed security definitions, in terms of a *transcript* of data interchange between the TTP and its host. Finally, in Section 8.9 and Section 8.10 we present the Faerieplay security argument, against a passive and an active adversary respectively.

8.1 The user's view

We begin with the view of a single user of the Faerieplay system. Since Faerieplay is an SMC system, the user's view will be a more specific form of the general SMC picture shown in Figure 6.1. Assume for simplicity that we have two users, Agnes and Boris. In general, there could be N users, $U_1 \ldots U_N$. We will focus on Agnes, keeping in mind that the analysis is symmetrical for all the other users.

Agnes has a value $a \in A$. She also has a function $f : A \times B \rightarrow Y \times Z$. She wants to learn the partial value y of $f(a, b)$, where $b \in B$ comes from Boris. We temporarily hold off deeper analysis of "comes from Boris", beyond the intuitive understanding that b is Boris's input for the joint computation. Boris's view is symmetrical: he has a value b and the same function f, and wants to learn the partial value z of $f(a, b)$, where a comes from Agnes. Note that these are just the users' functional expectations, we will address their security requirements shortly.

8.1.1 Secure computation server

In Faerieplay, Agnes will learn the result of the computation by sending a and f to the *Secure Computation Server* (SCS) which carries out the computation securely, and sends the results back. As we elaborate in Section 8.5, the main component of the SCS is a tiny TTP. This setting is illustrated in Figure 8.1.

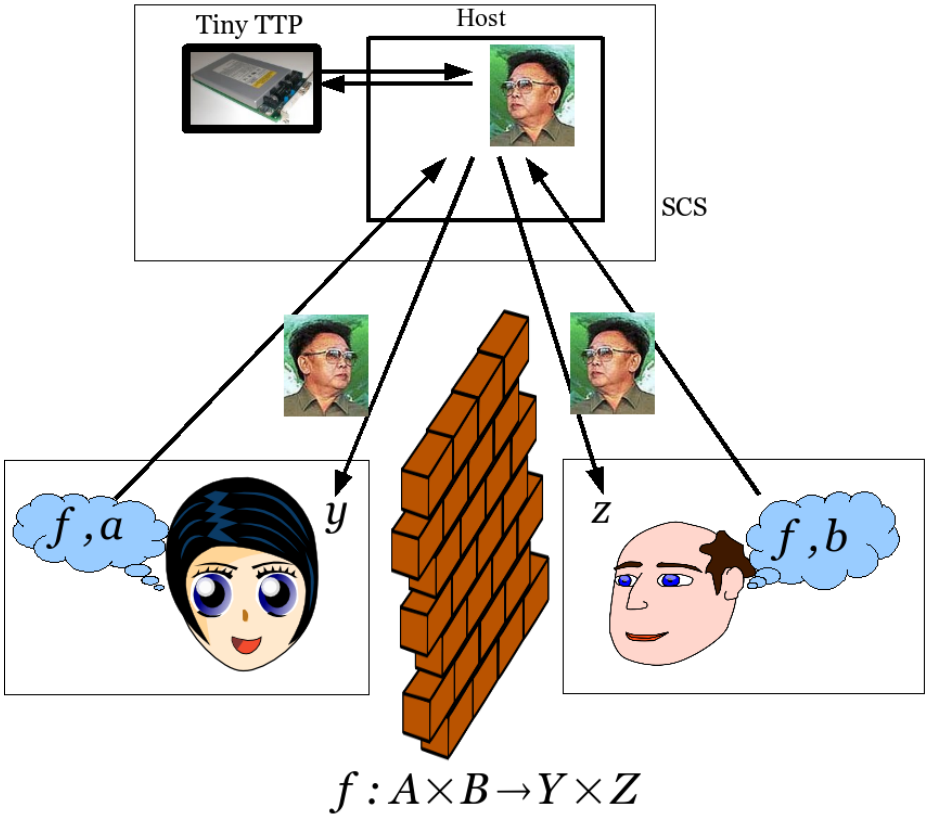


Figure 8.1: The users' abstract view in Faerieplay. Agnes has a, Boris has b, and they want to learn their respective parts of $f(a, b)$ without divulging their respective input data. They accomplish this by having the *Secure Computation Server* (SCS) perform the computation on their inputs. As we detail in Section 8.5, the Faerieplay SCS is based on a tiny TTP. This is a more specific instance of the general SMC model shown in Figure 6.1. The adversary controls the communication as well as the TTP's host.

8.2 Adversary model

Having established the structure of a Faerieplay system, we can specify what capabilities we assume the adversary to have, and what the users' *trusted computing base* (TCB) is (see Section 4.1 for an introduction to TCB). We start with the users' TCB, as that frames the adversary capabilities: system components not in the TCB are controlled by the adversary.

Definition 2 (Faerieplay user TCB) *A Faerieplay user's TCB (say Agnes's) is her own computer, as in the general SMC model, but also the tiny TTP in the SCS. The adversary controls everything else, including peer users.*

Adversary's allowed access

Concretely, we give the adversary Mallory the following abilities:

- Mallory controls all the networks in the system, eg. between Agnes and the SCS. This provision is a standard one, and is part of the formal Yao-Dolev model [32, 33]. It accounts for the presence of unreliable and likely adversarial networks and other third parties.
- Mallory controls the TTP's host in the SCS. The host handles part of the communication from Agnes to the TTP, and also provides data storage services for the TTP.

Adversary's a-priori knowledge

The data security we want our system to provide is with reference to the information available to the adversary a-priori. The system must hide information which was *not* specified to be available to the adversary. In our case, the adversary knows the function f to be computed. Thus, the system does not need to hide f.

Adversary's abilities

In addition to specifying what access the adversary has to system components, the model specifies what the adversary's capabilities are. In developing the security model later in this chapter, we consider all the standard classifications of adversary capabilities which we introduced in Section 4.6: passive vs. active, and unbounded vs. computationally bounded.

- All computation in the TTP is inaccessible to the adversary,
- but the adversary controls the environment outside the TTP.
- The protocol participants can authenticate the TTP, and establish a secure channel to it.
- We do not consider denial of service attacks by the host—dealing with those is orthogonal to ensuring the privacy of participants' data. In Section 8.2.1 we outline how DOS by the host could be mitigated.

How good is the 4758 as a TTP?

We assume that the concrete hardware we use as a TTP is good at what it does:

- Implements its API (OS and standard libraries) in a way which does not expose timing differences with different runtime values. Thus, a correctly written program running on the device will itself be secure against timing attacks.
- Does not present other side channels to the adversary outside, like radiation, acoustic or variations in power usage.

How much of Faerieplay CVM is in the TCB?

Looking at the component breakdown of the CVM in Figure 7.2, we can see that all components except the host I/O one can break some aspect of the system's security guarantee, so they must be in the TCB.

8.2.1 Apropos: denial of service by the host

Since the TTP depends on its host for any external communication, as well as data storage, it cannot prevent the host from denying it those services, and thus preventing a computation from completing. The TTP's provisions to detect tampering by the host (see Section 8.10) give the host another way to prevent successful completion of a computation.

The TTP itself cannot recover from such denial of service (DOS) attacks from its host. It will not let the attacks violate the security properties, but it cannot force the computation to proceed. A solution to DOS has to involve the whole system including the first parties. Depending on the setting of a Faerieplay deployment, the DOS problem could be addressed in different ways.

Commercial SMC service with SLA If a service provider is offering a commercial SMC service using Faerieplay, and if they provide a Service Level Agreement (SLA) as part of that service, then their customers can demand compensation if the service provider does not meet the SLA, eg. by

taking too long for a circuit evaluation, or aborting too frequently. Thus the service provider will have incentive to ensure that their hosts function correctly and efficiently.

Consumer or non-commercial service If the TTP is provided as a commercial service without an SLA (consumer level), or is non-commercial, eg. provided by a public interest organization, some form of a reputation system, with feedback from users, can be used to flag problematic hosts.

We do not address the DOS problem any further in this work.

8.3 Users' security properties

Now that we know the overall structure of a Faerieplay system, and the adversary we have to protect against, we can begin to state the system's security properties. We begin with the properties presented to the users.

8.3.1 Security summary

The security properties Agnes wants are the usual ones of integrity and confidentiality[1]:

correctness Agnes wants to be able to check that her result is actually consistent with $f(a, b)$.

data confidentiality Agnes wants to know that her input a is not revealed to anyone else, beyond what the result of the computation itself reveals. This is the standard security property in secure multiparty computation, which we outlined in Section 6.2.2.

8.3.2 Integrity and correctness

In our TTP-based model, Agnes trusts the TTP to perform the computation correctly. Thus, once she is satisfied that the function f is correct, she trusts that a result she receives from a TTP she has authenticated is correct.

[1] We do not consider *availability* in this thesis, beyond a brief examination of denial of service (DOS) in Section 8.2.1.

In theory, Agnes can verify correctness of the computation without relying on the TTP for this, by requiring that the result be accompanied by a Zero-knowledge (ZK) proof of its correctness, ie. a ZK proof that

$$(y, \_) = f(a, b)$$

(Notation: the underscore is a "don't care" value; in this case, Agnes does not need the proof to address the value of z, which is Boris's result.) A normal proof cannot be presented to Agnes, as that would have to reveal the value of b, which Boris does not want. However, ZK proofs of general properties are expensive, and unlikely to be feasible for large functions.

8.3.3 Data privacy

Having satisfied herself of the correctness of her result y, Agnes would like to be assured that no one she does not trust could have learned anything about her input a, beyond what the answer z tells its intended recipients. More concretely, as part of the joint computation, Boris learns the value of z, and this can in general imply some information about Agnes's a (unless z is constant, or otherwise independent of a). Agnes has to accept this as part of agreeing to compute f together with Boris, but she does not want Boris or anyone else to learn any more than this.

In Faerieplay, the participants have to trust the SCS, and most of our work in providing a security model is to show why this trust is warranted. In the next section we will turn to a view of Faerieplay from the SCS perspective. First we will mention several additional aspects of the system which impinge on the overall security, but are fairly orthogonal to our main concern with providing correctness and privacy for the computation of the function f.

8.3.4 Additional concerns

Authenticating the inputs

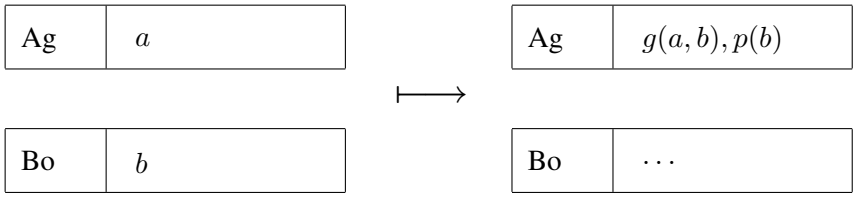
The definition of correctness for Agnes's result y depends on defining correctness for the inputs of the non-Agnes users, in this case b from Boris.

Non-repudiation A Faerieplay computation can provide non-repudiation based on the TTP's assertion about the correctness of the results y and z. Thus for example, if Agnes wanted to prove that she received a result y from a computation with Boris, Boris could not deny this, as the TTP's assertion to Agnes confirms that y was indeed computed with Boris's partial input.

Property checking Agnes could require extra verification of Boris's input b, which convince her that additional properties hold on b. These could be:

- Pure properties of b, which do not depend on anything else. For example, $b > 0$, or b is even. These are obviously dependent on what B (the domain of b) is.
- Asserted properties of y, eg. "Authority A asserts that y is a legitimate widget".

Agnes can encode property checking as part of the function f. For example, using the notation from Section 1.4.1, f could be:



ie. Agnes learns the value $g(a, b)$, but also the truth of predicate p on Boris's input b.

If Agnes wants an external assertion (say from Trent) about b, Boris would have to submit such an assertion as part of his input, and f would include checking the signature on the assertion, confirming that the assertion is on the actual value b, and including some indication of the assertion as part of Agnes's output, for example "Trent asserts that property p holds on Boris's input".

Authenticating the function f

The overall system has to provide for checking that it is computing the function that all users agree on. This can be done in various ways, for example

- Require that all users send f along with their inputs, check that all their fs are literally the same, and abort the computation if not.

- Allow the function (which could be a large program) to be sent by only one user, or obtained from a third party, but require that all users provide signatures on the function with their signing keys, thus stating that they have checked it.

The decision of how to handle function authentication is orthogonal to our main topic of protecting the users' data, and we will not address it further.

8.4 The system designer's view

Having specified the security properties guaranteed to a user of Faerieplay, we will now turn to the designer of the SCS. The SCS has the following interfaces:

- Receive Agnes's a and Boris's b and their function f,
- Compute $(y, z) = f(a, b)$.
- Return y to Agnes and z to Boris.

Again we note that communication between the users and the SCS is implemented via the adversary Mallory, which models adversarial networks and other intermediate parties.

Given this syntactic structure, the SCS needs to be able to assure its users that their correctness and privacy requirements are being met.

8.5 Programmable hardware TTP

In Faerieplay, the main component of the SCS is an embedded device which does the actual computation on plaintext values, and has to be trusted by the users. This device is a trusted third party (TTP) for the users[2]. The overall system may involve other TTPs, eg. for validating user inputs, but we will refer to the computational TTP which is at the heart of the system as *the* TTP.

In this hardware-TTP based approach, users' trust in the whole SCS comes from the users believing that:

[2]We introduced the notion of TTPs in Section 4.5.

User Belief 1 *the hardware TTP is trustworthy against the postulated adversary.*

User Belief 2 *the use of the TTP does not leak information to the adversary.*

Our main aim in this chapter is to demonstrate the latter part, namely to provide assurance that our use of a TTP is correct. We assume the trustworthiness of the TTP itself, as specified next.

8.5.1 TTP capability and security assumptions

The TTP in Faerieplay can provide the following functionality:

- Decrypt ciphertext encrypted for its private decryption key,
- Sign data securely, with its private signing key,
- Generate random bits, which is required for generating cryptographic keys and nonces,
- Execute a general program.

In providing the above functionality, the TTP also provides the following security assurances, which users have to trust.

- The TTP wields exclusive access to its decryption private key, and signing private key.
- The TTP carries out computations in a black-box manner—its host can observe its inputs and outputs, but cannot learn anything about the TTP's internal state during the computation. The one exception is that the host can observe how many computation steps a TTP computation takes. The black-box property requires that the TTP has no side-channels visible to the outside, beside computation time.

  This black-box property allows the TTP to perform general computing on ciphertexts: it can apply any function to a set of ciphertexts by decrypting them, applying the function to the plaintexts, encrypting the results, and returning the resulting ciphertext. Concretely, the TTP can implement a universal (2-argument in this example) function application, such that the inputs and outputs are ciphertexts; we illustrate this capability in Figure 8.2. We assume that a way to represent the function f has been agreed upon.

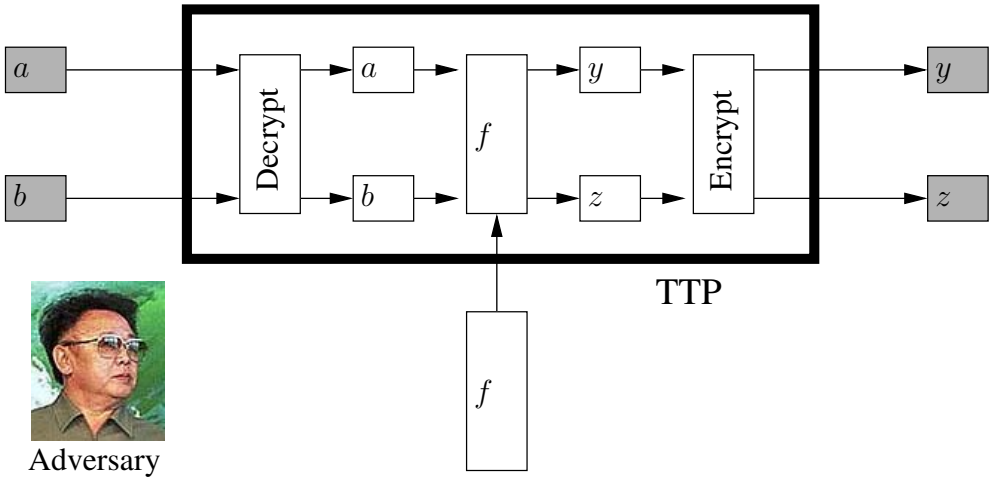


Figure 8.2: Illustration of TTP's ability to perform a computation on ciphertexts, which is not visible to the adversary outside the TTP. Notation: $\boxed{x}_K$ is the ciphertext of plaintext x under key K; the key can be omitted if it is not significant in the context.

Since the computation takes place inside the TTP, the host cannot learn anything about it, and thus cannot learn anything about the plaintexts, beyond what the ciphertexts tell him.

8.5.2 TTP memory limits—tiny TTP

We have described the size limitations of our TTP in Section 7.2.2 on page 98.

8.6 Computing with circuits

As we discussed in Section 7.2.1, Faerieplay represents programs as arithmetic circuits, augmented with special gates for indirect array access. Here we will introduce circuit evaluation to the extent necessary to develop a security model for computing using circuits and a hardware TTP. We will initially focus on "scalar" gates, and postpone the discussion of array gates till Section 8.11.

8.6.1 Faerieplay circuits

We briefly define the circuit evaluation for which we are developing a model. Faerieplay circuits are arithmetic circuits, where the arithmetic is all on fixed-size integers, eg. 32 bits. Each gate has

a small constant number of inputs, mostly 2 or 3, and one output. Each gate input comes from the output of another gate. Apart from standard arithmetic, Faerieplay supports a conditional gate, which is analogous to an electronic multiplexer: the output is either the value of input 1 or of input 2, depending on the value of a control bit which represents the condition. Each gate in the circuit has two main parameters: a static *gate code* and a computed *gate value*.

Gate code The code of each gate describes the gate statically. Given the gate code, a circuit evaluator can look up the input values, and compute the gate's output. The gate code consists of

- the gate operation, eg add, divide, select (conditional)
- input locations

The code for each gate is stored in a host container[3] called $gates$ (being too large to fit in the tiny TTP), which is indexed by gate number. $gates$ is not modified after initialization with the circuit for f.

Gate value The value associated with a gate is maintained in a host container $values$, indexed by gate number. A gate's value is uninitialized until the gate is evaluated, at which point the value is set.

An overview of this setup for TTP-based circuit evaluation is shown in Figure 8.3.

8.6.2 Computation process

Given a programmable tiny TTP, we load it with a circuit evaluation program, the *Circuit Virtual Machine* (CVM). The CVM can compute any function in circuit form: it is *universal*. The CVM can be given a circuit and inputs for that circuit, and it will evaluate the circuit on those inputs, and return the result. This circuit evaluation is done within the constraints of the tiny TTP's resource limits, and needs to satisfy security properties which we will develop below.

[3] A host container is an array stored on the host, which the TTP can access through the host I/O API. See Section 10.3.2 for details

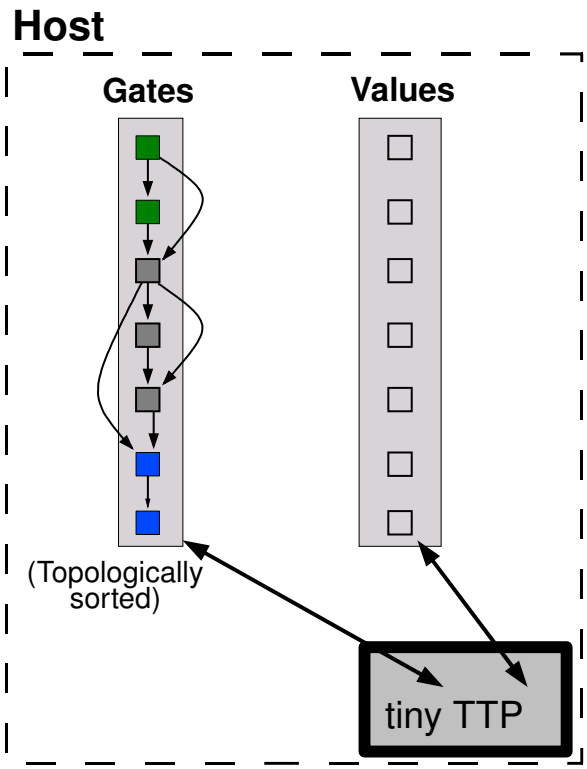


Figure 8.3: Setup for hardware-TTP-based circuit evaluation. The circuit is stored on the host as two arrays, for gate code and gate values. The arrows on 'gates' indicate the flow of data from output of one gate to input of another. The 'gates' array is initialized just once at the start of a computation, while 'values' is populated as each gate is computed.

Circuit evaluation consists of evaluating each circuit gate in an order which respects the data flow between the gates. Recalling that the circuit gates and gate values are stored on the TTP's host (as shown in Figure 8.3), for each gate the TTP needs to

1. Read the gate code from $gates$,
2. Read the gate input values from $values$,
3. Compute the gate's result,
4. Write out the result to $values$.

8.7 Modeling the computation with a transcript

We now have enough of a picture of how the SCS computes functions to begin developing a model for this TTP-based circuit computation.

The TTP executes its program on its data by doing internal computation, and interacting with its host for bulk data storage. As discussed in Section 8.2, the host should be thought of as being under the adversary's control. During the computation, the adversary sees various intermediate data involved in the computation, any of which could potentially provide him with information he should not have[4]. We will model this exchange of information with the adversary using a *transcript*. Note that a transcript is a standard tool used in security modeling.

Definition 3 (Transcript) *The transcript should be a complete record of the data which the TTP exchanges with its host during a computation.*

If some item is not in the transcript, then it should be the case that the host has never had possession of it. In addition to defining completeness for the transcript, this definition places a restriction on the implementation of the TTP's program: any data which are not specified to be in the transcript cannot be sent to the host.

8.7.1 Transcript form

We specify our transcript form aiming to categorize several kinds of data it contains, which will structure the subsequent security discussion. Thus, a Faerieplay transcript T consists of entries of the form

$$(Time, Operation, Data)$$

The components of an entry in T are:.

$Time$ the time of the entry. Times need to be in the transcript, as it is possible for the adversary to deduce information about the computation from the times of various events, if we do not

[4] We consider active or tampering attacks later in Section 8.10

take care to prevent this. For example, if adding two large numbers takes longer than adding two small numbers, observing the duration of an addition operation may tell the adversary something about the operands.

Operation The kind of operation the entry represents, one of a small fixed set of operations, which corresponds to the TTP-host API:

- READ, for reading from a specified container on the host.
- WRITE, for writing to a specified container on the host.
- READGATE, for reading a gate description. READGATE$(index, result) \equiv$ READ$(gates, index, result)$.
- READVALUE, for reading a gate value. READVALUE$(index, result) \equiv$ READ$(values, index, result)$.
- WRITEVALUE, for writing a gate value,

Data has to contain the complete data associated with the operation. For example, a read of an array element from the host needs to include the element index as well as the element contents.

For the security analysis, the use of the data is irrelevant, eg. whether it's an index into an array, or a gate opcode. The only significant thing about entries in T is that they are available to the adversary.

As we shall see soon, most of this data must be protected from the adversary by encryption.

8.8 Security of Faerieplay computation

At the end of a TTP-based computation of $(y, z) = f(a, b)$, Mallory has the complete transcript T for that computation. Note that this will usually include some entry corresponding to the results y and z, as they are communicated to the users via Mallory (see Figure 8.1).

Definition 4 (Security of TTP-based computation) *A TTP-based computation of function f on input (a, b) with resulting transcript T is secure against adversary Mallory if Mallory's knowledge about (a, b) is the same given T as it was before the computation, without knowing T.*

Another formulation of this condition is that T is independent of the input. This means that T can depend only on values other than the input, like the program to compute function f. (We shortly

address computational independence/indistinguishability of ciphertexts (Section 8.9.3) and aborted computations due to tampering (Section 8.10).)

Yet another formulation is that the process which is modeled by T is *oblivious*, with respect to its inputs, and against the postulated adversary. Often the term oblivious is used without these qualifying parameters, but they are always present.

Theorem 1 (Faerieplay computation is secure) *Faerieplay's TTP-based circuit evaluation is secure according to Definition 4, against both passive and active adversaries.*

PROOF In the case of self-contained gates, follows from Lemma 1 and Lemma 2. Lemma 4 covers array gates.

Lemma 1 (Security in terms of structure of the transcript) *A TTP-based computation of function f on input (a, b) with resulting transcript T is secure against a computationally bounded adversary $\mathcal{A}$, if all entries in T are either*

1. *independent of (a, b), or*
2. *encrypted by the TTP with an IND-CPA secure encryption scheme.*

$\mathcal{A}$ can be passive or active.

PROOF First, note that this lemma does not depend on the passive/active nature of the adversary, as long as the complete TTP-adversary interaction is contained in the transcript.
(1) The entries in T which are completely independent of the inputs clearly cannot reveal anything about the inputs.
(2) An encryption scheme which is IND-CPA secure guarantees than its adversary cannot learn anything about a plaintext from a ciphertext. Thus, if Mallory sees the ciphertexts of the encrypted entries of T, he cannot deduce anything about the underlying plaintexts, and thus has no basis to deduce anything about the computation inputs.

Lemma 2 (Security of Faerieplay transcript) *The transcript generated by a Faerieplay computation is secure, according to the definition in Lemma 1. This applies both to a passive and active adversary.*

PROOF In Section 8.9 we establish the detailed structure of the transcript produced by a Faerieplay computation with a passive adversary, and show that each part of it complies with Lemma 1.

In Section 8.10 we present the handling of active attacks, and show how they affect the transcript. Again we show that this augmented transcript complies with Lemma 1.

8.9 The transcript with passive adversary

Here we will see what the transcript looks like for a TTP-based circuit evaluation, which we outlined in Section 8.6. We will then prove Lemma 2 for the case of a passive adversary, by showing that this transcript represents a secure computation, as defined in Lemma 1.

8.9.1 Transcript contents

First we recall that for each gate in the circuit, the TTP (with its universal circuit evaluation program) carries out the following steps (refer to Figure 8.3):

1. Read the gate code from host array $gates$,
2. Read the gate input values from host array $values$,
3. Compute the gate's result,
4. Write out the result to $values$.

To write down the transcript entries resulting from these steps, we need a notation for specifying gate input locations:

Notation 1 $\text{IN}_i^a \stackrel{\text{def}}{=}$ *gate number where input number* a *(out of 2 or 3) of gate* i *is located.*

Now, to compute the value of a 2-input gate number i (eg. an addition gate) we need the sequence of data exchanges and transcript entries shown in Figure 8.4. Note that we have not yet thought about what parts of these entries need to be encrypted, but are focusing on carrying out the gate functionality.

1. $(t_1, \text{READGATE}, (i, gates_i))$,
2. $(t_2, \text{READVALUE}, (\text{IN}_i^1, vals_{\text{IN}_i^1}))$,
3. $(t_2, \text{READVALUE}, (\text{IN}_i^2, vals_{\text{IN}_i^2}))$,
4. $(t_3, \text{WRITEVALUE}, (i, vals_i))$

Figure 8.4: The sequence of data exchanges and transcript entries for evaluating a 2-input gate. This focuses on carrying out the gate functionality, and does not yet consider what needs to be encrypted.

8.9.2 Analysis of transcript contents

We can check if T is independent of the inputs by partitioning T into several components and checking if each of those components is input-independent. With reference to the transcript piece in Figure 8.4, we examine the following partition of T:

Operation times The implementation of the CVM must ensure that computing the result of gates which perform the same operation is done in a fixed amount of time, independent of the gate inputs. For example, an addition gate must take the same amount of time, independent of its inputs. The implementation can take advantage of similar guarantees made by the TTP, eg. that a 32-bit addition on the TTP hardware takes a fixed amount of time. If it is true for each gate individually, then it is also the case that each gate in a circuit is started and completed at a pre-determined time, independent of the circuit inputs. Thus, the $Time$ components of T are determined just by the circuit, provided the implementation ensures a fixed evaluation time for each kind of gate.

input-independent

Sequence of gate evaluations The order in which the evaluator computes the circuit gates can be fixed before evaluation begins, based on the circuit structure. Thus the gate index components (i) of T are independent of the input.

Because each gate's type and input locations are determined by the gate code, the fixed order of gate evaluations also ensures that the sequence of **gate types** ($gates_i$) and **gate input locations** (IN_i^j) are independent of the input.

input-independent

Sequence of operations Each kind of gate has a fixed number of inputs and a single output, and thus contributes a fixed number of READVALUE and WRITEVALUE operations to T. Given the circuit, the whole sequence of operations in T is fixed, and independent of the input.

input-independent

Gate values The gate values $vals_i$ clearly do depend on the input. We are obliged to hide the values from the adversary, as described in the next section. There we will derive what encryption the TTP must apply to its interaction with the host.

input-dependent

8.9.3 Encryption of input-dependent transcript entries: gate values

As we just showed, all components of the transcript T except gate values are independent of the computation input. Thus, we cannot have plaintext gate values in T, or the adversary could obtain full information about the input. Instead, the gate values must be somehow encrypted.

We will use a standard symmetric encryption arrangement—a secure block encryption algorithm with a secure mode of operation, eg. AES with CBC chaining mode and pseudorandom initialization vectors (IVs). More formally, the encryption scheme should satisfy the property of indistinguishability under chosen plaintext attack (IND-CPA) as introduced in Section 4.11.1 on page 78. Recall from Section 4.11.1 that IND-CPA secure encryption will hide everything about the plaintext, including its relationships to other plaintexts; eg. an adversary seeing two ciphertexts cannot tell if they come from the same plaintext.

Note that the security provisions only apply against an adversary who cannot brute-force the block cipher in use, and not against a *computationally unbounded* adversary.

In terms of key management, the TTP could use a single symmetric encryption key to encrypt and decrypt the values for one entire circuit evaluation, and generate a new key for the next evaluation.

Since encryption reveals the size of the plaintext, the Faerieplay implementation has to make

sure that the value of any given gate is the same length regardless of the computation inputs. For example, an integer has to take the same amount of space regardless of its value. If conceptually variable length values need to be accommodated, they would need to be broken down into a fixed number of fixed-length pieces, most likely during compilation into the circuit form.

Henceforth we shall indicate ciphertexts in a transcript entry using our standard color notation, e.g.

$$(t_3, \text{WRITE}, (i, \boxed{vals_i}))$$

to indicate that $vals_i$ is encrypted.

Having established which parts of the transcript need encryption, we can revise the initial form of T given in Figure 8.4 for a gate computation. The transcript with encryption included is shown in Figure 8.5.

1. $(t_1, \text{READGATE}, (i, gates_i))$,
2. $\left(t_2, \text{READVALUE}, \left(\text{IN}_i^1, \boxed{vals_{\text{IN}_i^1}}\right)\right)$,
3. $\left(t_2, \text{READVALUE}, \left(\text{IN}_i^2, \boxed{vals_{\text{IN}_i^2}}\right)\right)$,
4. $\left(t_3, \text{WRITEVALUE}, \left(i, \boxed{vals_i}\right)\right)$

Figure 8.5: Transcript entries for evaluating a two-input gate, with the necessary encryptions. A grey box represents an encrypted value.

8.9.4 Security proof conclusion and implementation requirements

Proof of Lemma 2 with passive adversary All components of the transcript T, except gate values, can be made unconditionally independent of the input with a proper implementation of the circuit evaluation. The TTP needs to encrypt gate values before passing them to the adversary (for storage etc.). This transcript satisfies Lemma 1 in the case of a passive adversary.

The circuit evaluation implementation must satisfy these requirements:

- Computing a given kind of gate takes a fixed amount of time, independent of the gate inputs.

- Gate values are of a fixed length, ie. each gate value's length is determined only by the static gate code. For different gate types, the value lengths can differ.
- The gates of a given circuit are computed in a fixed order.
- Gate values are securely encrypted before being passed to the adversary for storage.

8.10 Security against active adversary

In the last section we proved Lemma 2 in the case of a passive adversary. Here we move on to handling an active adversary. We will analyze the tampering options available to the adversary, and present the countermeasures of the TTP. We will show the effect these countermeasures can have on the transcript, and prove that the transcript still satisfies Lemma 2.

8.10.1 Adversary definition

An active adversary will attempt arbitrary attacks, without regard for the agreed protocol. Active adversaries are also called *tampering adversaries*. The protocol must ensure that tampering attacks cannot violate the required security properties.

Lemma 3 (Concrete capability of active adversary) *Within the Faerieplay circuit and TTP-based model, an adversary can mount an active attack only by returning incorrect gate descriptions and gate values to the TTP.*

PROOF As we specified in Section 8.2, the TTP does not provide any other channels for an active adversary to use.

8.10.2 Value identity

Defining what it means for a value to be correct requires that we define the *identity* of values in a computation. In general this notion varies across the kinds of values used in the computation, but in all cases it has to be unique and efficiently computable:

Definition 5 (Unique and efficiently computable identity) *A value identity is* unique *if no other value in a specified universe of computations has the same identity. It is* efficiently computable *if, given the context in which the value is used, the TTP can determine the value's identity in* constant *time.*[5]

For example, consider the identity of a gate description. The value identity should include the kind of value it is, in this case a gate description. The main part of the identity is the gate number, i_g. Whenever the TTP needs a gate description, it clearly must know the gate number, so computing i_g is a non-issue. However we must also consider the gate descriptions across different computations of the same circuit. We have several choices here:

- Parametrize the identity by a computation number i_c. Different circuit evaluations would have different numbers, thus precluding that the TTP could be confused into accepting a value from another computation.
- Limit the universe of computations we need to consider for identity correctness to just the current computation. This could be done by ensuring that the value legality check only succeeds for values in the current computation, for example by using a fresh MAC key per computation.

In summary, the identity of a gate description could be

$$\langle \text{GATEDESCRIPTION}, i_c, i_g \rangle.$$

8.10.3 Different kinds of tampering

There are two distinct approaches to dealing with active attacks, but they share the requirement that the TTP is able to detect deviations from the protocol, ie. detect when the adversary provides incorrect values. Here we will describe the kinds of tampering that the adversary can undertake.

Different kinds of authenticity and tampering

The adversary can undertake several kinds of tampering, which attempt to violate one of these aspects of authenticity:

[5] Note that this property is the same as the "time-labeled" property in ORAM.

Definition 6 (Integrity) *A* legal *value for a read operation is one which the TTP previously wrote to the host, during a specified universe of computations.*

Definition 7 (Identity) *A* proper *value for a read operation is a legal value which also has the unique identity which the TTP requested for the read.*

A legal value may not have the requested identity. For example, on a request for gate number i_g, the adversary could provide some other gate j_g from the same circuit, which has a correct MAC, and so represents a legal result for that read, but is actually not the correct value as it represent another gate.

Definition 8 (Freshness) *The* correct *value for a read operation is a proper value which is also the last value that the TTP wrote for the requested identity.*

8.10.4 Measures to detect tampering

A tamper-detection strategy must provide a way to check that each of the three correctness properties hold for a particular value obtained from the adversary.

Legality checking Establishing that a value is legal can be accomplished in the standard manner, by attaching a keyed MAC to each value given to the adversary, and then checking this MAC upon receiving the value from the adversary. This ensures that the value received was written out in the past, when the same MAC key was in use.

Identity checking The adversary may attempt a *substitution attack*, by returning a legal value which does not have the requested identity. To detect such an attack, the TTP must *label* each value with its identity, such that upon receiving it at a future time, the TTP can check that it is actually the expected value, and not a replacement (with the correct MAC).

Freshness checking Finally, if the value of a given identity can be written multiple times, the TTP must be able to check that a read from its host returns the latest value written, and not an older value. This is where we have an advantage over general RAM-integrity schemes (eg. [26]) which

cannot in general achieve online constant time verification for memory reads: *all TTP algorithms in Faerieplay write to a given identity only once*. Thus, freshness is not a problem for us, as any legal value is guaranteed to be fresh for its identity.

These measures dictate the following structure for a value v with identity id_v which the TTP wants to store on the host:

$$\text{MAC}_{KM}\left(\text{ENC}_{KE}(v), id_v\right)$$

where KM is the TTP's current MAC key, and KE is the current encryption key. If v does not need to be encrypted, then the application of ENC can be removed. The MAC is needed in all cases where an active adversary is postulated.

Given the ability to uniquely identify and verify values, we can state the two main approaches to maintaining security against an active adversary. The difference between the two is in the action taken by the TTP upon detecting a tampered value.

8.10.5 Tamper response: finish computation with NIL values

In this approach, an incorrect value received from the adversary will cause the TTP to replace the tampered value with a special NIL value, and continue the computation. The circuit evaluation semantics already specify what to do with NIL values, as elaborated in Section 9.7.3. The computation results may themselves become NIL due to the unplanned tampering, but the adversary will not see this. [6]

Proof of Lemma 2 The transcript will remain the same as shown in Figure 8.5, but in the event of a tamper, some of the gate values will be NIL. All values are still encrypted, and so this transcript still satisfies Lemma 2.

[6]Note that the computation principals, Agnes and Boris, may see that tampering has taken place, as they do not normally expect any NIL values in their results. This opens the possibility that the adversary will collude with one of the principals, and thus learn more than allowed by our security requirements. We do not consider such collusion scenarios, but note that we can prevent information leakage in this case by ensuring that *any tampering causes the same outcome* to be visible to the participants, for example that all result values become NIL. Then, all that a cheating Agnes or Boris could learn is that *some* tampering took place, but the cheater would know this already, as they planned it with the adversary. To implement this defense, the TTP would need to explicitly ensure this, for example by setting an internal flag after tampering is detected, and then making sure that all result values are NIL at the end.

8.10.6 Tamper response: abort computation immediately

Here, the TTP aborts the computation immediately upon detecting an incorrect value. This results in the TTP attempting to inform the principals Agnes and Boris of the problem, without providing any other result for the computation. The abort can be indicated with an ABORT action in the transcript T.

An example is tampering of a gate value, which results in T like:

- $(t_1, \text{WRITEVALUE}, (i, \boxed{vals_i}))$

  Tampering: host writes an incorrect value into $vals_i$

- Intermediate computation steps ...

- ...

- $(t_2, \text{READVALUE}, (\text{IN}_j^a, \boxed{\text{tampered}}))$

  where $\text{IN}_j^a = i$, and moreover j is the first gate which reads i's value. The TTP detects the tampering here, as the checksum on the value will fail, or the gate's address will not be the expected one in case of a substitution attack.

- $(t_3, \text{ABORT}, ())$

Generalizing slightly to allow for any kind of tampering attack, not just one carried out when writing a value: after the adversary tampers a value in the computation, the time when this tampering is detected, and generates an ABORT action, is when the TTP first reads the tampered value.

Proof of Lemma 2 The point at which the TTP detects a tamper and generates an ABORT action depends only on the time of the tamper and the structure of the circuit. It is independent of the computation input. Thus, the transcript for such an aborted computation still satisfies Lemma 2.

8.10.7 Choosing a tamper response

From a security viewpoint, both options we have presented provide the desired properties that a successful computation is correct and does not leak any data to the adversary. Some differences

manifest themselves in secondary considerations, which may be important in particular deployment situations.

The first approach, completing the computation with NIL values, may avoid alerting the adversary that their attack has been detected. This may be helpful in prosecuting them successfully.

The second approach, aborting immediately, does not waste resources on a computation which is effectively over as soon as the TTP detects an incorrect value, even if the TTP continues the computation with NIL values.

8.11 Considering array gates

So far we have examined the security requirements and properties of a circuit-based function evaluation, where the circuit is composed of self-contained gates: the TTP can evaluate each gate by reading in the entire input values, computing the result internally, and writing it out.

The Faerieplay system consists mostly of such self-contained operations, however it also makes use of more complex gate evaluation algorithms, for array gates. In particular, the PIR/W algorithm which we use to implement array gates reads and writes data additional to the inputs and outputs of the gate.

This additional interaction with the host does not itself affect the security definition (Definition 4). However, it does require an expansion of Lemma 1, to cover transcript entries which are made oblivious by means other than simple encryption.

Lemma 4 (Security in terms of structure of the transcript, including PIR/W) *A TTP-based computation of function f on input (a, b) with resulting transcript T is secure if all transcript entries are described by one of the conditions in Lemma 1, OR they are part of a PIR/W operation.*

PROOF We have already shown how a transcript of entries which satisfy Lemma 1 indicates a secure computation. Let's now add transcript entries corresponding to PIR/W operations. To show that these entries satisfy our security definition, we appeal to existing proofs of security for various forms of the PIR/W algorithm, eg. by Goldreich/Ostrovsky [41] and Wang et al. [90]. These works show that the respective algorithms ensure that an adversary outside the secure hardware learns

nothing from observing the protocol, in the same setting as Faerieplay.

8.11.1 Unconditional array access

In Figure 8.6 we show an example snippet of the transcript generated by an array read. This is just an illustration to help provide an understanding of how PIR/W execution looks in the transcript. As just mentioned, the security of this algorithm has been proved separately [41, 90].

1 t_1, READGATE, $(i, gates_i)$ ▷ Say that $gates_i$ is: READDYNARRAY$(\mathrm{loc}(A), \mathrm{loc}(idxA))$
▷ Now need to do a PIR operation to read the array value
2 t_2, READVALUE, $(\mathrm{loc}(A), \boxed{A})$, ▷ Note that A does not *need* to be encrypted, as array
▷ references are not sensitive.
3 t_3, READVALUE, $(\mathrm{loc}(idxA), \boxed{idxA})$

▷ And now we start the PIR algorithm.
▷ Assume WLOG that we have not read $A[idxA]$ yet in this working session on A.
4 t_4, READ, $(\mathrm{WORK}(A), 0, \boxed{\cdots})$ ▷ WORK(A) is the working area for array A .

▷ More reads of sequential indices from working area . . .

▷ Let j be the index of next slot in working area of A
5 t_m, READ, $(\mathrm{WORK}(A), j, \boxed{\cdots})$
▷ Done with scan over working area.
6 t_{m+1}, READ, $(\Pi(A), \pi(idxA), \boxed{\cdots})$ ▷ Π(A) is the copy of A permuted with π
▷ Let's call the obtained value $\boxed{\mathrm{v}}$
7 t_{m+2}, WRITE, $(\mathrm{WORK}(A), j+1, \boxed{\mathrm{v}})$

▷ PIR is done, now set array gate value and we're done.
8 t_{m+3}, WRITEVALUE, $(i, \boxed{\mathrm{v}})$

Figure 8.6: Transcript entries for evaluating an array gate: reading the value of $A[idxA]$. The transcript includes the execution of the PIR algorithm with which we implement array gates. Almost all data here is, as before, either fixed or simply encrypted. However on line 6 we have $idxA$, a non-static value, not encrypted but just mapped by π, a secret random permutation.

8.11.2 Array gates in conditionals

The Faerieplay CVM handles an array write gate which is inside a non-selected conditional branch by executing a dummy write. See Section 9.7.2 for details of this technique. In Figure 8.7 we show

the transcript for a disabled array write, and point out how the differences from an enabled array write are all hidden by encryption.

Also note that the array handle parameter to an array gate is not sensitive—the circuit determines which array is read/written by an array gate, and this does not depend on evaluation inputs.

1 t_1, READGATE, $(i, gates_i)$
▷ Say that $gates_i$ is: WRITEDYNARRAY$(\text{loc}(A), \text{loc}(idxA), \text{loc}(newVal), \text{loc}(enabled))$

▷ Now read the parameters.
2 t_2, READVALUE, $(\text{loc}(A), \boxed{A})$, ▷ Note that A does not *need* to be encrypted, as array
▷ references are not sensitive.
3 t_3, READVALUE, $(\text{loc}(enabled), \boxed{\text{FALSE}})$
4 t_4, READVALUE, $(\text{loc}(idxA), \boxed{idxA})$
5 t_5, READVALUE, $(\text{loc}(newVal), \boxed{newVal})$
▷ Since $enabled$=FALSE, $newVal$ is likely to be NIL.

▷ And now we start the PIW algorithm.
▷ Assume WLOG that we have not written (or read) $A[idxA]$ yet in this working session on A.
6 t_6, READ, $(\text{WORK}(A), 0, \boxed{v_0})$ ▷ WORK(A) is the working area for array A.
7 t_7, WRITE, $(\text{WORK}(A), 0, \boxed{v_0})$

▷ More reads and writes of sequential indices from working area . . .

▷ Let j be the index of next slot in working area of A
8 t_m, READ, $(\text{WORK}(A), j, \boxed{v_j})$
9 t_{m+1}, WRITE, $(\text{WORK}(A), j, \boxed{v_j})$
▷ Done with scan over working area.

10 t_{m+2}, READ, $(\Pi(A), \pi(idxA), \boxed{curVal})$ ▷ Π(A) is the copy of A permuted with π
▷ If write was enabled, would be writing $newVal$.
11 t_{m+3}, WRITE, $(\text{WORK}(A), j+1, \boxed{curVal})$

▷ PIW is done.

Figure 8.7: Transcript entries for evaluating an array write gate. This shows a disabled write, but an enabled write is almost identical (The write is disabled if it is inside a non-selected conditional branch.)
The differences from an enabled write are on line 3, where we get the value of the $enabled$ bit, and on line 11 where we write the old value back, instead of writing $newVal$. These values are encrypted in the transcript, so indistinguishable.

Chapter 9

Faerieplay Specifications

9.1 Introduction

Having presented the Faerieplay security specifications, in this chapter we will give the remaining specifications needed by an implementer of a Faerieplay system. The main components which we need to specify are the *source languages* and the *circuit virtual machine* (CVM).

To bring attention to the most interesting and challenging parts of developing this specification, we use a series of **Challenge** paragraphs.

The source languages we support share many features with contemporary imperative programming languages[1]. The main difference from languages targeted at a standard RAM machine is that many program parameters, like array sizes, loop iteration count and recursion termination, *must* be based on values known at compile time.

As discussed earlier, Faerieplay's virtual machine is very different from a RAM machine—it executes an arithmetic circuit by computing the value of each gate in that circuit, starting from *input gates* whose values are provided externally as inputs to the computation.

[1] There is no reason why other types of high-level languages, like functional ones, could not be supported to a similar degree.

9.1.1 Chapter outline

In Section 9.3 and Section 9.4 we specify briefly the two **source languages** currently supported by Faerieplay: a modified version of Fairplay's SFDL, and a subset of C++. We do not go into much detail of the languages' semantics, as they are very similar to existing imperative languages.

Then we turn to the pieces which comprise the execution component of Faerieplay, or the *Circuit Virtual Machine* (CVM). Section 9.6 describes the different kinds of **data** used during execution, and their concrete representation. Section 9.7 specifies the different kinds of **gates** which appear in a Faerieplay circuit, in particular their semantics and representation. Finally Section 9.8 specifies the representation of **inputs and outputs** of a Faerieplay execution.

9.2 High-Level Languages

The first high-level language supported by Faerieplay was Fairplay's Secure Function Definition Language (SFDL). We wanted to be compatible with Fairplay so we could test execution of the same function on both systems.

As Faerieplay progressed, we found that diverging from SFDL in some ways was quite useful; we outline the differences in Section 9.3.5. As we progressed in the complexity of problems we wanted to run on Faerieplay, we concluded that the Fairplay approach has a fundamental problem: it uses a custom language, and thus precludes reuse of existing tools for tasks like program verification and debugging. Thus, we extended our compiler to support a subset of C++ as a source language. In hindsight, we would emphasize the importance of basing experimental languages on existing ones—the cost of losing compatibility with an existing software development toolset is very high once any non-trivial use of the new language is to be made.

9.3 Faerieplay SFDL

From now we will refer to the Faerieplay SFDL simply as SFDL; We will refer to *Fairplay*'s SFDL as *SFDL0*.

9.3.1 Syntax

The syntax of SFDL is fully specified in Appendix B. This specification was generated by the compiler tool we use for parsing—the BNF compiler (`bnfc`, see Section 10.1). The syntax is a mix of C and Pascal. We have included some small sample programs in Appendix C.

9.3.2 Entry point

The entry point to an SFDL program is a specific function—`sfdlmain`. Parameters and return values of `sfdlmain` are the inputs and outputs of the computation respectively. The type of `sfdlmain` is not fixed but reflects the types of the program inputs and outputs.

9.3.3 Type rules

SFDL is strongly typed. Each expression and variable has a type. Variable types must be provided when the variable is declared–in this SFDL is similar to statically typed languages without type inference, like C and Java.

The *Left-hand side* (LHS) and *Right-hand side* (RHS) of an assignment must be of compatible types. The type agreement rules for formal and actual parameters of a function are the same as for an assignment of the actual value to the formal parameter variable.

SFDL does not provide explicit type casting.

9.3.4 Semantics

SFDL semantics are similar to the C-family of imperative languages, with a sprinkling of Pascal: the result of a function is specified by assigning to a variable with the same name as the function. We do not spend any more time on them here.

9.3.5 Differences from SFDL0

Challenge 1 *As we attacked more problems with Faerieplay, we found SFDL0 difficult to work with in some ways, and addressed the problems by adding these features to it.*

- We added support for passing function parameters by reference, so that assignments to a parameter inside a function are visible once it returns. In the absence of this, the programmer has to resort to defining and returning structures containing all the modified variables.
- We enhanced `for` loops to count down as well as up, with the same syntax. The choice depends on which end-point is larger.
- We support a `print` statement which allows a snapshot of program state to be obtained, for debugging (see Section 10.4).
- We define outputs of the computation to be the return value of `sfdlmain`. SFDL0's entry point is the function `main`, and it passes inputs and outputs in specified fields in the structures which are passed to `main`.
- We do not specify a convention for naming parameters to the main function with the names of their owners, eg. Alice and Bob. Binding data owners to parameter names is left for a higher-level Faerieplay component.

9.4 Faerieplay C++

Challenge 2 *We continued to increase the complexity and size of the problems we tackled with Faerieplay: see our electricity power market example (Section 7.5). At this point, our extensions to SFDL0 (Section 9.3.5) still left the development process quite difficult. We realized that we needed a different approach: enable reuse of an existing development toolset.*

A way to do this in Faerieplay (and other systems with experimental execution environments) is to use a high-level language which can be passed both to our system (compiler, circuit machine,

etc.) for specialized execution, *and* to an existing development and execution toolset for auxiliary tasks like debugging and code analysis.

We selected C++ for this job, and we call the resulting language *Faerie C++* (FC++).

We chose C++ because:

- it has a rich enough syntax to provide all the functionality in Faerieplay.
- The C preprocessor allows us some flexibility in specifying our language, and patching up any difference to standard C++. Eg. FC++ *requires* a keyword `var` before a variable declaration. Such keywords make the grammar more regular and a custom parser's job much easier (in this case Faerieplay's parser). When compiling with a standard C++ compiler, we simply need a `#define var` somewhere in the program.
- C++ has a well-developed toolset.
- C++ is one of the most popular programming languages, so it is likely that a prospective Faerieplay programmer will already be familiar with it.

9.4.1 Syntax and semantics

The syntax specification for FC++ is shown in Appendix D, again generated by `bnfc`. We provide several example programs in Appendix E.

The semantics of FC++ are the same as for SFDL, and indistinguishable from those of the C++ subset it covers.

9.4.2 Limitations

We do not aim to implement the complete C++ in Faerieplay, but a simple subset of it, which maps to the functionality provided by Faerieplay SFDL. Some of the differences with standard C++ are:

- The types supported are quite limited, and match those supported by SFDL (see Section 9.3.3).
- Pointers are not included, but reference parameters are available, as with Faerieplay's version of SFDL. The syntax is as in C++.

- Only `for` loops are supported, with a limited structure: only one initialization and update statement, and as with SFDL, the loop termination condition must be determined by static values. The update statement must also reference only static values, to allow compile-time loop unrolling. Note that the loop counter becomes a series of static values after unrolling, so the loop update statement can refer to the loop counter.
- Defining classes is not supported, but `structs` are supported.

9.4.3 Integration of Faerieplay and C++ toolsets

This is a summary of how a programmer, Peggy, would use both the Faerieplay and C++ toolsets to develop, debug and securely run an FC++ program.

First, Peggy writes FC++ code whose entry point is function `sfdlmain`. `sfdlmain` can have any parameters and return values. The Faerieplay compiler generates some boilerplate C++ code, to enable compiling and running the program with a C++ compiler and toolset. This helper code is just a `main` function customized for the program input and output structures. This generated `main` function:

1. Interprets inputs to the program. These can come either as (string) parameters to `main`, or through a specified API which accepts input from the user in some manner, eg. through the standard input.
2. With the input data, prepares parameters for `sfdlmain`.
3. Calls `sfdlmain`, and collects its return value.
4. Returns the output to the outside, eg. by printing it to standard output in a specified encoding.

Then Peggy compiles her program, together with the generated helper file, using a standard compiler like `g++`. She can debug this program using debuggers available in the C++ toolset. She can also perform other correctness verification, like information-flow analysis. At this point, she can be confident that her code is correct for the desired functionality. Finally, she compiles her program with the Faerieplay compiler to obtain a circuit, and runs it with the CVM to obtain the Faerieplay security properties.

9.5 Circuit and Circuit Virtual Machine

In these sections we present the specifications of the Faerieplay *Circuit Virtual Machine* (CVM), and also the executable format for the CVM, which specifies the format of circuits given to the CVM to execute. We refer to the specification of the circuit as the *circuit format* (CF).

As in the usual program execution setting, we have a "machine" which executes a circuit—the CVM. It is currently implemented in software (hence virtual), but in principle there is no reason not to implement it in hardware. After all there have been many unusual machines built before, like various Lisp machines [52] and hardware Java machines [16].

A circuit is conceptually a directed acyclic graph (DAG), with vertexes corresponding to gates of several kinds, and edges corresponding to data flow from gate outputs to subsequent gate inputs. We will next outline the data which flows along the circuit wires of a Faerieplay circuit, and then give the detailed specification of circuit gates.

Preliminaries

The circuit diagrams in this section are all produced by uDraw(Graph)[2], which displays interactive visualizations of graphs, based on a textual representation of the graph. The Faerieplay compiler can produce graph files for uDraw(Graph). For this exposition, we added some embellishments by hand to the uDrawGraph output.

In the circuit diagrams, we show input and output gates (ie. gates which represent input values and output values) with arrow boxes pointing into and out of the circuit respectively. A simple example is shown in Figure 9.1.

9.6 Data

The CVM supports several kinds of data on the circuit data wires (edges). First we will describe the different kinds of data types supported by the CVM, and then specify the bit format used to

[2] http://www.informatik.uni-bremen.de/uDrawGraph/en/uDrawGraph/

```
// -*- c -*-
// simple program to illustrate
// our circuit diagrams.
program HelloAdd
{
    type Word = Int<32>;

    function Word
    sfdlmain (Word a, Word b)
    {
        sfdlmain = (a + b) / 2;
    }

}
```

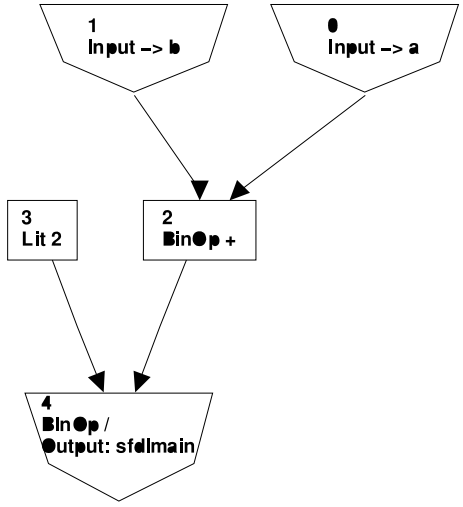


Figure 9.1: Simple example of SFDL code and compiled circuit, to illustrate the circuit notation.

represent them.

Scalars

The most basic data types is an integer scalar, which is defined to be the same as a word on the underlying machine where the circuit virtual machine is run, eg. 32 bits on a 32-bit machine, or 64 bits on a 64 bit machine. In future work, we could decouple the word size specified by the CVM from the the word size of the native machine. This would increase implementation complexity, but improve portability of Faerieplay code.

Array references

Indirect arrays are represented in the CF as opaque references, with the same physical representation as scalars. The circuit machine can implement these references as it chooses, provided that it satisfies the required obliviousness properties, which were introduced in Section 8.8. The CVM will have to maintain the actual array as well as any extra needed state (like a permuted version of the array) on the host, and will probably use the array reference as a pointer to the correct actual array to operate on. We'll refer to the state (on the host and in the TTP) associated with an array as the array representation.

The CF specifies an important property of array references, which enables the circuit machine to implement array accesses efficiently: array references are *linear*, ie. they are used only once. Each array gate has as part of its result a new array reference, and a subsequent operation on that array must use the new reference.

This rule allows the CVM to perform array operations in place, without concern that a subsequent operation will expect the old array representation to be preserved. This license is significant, as an array read or write could trigger a full re-permutation of the array representation, and the CVM can safely perform this without worrying about preserving the previous state of the permuted array.

In contrast, there is no requirement that a scalar value be linear—a single gate's scalar value can be used by many subsequent gates, for example if multiple expressions refer to a single variable. This is safe as a gate's scalar value never changes after it is set.

Structures

Challenge 3 (Supporting structures efficiently) *The CF supports structures of scalars and array references. At a first glance, it seems we could support just scalars and array references in the CF, and have the compiler map from structures in the code. For example, an array of structures would have to be compiled to multiple "parallel" arrays, one for each scalar field in the structure. However, this would be inefficient, as each array access takes more than constant time, and (eg.) reading an element out of an array of structures would have to translate to multiple array reads, one to each of the components arrays.*

The CF supports structures as first-class values, and provides a construct to extract fields from them—the SLICER gate, described in Section 9.7.3.

Array-read gates produce complete structure values. Fields are extracted by SLICER gates if needed by subsequent gates. Array write gates take a parameter which can limit the write to just part of a structure (usually one field); a whole structure value can also be written to an array. See the definition of the READDYNARRAY and WRITEDYNARRAY gates in Section 9.7.3 for the details.

NIL

Challenge 4 (Error handling in the SMC setting) *What does the CVM do in the case of invalid gate inputs, like a zero-divisor, or an out-of-range array index? Since circuit evaluation should look the same outside the TTP, independent of inputs, terminating execution is not an acceptable error-handling strategy, as it can tell the adversary when some particular event occurs, eg. when some value is zero*[3].

Thus, all invalid operations succeed, but return an error value, NIL. All gate operations are defined on normal values as well as on NIL, as described in Section 9.7.3.

Note that the security model requires that all values sent to the host (and thus visible to the adversary) be securely encrypted, and thus the adversary cannot tell if any given value is NILor not.

9.6.1 Data representation

Here we specify the bit formats of the data types which the CVM supports. This is important as the input and output values for each gate computation are stored on the host, and must be marshaled for the SCop-host communication and for storage.

NIL We start with the NIL value, as any value in a circuit execution can be NIL, as well as a normal non-NIL. Whether a value is NIL or not is indicated by one bit in a *tag byte* which is prepended to the marshaled form of the value itself. The first (low-order) bit of the tag byte determines if a value is NIL or not. If the bit is 0, the value is NIL, and the actual value bits are ignored. Otherwise, the value is not NIL, but is equal to the value contained in the bytes after the tag byte.

Scalar Scalar integers are simply represented in the same way as the underlying machine represents integers natively, eg. as 32 bits in little-endian format on x86 processors. Future work can make this representation more portable, eg. using the XDR standard for representing data.

[3] On the other hand, handling a tamper attempt by the adversary (on the host) is best done by aborting immediately, as described in Section 8.10

Array references Array references are represented identically to scalars. The array management component of the CVM is responsible for instantiating array references, and knows how to interpret them. They are opaque to the rest of the CVM.

Structure values A structure is marshaled as the (recursive) concatenation of its marshaled component fields, without any additional markup. A structure value has its own tag byte, which may mark it as NIL. Its fields also all have individual tag bytes, as they may be NIL-valued.

9.7 Gates

Gates are the circuit equivalent of program instructions. In this section we will start with an outline of Faerieplay circuit gates. Then we will digress to elaborate on conditional expressions, which require special treatment in some cases. Then we present a detailed semantics of the different kinds of gates. Finally we explain how user input data is supplied to the CVM, or, how INPUT gates obtain their values before circuit evaluation.

9.7.1 Outline

Gates are described by several parameters:

number All gates are numbered; the numbers are used for identification in general, and specifically to locate the input gates for a gate in execution.

operation The operation that the gate performs, as described in the next section.

inputs Gate numbers where the inputs for this gate are.

type The type of the gate's output. The compiler maintains detailed type information for each gate, but the executable format only specifies simplified type information:

- `scalar`: a word-size value representing an integer or a boolean.
- `array <length> <element_size>`: a reference to an array of `length` elements, each of `element_size` bytes. Note that since elements can be structures, the

element size of an array can be more than the size of a scalar.

Only a few gates need type information—INITDYNARRAY, INPUT, and Output gates. For all other gates, the type of their output is either not required to be known from the gate itself, or is determined by the gate operation, and does not need to be specified.

flags Zero or more gate flags. Some are compiler-internal, others are part of the CF, eg. to tell which gates carry a circuit output value.

comment A string saying something about the gate, eg. the name of the variable it represents. The comment is important for input and output gates, so the evaluator can specify what variables it is reading or producing. For other gates, the comment is just debugging information.

9.7.2 Conditionals

Evaluating an (unclocked) hardware circuit involves evaluating every gate once, whether its value is used in the end or not. For example, a multiplexer could discard values produced by earlier gates, but those gates produce their values anyway. Faerieplay follows the same principle for most gates—all are evaluated, even if their result is subsequently discarded, in our case by a SELECTOR gate. For Faerieplay, this behavior is mandated by the requirement to make every execution look the same to the adversary, not because the CVM could not compute the circuit more efficiently. The CVM could just compute taken branches (and thus reduce its work), but this would provide information to the adversary.

Since both branches of a conditional expression are always evaluated, the evaluator must expect illegal expressions in the not-taken branches, like division by zero or out-of-range array access. Such gates are evaluated as always, but yield a NIL value. If the code is correct, NIL values will be un-selected by a subsequent `select` gate. An example of a conditional division if the divisor is non-zero is shown in Figure 9.2.

```
function
sfdlmain (Word x,
          Word y)
{
  if (x != 0) {
      sfdlmain = y/x;
  }
  else {
      sfdlmain = y;
  }
}
```

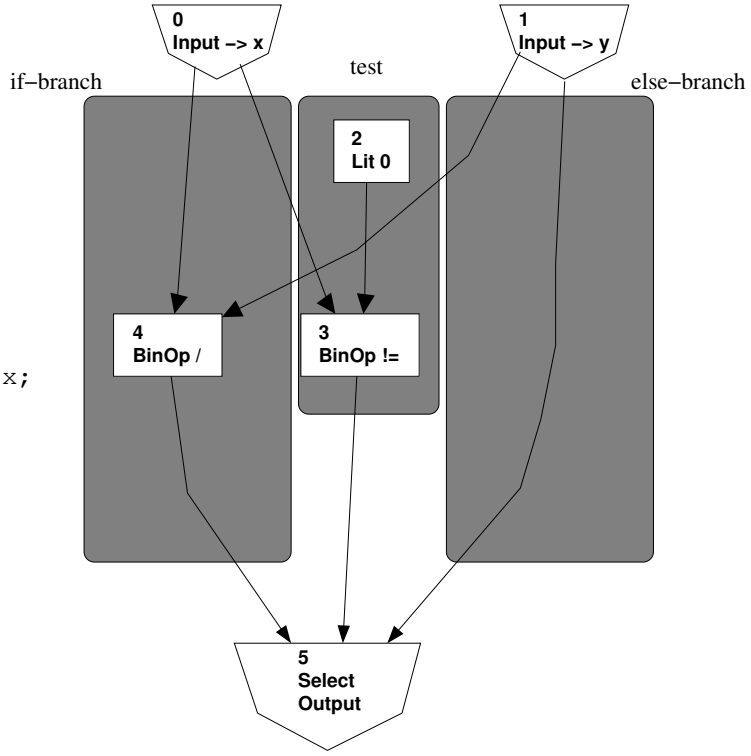


Figure 9.2: SFDL code and compiled circuit for a conditional statement. The division in the true branch is performed even if the divisor is zero, and the branch was not taken. In this case however the resulting NIL value is discarded by the SELECT gate.

Arrays in conditionals

Challenge 5 *We could use the same approach for arrays as for scalars: always carry out the reads and updates and keep all copies until the end of the conditional section, when we can discard the un-selected copies and continue with the selected ones. But doing this efficiently is complicated, and has runtime overhead (a factor of b with respect to the existing PIR/W overhead, where b is the branching depth—the number of active conditional branches).*

The CF specifies a different treatment for arrays inside conditionals, which saves considerable work for the CVM. Array gates are supplied with an additional "gate enable" input, a boolean which specifies whether the gate is enabled or not, depending on the enclosing conditional values. Then during execution, only enabled array accesses modify the array, while disabled accesses perform a dummy operation. Using the PIR/W algorithm for array update, the CVM can easily make a dummy write look indistinguishable from a real write. The time cost of an enabled and disabled array access

must of course be the same, or they would look different to the adversary.

An example of array updates inside a conditional is shown in Figure 9.3.

Arrays are thus not selected on at the end of a conditional block, as the selection has been implicit during the execution of the branches.

We decided not to use the gate-enable approach for scalar gates, but stick to SELECT gates to implement conditionals. The main reason is that scalar SELECT gates impose no more overhead than maintaining an enable bit inside conditionals, and they provide a more modular approach to conditional evaluation than do gates with an enable input.

9.7.3 Gate semantics

We draw a distinction between the *functional semantics* of gates, which specify what values the gates compute, and the *security semantics*, which specify what security properties the computation should provide. Here we give the functional semantics of Faerieplay circuit gates. The security semantics are given in Chapter 8.

Note that the semantics presented here would be highly informal to a practitioner of language design, but complete formalization is out of scope for this work.

Functional Semantics

In Table 9.1 we describe the (slightly) formal semantics of the main gates comprising our circuit format: BIN, UN, LIT, READDYNARRAY, WRITEDYNARRAY, and SELECT.

Mostly the gates semantics are obvious, but the possibility of NIL values (see Section 9.6) almost everywhere complicates the picture.

Challenge 6 (Defining semantics on NIL-values) *All gates can get a* NIL *input, and the semantics must specify how to handle it. In a correct program, none of the output gates will produce a* NIL *value. However, inside a non-selected conditional branch,* NIL *values will occur normally. Finally, the circuit equivalent of* shortcutting *boolean operations (*AND, OR*) is implemented by appropriate definitions of those operations on* NIL *values.*

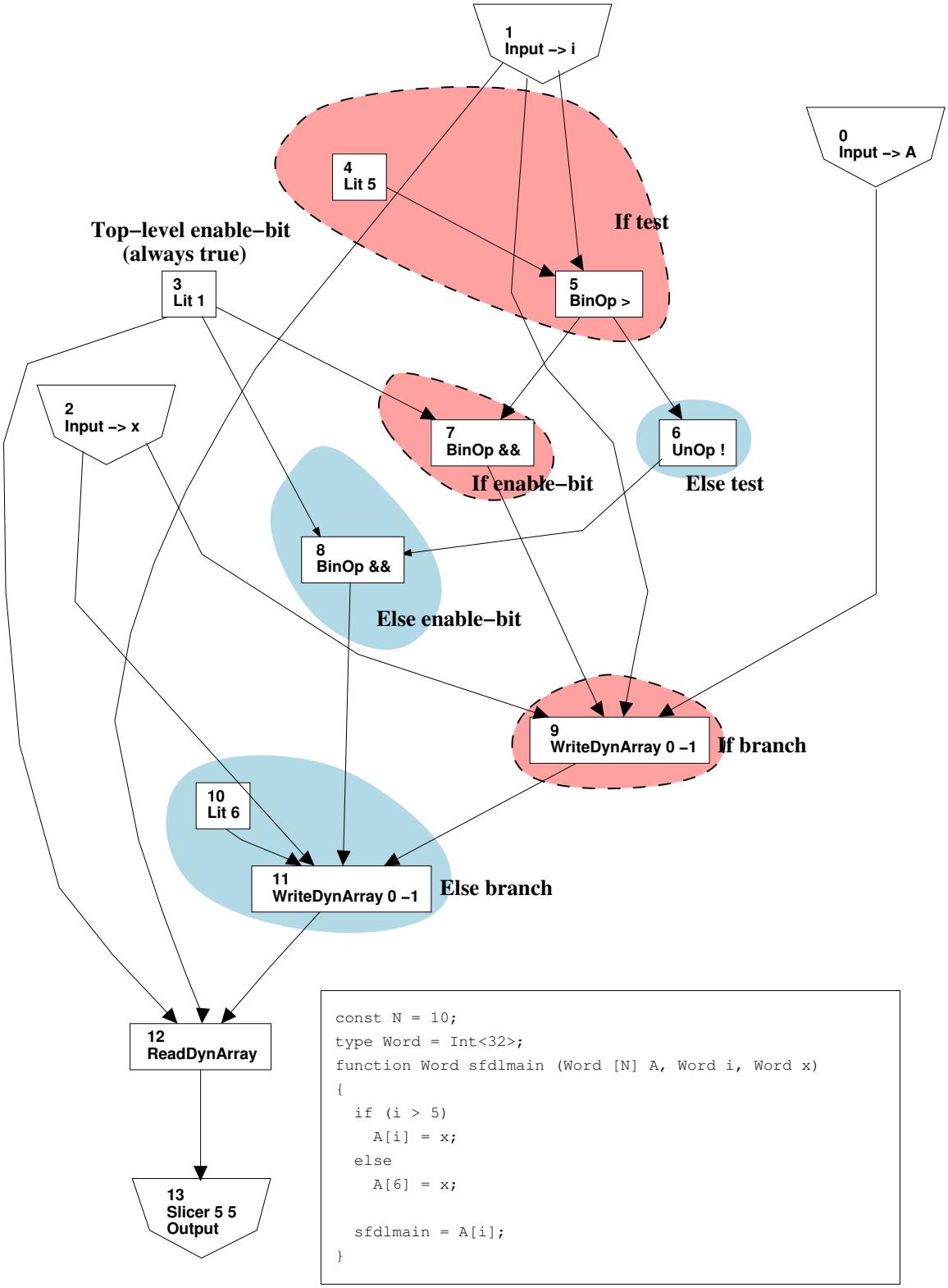


Figure 9.3: Conditional array access code. Highlight notation:
- gates of the true-branch: pink with a dashed border,
- gates of the else-branch: blue with no border.
Gates 7 and 8 respectively provide the enable bits for the two array write gates. Gate 3 (literal 1 or true) represents the conditional value in the enclosing conditional block, which in this case is the top-level “block”, which is active as expected. This value AND the current conditional block value provide the enable bit value for array updates in the current block.

Gate	Output		Comment
BIN[$\oplus$](x, y)	NIL	**if** $x =$ NIL or $y =$ NIL	Binary operator; Note that we propagate NIL generously; one could argue for (eg.) BIN[EQ](NIL, NIL) $\rightarrow$ TRUE. Our current approach is defensive—less likely that two errors will produce a valid (non-NIL) result and this remain undetected.
	$x \oplus y$	otherwise	
	Except		
BIN[AND](x, y)	TRUE	**if** $x =$ TRUE and $y =$ TRUE	
	FALSE	**if** $x =$ FALSE or $y =$ FALSE	This implements shortcutting AND. Eg. consider `x != 0 && y/x > 1`. If x is 0, this should be FALSE even though the RHS is NIL due to division by zero. Note that the parameters to `&&` could be reversed, and the same would apply.
	NIL	otherwise	ie. one TRUE and one NIL.
BIN[OR](x, y)	TRUE	**if** $x =$ TRUE or $y =$ TRUE	This implements shortcutting OR. Eg. consider `x == 0 || y/x > 1`. If x is 0, this should be TRUE even though the RHS is NIL due to division by zero.
	FALSE	**if** $x =$ FALSE and $y =$ FALSE	
	NIL	otherwise	ie. one FALSE and one NIL.
UN$[op](x)$	NIL	**if** $x =$ NIL	Unary operator
	‘op‘x	otherwise	
LIT$[i]$	i		A literal scalar.
SELECT(sel, t, f)	t	**if** $sel =$ TRUE	
	f	**if** $sel =$ FALSE	
	NIL	**if** $sel =$ NIL	A NIL selector should only happen in an un-selected conditional branch, unless the program has an error.
READDYNARRAY(A, i)	$(A'$, NIL$)$	**if** $i =$ NIL	
	$(A'$, NIL$)$	**if** $i \geq length[A]$	$A'[i] = A[i], \forall i$
	$(A', A[i])$	otherwise	
WRITEDYNARRAY$[slice](A, i, x, e)$	A'	**if** $e =$ FALSE	e is an enable bit, set to FALSE if gate is inside a non-selected branch.
	A'	**if** $i =$ NIL	$A'[i] = A[i], \forall i$
	A'	**if** $i \geq length[A]$	$$\forall j, A''[j] \mid slice = \begin{cases} x, & \text{if } j = i \\ A[j] \mid slice, & \text{otherwise} \end{cases}$$
	A''	otherwise	$slice$ specifies which (contiguous) part of the array element to update. Used in case of updating part of a structure. **Notation:** $x \mid slice$ is the part of x specified by $slice$, eg. one field from a structure.

Table 9.1: Gate semantics. A gate's static parameters are shown in square brackets (they can be considered part of the opcode), and the runtime parameters (which are obtained from earlier gates, with addresses specified by this gate's *inputs* parameter), are in parentheses.

The remaining gates have more of a support role, and thus do not require a formal definition:

INPUT A gate whose value is provided as an input to the circuit. The CVM populates INPUT gates with the input values provided by the players before circuit evaluation (see Section 7.6.4). Let V_g be the (output) value of gate g (this is the value which subsequent gates will see if they need an input from g). For an input gate g_{IN}:

- for a scalar value i, the CVM directly sets $V_{g_{\text{IN}}} \leftarrow i$.
- for an array input A, the CVM loads A into a new *array representation* S_A (which will be largely on the host, see Section 9.6), and sets the gate value to a reference to S_A: $V_{g_{\text{IN}}} \leftarrow \text{ref}(S_A)$.

The format in which users provide inputs data to the CVM is described in Section 9.8.

SLICER A SLICER gate extracts a subset of a tuple of values, which implements reading a field of a structure. The output of array read gates consists of an array reference as well as the array element value. To preserve simplicity, the CF specifies that when a gate reads an input from another gate, it reads the whole value, even if it is a structure of multiple parts. But, how does a gate get access to only a part of a structure output produced by a previous gate? This is facilitated by the SLICER gate, which has a static substring specification parameter, and extracts that substring of its input. See Figure 9.4 for an example of code and circuit utilizing SLICER gates.

PRINT This gate has a static prompt string, and inputs from an arbitrary number of gates. When executed, it prints out the values of all its inputs. It is intended to help with debugging SFDL code; see Section 10.4 for more details.

INITDYNARRAY This gate tells the CVM to prepare a blank array of specified length and element size, and has the array reference as its output.

In addition to the normal gate operation, additional information is specified by various gate *flags*:

Output This is an output gate. The gate's comment specifies the name of the output. The mapping of named data to data owners for the outermost computation is specified outside the CVM; the CVM is just concerned with securely computing named outputs from named inputs. The complete setting which includes the computation participants and their interaction with the Faerieplay computation system is described in Section 7.6.4.

```
// -*- c -*-

program SimplestSlicer
{

  const WordSize = 32;
  const N = 10;

  type Word = Int<WordSize'>;

  type RetType = struct { Word[N] A,
                          Word x
                          };

  function
  RetType sfdlmain (Word [N] A,
                    Word i)
  {
    var Word x;

    x = A[i];
    A[i*2] = x;

    sfdlmain.x = x;
    sfdlmain.A = A;
  }

}
```

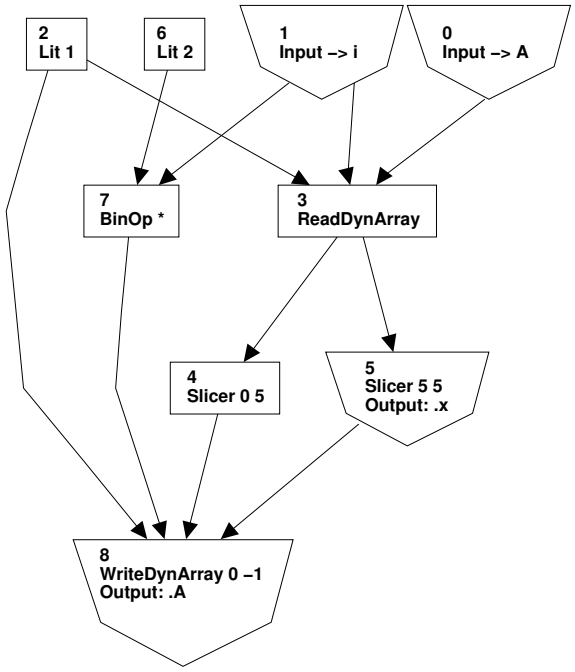


Figure 9.4: A sample of code and circuit which utilize SLICER gates. The READDYNARRAY gate, gate 3, produces two values: the integer at `A[i]`, and a new array pointer. Both values are represented with 5 bytes—4 bytes for a 32-bit integer value, and 1 byte as a NIL-indicator (see Section 9.6). These are concatenated by the CVM into a 10 byte value. The two subsequent SLICER gates extract each value: gate 4 extracts the new array pointer at bytes 0 through 4, and gate 5 extracts the integer at bytes 5 through 9. Both values are used by the WRITEDYNARRAY gate, as the array and new value respectively. Note that the first input to a WRITEDYNARRAY and READDYNARRAY is the enable bit, set to 1 or TRUE here.

9.7.4 Executable gate format

The design of the executable form of the circuit, which is read and executed by the CVM, is strongly influenced by the desire to minimize the complexity of the CVM.

Most significantly, the executable form of the circuit is a topologically-sorted list of gates, numbered with all the integers from 0 to $|Circ|$. In order to simplify the compiler's job, the gates do not have to be numbered in the topological order they are stored in. Execution consists of evaluating the gates one by one in the order in which they are stored, which makes the execution control logic very simple.

In the executable circuit format, circuit edges, which represent flow of data from one gate's output to another gate's input, are specified at the target gate, as a list of gate numbers where that gate's inputs come from.

Executable gate format specification

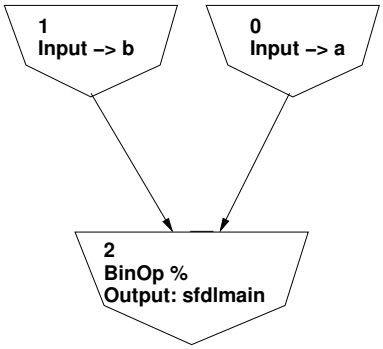
Our current format for the executable circuit is textual and very simple, to enable easy inspection of a circuit file, and to keep the CVM's job of parsing gates very simple. Each important field of a gate is represented as a line, and gates are demarcated by blank lines. The fields/lines currently are:

1. The gate number
2. The gate's flags, as a comma-separated list, or `noflags` if the gate has no flags.
3. The gate's type. Currently the options are:
   - `scalar`
   - `array` <elem size> <length>
4. The gate's operation, eg. SELECT, together with the static operation parameters, eg.
   - `BinOp` +
   - `Lit` 24
5. The gate's input gates, as a space-separated list of (gate) numbers, or `nosrc` if the gate has no runtime inputs (eg. a LIT gate).
6. A free-form comment, which we use to show the variable name, from the high-level source, whose value is at the gate. `nocomm` if the gate has no comment.

In Figure 9.5 we show an example of code, circuit and executable circuit for a very simple program.

```
program Mod {
  type Word = Int<32>;

  function Word
  sfdlmain (Word a, Word b) {
    sfdlmain = a % b;
  }
}
```




```
1                   # gate number
noflags             # flags (none)
scalar              # gate type
Input               # gate op
nosrc               # input gates (none)
b

0
noflags
scalar
Input
nosrc
a

2
Output Terminal     # flag: output gate
scalar
BinOp %             # op with a static param
0 1                 # input gate numbers
sfdlmain            # comment: name of variable
```

Figure 9.5: Executable circuit example: code, circuit visualization, and commented executable circuit: the comments after a '#' character do not appear in an actual circuit file.

Future work would be to design a compact (probably non-textual) encoding for the circuit, in the spirit of standard machine language formats, like MIPS32 or x86. Some aspects of our CF would make it difficult to represent all gates in 32 bits, for example the ability of a WRITEDYNARRAY gate to take inputs from multiple gates. However, the majority of gates could fit into 32 bits, and the remainder would not need much more space.

9.8 Input and output

Challenge 7 (Representing and mapping inputs) *Since the CVM is a constrained environment, and does not support general Input/Output, we need to specify how it can obtain computation inputs and return outputs. Both can be arbitrarily complex.*

The CF specifies that inputs to the circuit are provided as values of INPUT gates, while outputs are found as values of the gates with the OUTPUT flag present. Both of these gates have an annotation which specifies the name of the input/output.

As part of the computation of a circuit on input data, and before the main circuit computation begins, the CVM must somehow obtain the input values, and write them as values of the corresponding INPUT gates. Faerieplay has two specifications related to input values:

- How the input values are represented, keeping in mind that there may be multiple inputs, each of which can be not only scalars, but also any mixture of arrays and structures.
- How the CVM can map from input names to the actual values.

Data representation using CSON

To answer the first question—how do we represent the various kinds of values we want to present as inputs to the CVM—we have developed a simple textual format for representing data. It is similar to the JSON format in which the Javascript language can represent value literals, but is a little cleaner. If the C language could express arbitrary literals, this is probably how it would do it. Given these influences, we call the format CSON.

CSON can express values which are scalars, tuples with named fields, and ordered lists of other values. Lists are delimited with `[]`, and tuples are delimited with `{}`. The top-level value must be one or more tuples, where each tuple is delimited by `{}` as usual, but they are separated not by commas but by an empty space, ie. optional whitespace or comments. Logically, the top-level is one tuple, but we provide more flexibility to allow concatenating different parts of a complex value from different sources. Here is a simple example:

```
{ a = 1,
  b = 2,
  c = [ {x=1, y=2},
        {x=11, y=12}
      ],
  d = { a = 23, b = 25 }
}
{
  xs = [1,2,3]
}
```

The value expressed is presented as two tuples, though logically it is just one. The logical tuple has fields a and b which are scalars, c is a list of tuples, d is a tuple, and xs is a list of scalars.

The full grammar of CSON is given in Appendix F, again generated by BNFC (the same parsing tool we use in the Faerieplay compiler).

Input data example

Consider a program which computes whether a point is inside a polygon defined by a set of points. We could have the SFDL definitions:

```
const WordSize = 32;
// how many points on the polygon?
const MAX_POLY_SIZE = 63;
type Word = Int<WordSize>;
type Point = struct { Word x, Word y };
```

And the main function would have the following signature:

```
function Boolean sfdlmain (Point p, Point[MAX_POLY_SIZE] poly)
```

The Faerieplay compiler would generate three INPUT gates, annotated with the variable names `poly` for the array reference, and `p.x` and `p.y` for the scalar `Point`. Equivalently it could gener-

ate just one INPUT gate for p. An example input for this program, written in CSON, could be:

```
{
 poly = [
          {x=0, y=0},
          {x=-21, y=83},
          // ...
          {y=10, x=-23}
         ]
}
{
 p = { x=12, y=34 }
}
```

For each of the INPUT gates listed above, the CVM extracts the corresponding value from this piece of CSON.

It is up to the broader SMC system, which we call the *Secure Computation Server* (SCS) in Section 8.4, to map inputs and outputs to their owners, and ensure that the communication with those owners occurs securely.

9.9 Succinct representation of loops

Challenge 8 *In Fairplay, and for combinatorial circuits in general, loops are translated by unrolling them entirely. However, this tends to result in very large circuits. This can make compilation a bottleneck: the compiler may be inefficient or unable to produce such large circuits.*

For example, our Dijkstra implementation for a graph of 127 vertexes and about $8 \cdot 127 \approx 1000$ edges compiled into about $350,000$ gates. The compilation took a long time, and produced a large circuit file. With bigger problem sizes, the compiler starts to run out of memory (on a 4GB server). It was also impossible to load into our graph visualization program, uDraw(Graph), thus requiring that we first obtain a sub-graph of an area of interest before being able to examine it graphically.

Thus, we designed a few extensions to the circuit model to enable loops to be expressed in the circuit, without unrolling. These extensions have not been implemented in the current version of Faerieplay.

The basic extension was a gate which conditionally sends control flow to another gate, but several other constructs were needed to minimize the evaluator's complexity, and enable the compiler to produce the desired executable circuit form.

For this discussion we are always referring to the executable circuit, ie. after it is topologically sorted. The circuit graph needs a few more helper gate types to enable the generation of the executable circuit described here.

9.9.1 Some concepts

Each loop has several important parameters:

- A *loop counter*, whose value is used inside the loop, but which also controls loop execution. These two notions are separate, even though they may be applied in the same way, ie. by accessing the loop counter's value.
- a set of variables whose values are updated on each pass through the loop. We refer to these as the *loop variables*.
- The loop consists of a set of gates which is well-demarcated from the rest of the circuit. We refer to these gates as the *loop sub-circuit*. It is the compiler's responsibility to ensure that the loop sub-circuit consists only of gates which should be executed multiple times as part of the loop.

9.9.2 Executable gate types for loops

We introduce the following gate types to support loops in the executable circuit format. We define their semantics in Table 9.2.

GOTO Each loop has one Goto gate at the end of its sub-circuit, which sends control back to the

beginning, based on a condition. Since this condition could, from the evaluator's viewpoint, be based on a dynamic value (the CVM does not differentiate between dynamic and static values), the compiler should ensure that the condition does actually refer to a static value, and more specifically to the loop counter.

STATICSELECT Similar to a SELECT gate, it obtains inputs from two sources, and returns one of them as specified by some condition, namely the loop counter value. Unlike a SELECT gate, it does not have to hide which input was selected as this is not input-dependent[4]. On the first pass through the loop, they should get values from their locations before the loop sub-circuit. On all subsequent passes, their inputs come from the end (of the previous pass) of the loop, ie. from inside the loop sub-circuit. It is the compiler's job to determine where these two input locations are for each loop variable.

LOOPCOUNTER The loop counter cannot be treated as a normal loop variable, as it itself provides the value which enables the StaticSelect gates to decide whether they are executing the first pass or subsequent passes. Thus, each loop has a dedicated loop counter gate, whose parameters include the start and end values.

[4]Since ensuring a SELECT gate's obliviousness is not onerous in the case of a scalar, and since only scalars are sent through SELECT gates, the difference between SELECT and STATICSELECT at the CVM is for now just in name

Gate	Output		Comment
LOOPCOUNTER$[start, end]$	$i \leftarrow start$ **return** i	**if** first pass	i is a state variable maintained for the gate.
	$i \leftarrow i + 1$ **return** i	otherwise	
STATICSELECT(sel, t, f)	t	**if** $sel =$ TRUE	
	f	**if** $sel =$ FALSE	Same as SELECT except not required to be oblivious.
GOTO$[target](cond)$	no result value		1 **if** $cond =$ TRUE 2 **then** send control to gate number $target$ 3 **else** do nothing (execution at next gate as usual)

Table 9.2: Semantics of executable loop gates. See Table 9.1 for an explanation of the notation. Note that the condition values for STATICSELECT and GOTO should be generated (by the compiler) to compare the value of this loop's LOOPCOUNTER value to the loop's end value.

Chapter 10

Faerieplay Implementation and Experiments

In the last three chapters we introduced Faerieplay (Chapter 7), developed a security model for it and showed that it is secure under that model (Chapter 8), and then gave a specification for an implementation (Chapter 9). In this chapter we present our implementation of the system.

Much of this chapter will be routine for readers familiar with compilers, processors and software engineering practices. For such advanced readers, we have summarized our main challenges and experiences in the beginning of the respective sections. We have included the more routine material as it may not be familiar to other readers, and we apply it in an unusual context. More detail is in the implementation itself.

We start with a description of our compiler, which translates high-level source code to a circuit. Then we describe our *Circuit Virtual Machine* (CVM), which executes the circuit while providing the security properties laid out in Chapter 8. We take a detour into the debugging provisions we developed. Finally we show an implementation of a tiny TTP using the Oblivious RAM (ORAM) algorithm. We used this implementation to compare Faerieplay with ORAM. We present our experiments and results on this system in Section 10.6.

10.1 Compiler

10.1.1 Why a new compiler

We could have tried to build on the Fairplay compiler instead of implementing ours from scratch, but we decided to build our own for two main reasons:

- We anticipated the need to modify the compiler extensively for different purposes, which is easier to do with one's own code. Additionally, Fairplay's compiler had a hand-written recursive descent parser which seemed too inflexible on the front end. We did indeed modify our compiler extensively after its initial version, for several main purposes:
  - To extend Fairplay's language, as we described in Section 9.3.5.
  - To support a second source language. The flexibility of a capable parsing tool was very helpful here, and allowed us to add the second language easily.
  - To produce different forms of output: the executable circuit, various circuit visualizations, and a circuit simulator which helped with debugging the CVM.
- Fairplay's compiler focused on manipulating a boolean circuit, whereas we needed an arithmetic circuit, and special array gates. This difference substantially reduced how much of the Fairplay compiler we could have reused.

10.1.2 Implementation experience

We implemented the compiler in Haskell, which is strong in manipulating tree-structured data, like abstract syntax trees. Haskell is additionally a very succinct and high-level language, which allowed us to enhance the compiler with fairly little effort. For circuit generation and manipulation, we extensively used the *Functional Graph Library* (fgl) which is included in the *Glasgow Haskell Compiler* (GHC) distribution.

Inflexibility of a pure functional language We did experience some difficulties stemming from Haskell's strong static type system, and functional purity. We implemented the initial version of

our compiler without any provision for tracking source code line numbers[1]. Later, as we compiled longer and more complicated programs, error messages produced by our compiler became difficult to link with the source. However, adding line number information to the AST structures used in the compiler would have required changes to every function which touched those structures, and would have made all those structures more awkward. Thus we chose a compromise solution: report several levels of context with an error message, which usually suffices to orient the user to the error's source location.

Memory efficiency Additionally, it is possible that our compiler is less scalable than it could have been in a lower-level language—we do hit memory limits when compiling programs which result in very large circuits. However it would require an alternate implementation to establish how much can be gained by optimizing. Fairplay's compiler (written in Java) also fails in a similar manner for large circuits, though we do not have a quantitative comparison.

10.1.3 Compiler outline

We used the compiler structure laid out in Appel's *Modern Compiler Construction* series [5]. Guided by this framework, the implementation went fairly smoothly. The last step, circuit generation, is not covered in compiler textbooks, but was not difficult to implement to the extent we needed, as we detail in Section 10.2. We did not see the need to produce a *static single assignment* (SSA) form of the program tree, as was done in Fairplay.

The compiler makes several passes over the code:

1. Parsing, using the BNF converter (BNFC). BNFC defines Haskell types corresponding to the syntactic elements, and automatically produces an abstract syntax tree (AST) composed of those types.

2. Type checking: checking the type correctness of the AST, building symbol tables, and converting the AST to an *intermediate form*.

[1] The parser does report line numbers, but all later stages of our compiler, like the type checker, do not.

3. Conversion to canonical form, where some of the constructs in the source code are converted to a smaller number of canonical constructs,
4. Function expansion and loop unrolling,
5. Circuit generation,
6. Circuit cleanup and topological sorting.

We describe these passes in more detail in Section 10.1.4.

10.1.4 Compiler passes

Parsing

This phase is implemented with the BNF converter (BNFC)[2], which takes a **labeled BNF!** grammar description, and produces:

- A set of Haskell data types for the abstract syntax,
- A set of functions to lex and parse text into those data types,
- A set of functions to pretty-print the abstract syntax tree,
- A formatted version of the grammar.

Figure 10.1 shows a snippet of the BNFC grammar for SFDL, which generates statements. We include the full syntax specification for SFDL in Appendix B.

Type checking and *Intermediate Format* generation.

Here the compiler does initial type-checking, and converts the abstract syntax tree (AST) produced by the BNFC-generated parser to the Intermediate Format (IF) defined in `Intermediate.hs`. The IF node types differ from the BNFC-generated AST nodes in several ways:

[2] http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/

```
1  -- statements
2
3  -- a block statement is a "{", followed by a list of Declarations,
4  -- followed by a list of Statements, and closed by a "}".
5  SBlock.     Stm     ::= "{" [Dec] [Stm] "}" ;
6
7  -- assignment
8  SAss.       Stm     ::= LVal "=" Exp ";" ;
9
10 -- and for debugging, a 'print' statement:
11 -- the String in there is a double-quoted literal string
12 SPrint.     Stm     ::= "print" "(" String "," [Exp] ")" ;
13 separator Exp "," ;
14
15 -- none of these statements need a trailing semicolon. so, only
16 -- Assignment needs it (as specified above)
17 SFor.       Stm     ::= "for" "(" Ident "=" Exp "to" Exp ")" Stm ;
18 SIf.        Stm     ::= "if" "(" Exp ")" Stm ;
19 SIfElse.    Stm     ::= "if" "(" Exp ")" Stm "else" Stm ;
```

Figure 10.1: A snippet of the *labeled BNF* grammar for SFDL, in the format defined by the `bnfc` tool. This part of the grammar generates statements.
Comment lines begin with a double dash. The basic `bnfc` syntax is:
`<Label>. <NonTerminal> ::= <Production> ;`
Double-quoted strings in the production correspond to keywords and other fixed terminals in the language. Rules in `bnfc` are terminated by a semicolon.

- Some of them carry more abstract information, eg. the full variable information for a loop counter, instead of just the name.
- Some of them carry scope information, like the set of variables declared in a scope.

Convert to canonical form

Here the compiler simplifies the intermediate format tree by converting some constructs to a canonical form. After this phase:

- All function calls are alone on the right side of an assignment (to a temporary variable if needed),
- All conditions are single variables, not complex expressions,
- All SPRINT nodes have single variables as parameters,

- SIF statement nodes are eliminated; only SIFELSE remain,
- As a side-effect, there are many generated variables and new assignment expressions.

Loop unrolling

The compiler unrolls all loops, and inlines function calls. At the end of this pass, there are no function calls, and no for-loops. There are also no scope-creating nodes in the intermediate tree (ie. no SBLOCK nodes). Since scope information is required to distinguish variables from different scopes but identical names, scope information is kept with annotations on variables. A scope annotation consists of a stack of integers. Figure 10.2 illustrates the semantics of variable scopes.

```
-       1
  -     1,1
  -     1,2
    -   1,2,1
    -   1,2,2
  -     1,3
-       2
-       3
```

Figure 10.2: Semantics of compiler-generated scope annotations for variables. A dash is a scope-creating object, ie. an SBLOCK node in the IF tree; indentation indicates nesting; and the annotation for variables at that location is shown on the right. Scope-creating objects are explicit blocks (SBLOCK node in the IF tree), instances of unrolled for-loop bodies, and inlined function bodies. A new scope depth is generated (during unrolling) by a scope-creating object, and every consecutive scope-creating object at the same depth uses an incremented value for its scope depth.

10.2 Compiler: circuit generation

This step is the most involved in the compiler, so it warrants its own section. Recall that before circuit generation begins, the compiler has unrolled loops and macro-expanded all function calls, so the tree of statements is almost linear, with branching only for conditionals. In particular, tree branches due to loops are gone. A graphical example of the IF tree of statements at this point is shown in Figure 10.3.

Circuit generation consists of "executing" this whole IF tree, statement by statement and starting

```
x = a+b;

if (x > 0)
{
  for (i = 1 to 3)
    A[i] += i;
  y = x + 7;
}
else
{
  y = x + 5;

  if (y > 5 && l > 0)
    y = A[l];
  else
    y = A[k];
}

j = x;
y = A[j];
```

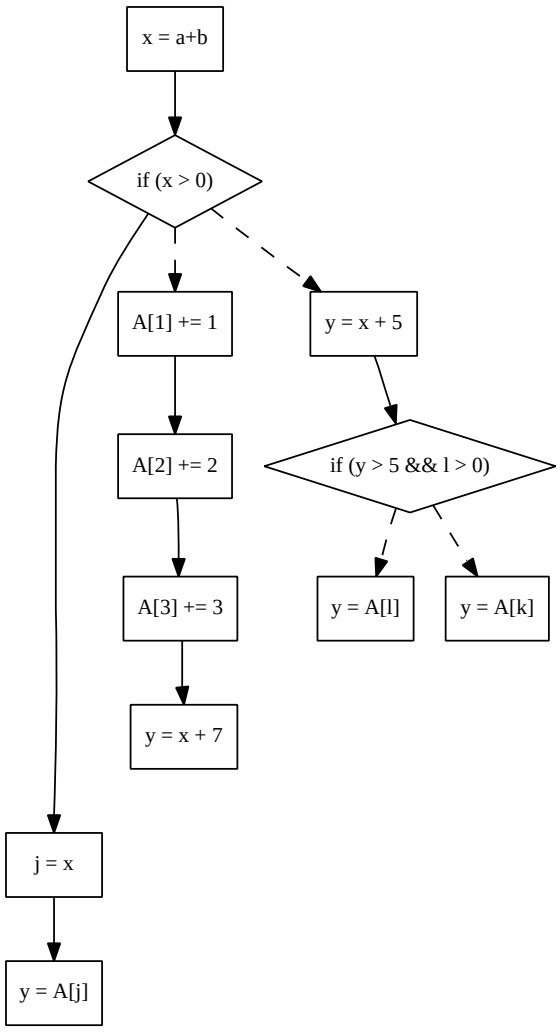


Figure 10.3: An illustrative sample of the AST after loop unrolling. The source snippet is on the left. Conditional branches are indicated by dashed lines. Note that at this point of compilation, conditionals are the only constructs which branch off the straight-line control flow.

from the first statement in the `sfdlmain` function, and generating the corresponding circuit as the output of this execution.

10.2.1 Preliminaries

Current location

An important notion during circuit generation is that of *current location*, in the IF tree. The compiler updates its state as it progresses, and thus specifying its state must be done with respect to the compiler's location in the generation process.

10.2.2 Compiler state during generation

Variable locations map

Our compiler maintains a mapping between a gate, and the high-level variable whose value (or part of value in the case of structures) is at that gate. This is required during generation so that when the IF tree contains a reference to a variable, the compiler knows at which gate, previously generated, is that variable's value currently located.

Let LOC(x) be the gate number where variable x's value is currently located. In other words, when the compiler is generating gates for some IF node, and that node references x, LOC(x) is the gate number where a value for x was last stored and can now be obtained. The compiler will thus generate an edge from LOC(x) to the gate corresponding to the IF node referencing x.

Gate counter

The compiler maintains an incrementing counter, and uses it to number the gates it generates. Thus, the earliest generated gates have the lowest numbers. The compiler currently generates a gate once all the gates it depends on for inputs have been generated, and thus the ordering of the gate numbers is already topological. Relying on this for topological sorting is brittle though, as changes to the compiler could easily result in it generating gates with numbers out of order, depending on how it obtains the gate numbers.

10.2.3 Translating various constructs

Now we go into how the compiler translates the IF tree, categorized by the various constructs present in the tree. Most of the exposition here is by example: how does the compiler translate a high-level code snippet into a circuit snippet?

Notation We use the following symbols:

$\Leftarrow$	construction of a new gate
$\leftarrow$	updating a variable's location

The notation we use to specify gates is the same as in the semantics section (Section 9.7.3). We specify gate inputs as a gate number (during compilation the compiler obtains this number using the LOC() function) from where the input value is to be fetched.

Inputs

The compiler generates INPUT gates for each input variable, where inputs are defined to be the parameters to the `sfdlmain` function. The type of each INPUT gate is determined by the type of the corresponding parameter. The name of the variable is also stored in the gate (in its comment field), so the CVM can obtain the input of that name and type during circuit execution.

`function void`
`sfdlmain (Word x) {`
`  ...`
`}`
	1 $g \leftarrow \text{INPUT}[comment = \text{x}]$
	2 $\text{LOC}(\text{x}) \leftarrow g$
	...

Variable assignments

Assignments just generate a location update, and no new gates:

`x = y;`	1 $\text{LOC}(\text{x}) \leftarrow \text{LOC}(\text{y})$

One consequence here is that all the simple assignments to generated variables, which the compiler generated in the IF tree canonicalization pass do not result in circuit gates, just in location updates, so are in a sense “cheap”.

Expressions

Gates for expressions are generated as needed, for example here an addition gate is generated, followed of course by a location update.

Arithmetic Arithmetic expressions simply produce arithmetic gates:

`x = y + z;`	1 $g_1 \Leftarrow \text{BIN}[\text{PLUS}](\text{LOC}(\texttt{y}), \text{LOC}(\texttt{z}))$ 2 $\text{LOC}(\texttt{x}) \leftarrow g_1$

Literals Literal expressions result in a LIT gate. See the next section for how LIT gates are compacted by the compiler, so each literal value is produced by only one LIT gate.

`x = 5;`	1 $g_1 \Leftarrow \text{LIT}[5]$ 2 $\text{LOC}(\texttt{x}) \leftarrow g_1$

Array read and *slicing* As described in Section 9.7.3, SLICER gates are used to split apart the output of an READDYNARRAY gate, which always contains at least two elements. For example, a simple array read in the code generates a READDYNARRAY gate and two SLICER gates:

```
var Int<32>[10] ints;
// an index:
var Int<4> i;
// a value:
var Int<32> x;
// ...
x = ints[i];
```

1 $g_3 \Leftarrow \text{READDYNARRAY}(\text{LOC}(\texttt{ints}), \text{LOC}(\texttt{i}))$
▷ SLICER static parameters are a byte offset and length
▷ Array references and integers are 4 bytes + 1 byte
▷ to indicate if this is a NIL value, see Section 9.6
2 $g_4 \Leftarrow \text{SLICER}[0, 5](g_3)$ ▷ The array reference, 5 bytes
3 $g_5 \Leftarrow \text{SLICER}[5, 5](g_3)$ ▷ The integer element, 5 bytes
4 $\text{LOC}(\texttt{ints}) \leftarrow g_4$ ▷ Array variable location updated,
▷ as it can only be used once.
5 $\text{LOC}(\texttt{x}) \leftarrow g_5$

10.2.4 Array writes

An assignment to an indirect array index requires an array write gate, and not just a location update:

`A[i] = x;`	1 $g_2 \Leftarrow \text{WRITEDYNARRAY}(\text{LOC}(\texttt{A}), \text{LOC}(\texttt{i}), \text{LOC}(\texttt{x}))$ 2 $\text{LOC}(\texttt{A}) \leftarrow g_2$

10.2.5 Conditionals

Conditionals require a generalization of the variable location map, so that each level of nested conditionals has its own map. Modify the location function to include a nesting depth: $\text{LOC}_d(\texttt{x})$ is the gate location of variable x at nesting depth d (given that the execution point is inside a nesting depth of at least d).

Definition 9 *$\text{LOC}_d(x)$ is the location of x at $\max d'$ s.t. $d' \leq d$, and x has been assigned at level d'.*

A simple conditional is translated as follows. The compiler generates gates and location updates corresponding to the conditional's body, and then SELECT gates to propagate the correct value of each scalar variable updated in the conditional, depending on the condition. Assuming the ambient nesting depth is d and inside the conditional is $d + 1$, this is a simple conditional example:

`if (t) {`
`  x = z + y;`
`}`
	▷ Conditional body; nesting depth is $d + 1$
	1 $g_1 \Leftarrow \text{BIN[PLUS]}(\text{LOC}_{d+1}(\texttt{z}), \text{LOC}_{d+1}(\texttt{y}))$
	2 $\text{LOC}_{d+1}(\texttt{x}) \leftarrow g_1$
	▷ and conditional exit
	3 $g_2 \Leftarrow \text{SELECT}(\text{LOC}_d(\texttt{t}), \text{LOC}_{d+1}(\texttt{x}), \text{LOC}_d(\texttt{x}))$
	4 $\text{LOC}_d(\texttt{x}) \leftarrow g_2$

10.3 CVM implementation

For the implementation of the circuit virtual machine (CVM), we had to consider the resource constraints of the 4758 secure coprocessor, and use an efficient language. We settled on C++ as providing a good mix of runtime efficiency and high-level abstraction, which helps with achieving a correct implementation.

10.3.1 Challenges and retrospective

The main challenges in implementing the CVM revolved around achieving correctness. They were not very significant, but did force us to use debugging and testing techniques which an easy project

may not need.

- A modular program structure enabled us to
  - test components independently, eg. the CVM–host API, encryption and integrity checking, and the PIR/W implementation. The latter presents a simple read-write API, so we could automatically generate a long sequence of operations to test it.
  - run the whole CVM with parts of it disabled to narrow down the location of an error. Eg. we could disable encryption, or disable the whole PIR/W algorithm and access arrays directly.
- We implemented a simulator of the CVM in our compiler. This simulator was very simple, so we could be reasonably sure of its correctness. It also serves as a specification of the functional semantics of the CVM (see Section 9.7.3 on page 153).

  For debugging the CVM, we had both the CVM and the simulator output an identical textual trace of gate values computed during a circuit evaluation. Then we could use the standard `diff` tool to see where the CVM diverged from the simulator.

Given the trust that Faerieplay vests in the CVM, a more rigorous validation of the CVM implementation would be appropriate, and especially of its security properties which are difficult to test using standard techniques.

We did not experience many memory management problems (leaks and invalid accesses), as we isolated memory management behind suitable abstractions, eg. in file `countarray.hpp`. This allowed fairly close interaction with raw C memory where needed, but also provided reference counting for automatic and timely deallocation.

10.3.2 Host containers

For mass-storage, ie. anything requiring $\omega(M + \log N)$ bits, the TTP has to use external space. The host provides such a mass-storage service in the form of contiguously-indexed *containers*. We show the TTP-to-host API in Section 7.6.3.

Recall from the security model in Chapter 8 that the data which the TTP sends to its host via the container API is available to the adversary (who controls the host)—the API is the direct communication between the TTP and its host/adversary. The TTP implements any security provisions, like encryption and blinded array access, on top of the host API.

Containers are currently named in a hierarchical manner, using a slash ("/") as a separator.

The host can implement containers as it sees fit, for example using a filesystem directory per container, and one file per element; or using one file for all of a container's elements[3], mapping the file into memory with `mmap(2)` and accessing elements via memory pointers; or even keeping all the elements in RAM if this is required for adequate speed.

10.3.3 PIR/W implementation

The algorithm we use for our earlier PIR/W implementation is described in Section 3.5. We use almost the same implementation here, so do not describe it again.

We had to add several features to the implementation as required by the Faerieplay setting:

- Array writes have a new `enable` input, which controls whether the write actually takes place or not. In either case, the CVM has to go through the same steps so the adversary cannot tell whether the write was a dummy or not. See Section 9.7.2 for details of the `enable` input.

We also did not need to use some features, in particular the session-continuity provisions (Section 3.4.2), which enabled re-permutation to be done concurrently with servicing client requests. The reason for those provisions was to maintain low interactive latency for online data accesses. In the Faerieplay setting, the only relevant cost is that of the whole computation, so there is no problem if the TTP halts the computation of gates while it carries out expensive pre-processing.

10.3.4 Array gates

A naive implementation would have a separate copy of the array for every gate, as is done for scalars. However, this induces a $O(N)$ cost for array gates, which is much more than the PIR/W

[3] All elements in a container are currently the same size, and will remain so for the common case at least.

cost per operation. We would like to spend no more than $\text{INDADDRTIME}(N)$ (amortized) time per gate (see Definition 1 on page 100). We can take advantage of the fact that the array pointer value at any gate is used by only that one gate—this is guaranteed by the circuit format (CF) as described in Section 9.6. Thus, since we know that only the current gate needs the current array value, we can do operations destructively without copying, and in particular update the array (ie. T on every R/W access, and A when re-permuting) in place.

10.4 Debugging

While developing the Dijkstra example, we reached the situation where bugs in the actual SFDL program were more problematic than bugs in the compiler or circuit evaluator. Since the entire platform is experimental, there are no development tools like debuggers to help with this. Moreover, the circuit model of evaluation results in a very different, and rather less intuitive, order of execution than a RAM implementation. Thus, we had to provide ourselves with several compiler capabilities to help with development.

10.4.1 A circuit simulator

First we added a circuit simulator to the compiler, which would evaluate a circuit without any of the PIR/W complications, and as close to the compiler as possible, so as to provide a baseline execution trace against which to compare runs of the 4758 evaluator. This was intended to help debug an SFDL program without worry that observed problems were caused by bugs in the circuit evaluator. This evaluator outputs a list of all gate values produced during the execution, including all internal gates. This trace helps to determine at what point the execution starts to go wrong.

However, the association between a particular gate and the corresponding SFDL source code is non-obvious, thus making source-level debugging difficult. To help with this, we introduced print gates, which serve as an explicit connection between the source and some gates in the circuit.

10.4.2 Print statements and gates

We introduced a `print` statement to SFDL, with a syntax:

print ("fixed prompt", v_1, ..., v_n);

The compiler translates this into a single PRINT gate with inputs v_1 to v_n, and the same outputs. This ensures that subsequent uses of the values come from that gate, ie. the values are printed just before they are used next, whatever circuit evaluation order is used. Being able to print multiple values at one gate is essential, as it allows more complicated questions with dependencies to be answered, like "What are the values of `i`, `A[i]` and array `H` at some given point" (when we suspect that `H` enters a bad state).

If not for the explicit tying together of the variables to be printed (eg. by having a separate `print` statement for each variable), the values could be printed out at very different times, depending on how the compiler ordered the gates into a linear list. This would make it impossible to build a coherent snapshot of the computation.

10.4.3 Circuit visualization.

Debugging the circuit evaluator, and following traces of circuit execution, is helped along by having a clear visualization of the circuit to look at. We have the compiler generate a graph for two popular graph formats:

- uDraw(Graph), which can be used to explore the circuit interactively, and
- Graphviz[4], which is best used through a front-end like ZGRViewer [5]. Graphviz and its front-ends have fewer capabilities than uDraw(Graph) but are open source.

Both of these systems have problems handling very large graphs (eg. 10K vertexes appears to be out of reach for both), so we provide the capability of specifying only a part of the circuit to be exported as a graph, eg. "the sub-graph e or fewer edges away from gate number g".

[4] http://www.graphviz.org/
[5] http://zvtm.sourceforge.net/zgrviewer.html

10.5 Oblivious RAM prototype

In order to check how Faerieplay compares to its direct competition, we implemented a simple but fairly complete ORAM prototype consisting of a MIPS emulator whose RAM we implemented using the same hw-PIR/W implementation as in Faerieplay.

As we discussed in Section 7.7 on page 109, a tiny TTP can be implemented as an Oblivious RAM system, which runs a RAM program, with oblivious indexing for every memory access (code and data) of the program. Faerieplay on the other hand only uses oblivious indexing where required for indirect array access, and cheap direct access for the rest of the program. This results in considerable time savings, as we demonstrate in Section 10.6.2.

Our MIPS emulator supports all of the integer MIPS instructions (MIPS32 to be precise); see the SPIM appendix in Hennessy and Patterson's *Computer Organization and Design* for a good description of the MIPS32 architecture [45]. We have tested it with some non-trivial programs, compiled with `gcc` with `-O2` optimization, culminating with our canonical Dijkstra example.

10.5.1 Memory handling

Physical memory The emulator exposes a plugin interface for its physical memory emulation[6], and now has two implementations:

- A simple array of bytes on the host machine, and
- a PIR/W array of 32-bit elements.

Virtual memory All user code issues virtual memory addresses. Our emulator performs memory mapping itself, onto physical addresses. It aims to squeeze the entire working set of the program into as small an amount of emulated physical memory as possible, to minimize the overhead of ORAM.

Given a virtual address, the emulator examines it and classifies it into one of several parts of memory:

[6] We use the standard `dlopen` function for loading the plugin implementation dynamically.

- code
- static data
- heap
- stack
- kernel memory (not used and illegal for now)

The boundaries are as illustrated in Figure 10.4. With this classification, it can compute where in the physical memory this virtual address should be.

Allocating physical memory efficiently The emulator obtains the code size and static data size from the executable file and thus allocates the physical memory for static sections precisely.

It maps the stack starting at the top of physical memory and growing downwards, and the heap from just above the static data section and growing up (as usual). Thus the program must really run out of physical memory before the heap and stack will collide.

The emulator can be run in a *memory diagnostic mode* (using the native array implementation of physical memory) which reports how much dynamic memory (stack and heap) the program actually used. A user can thus profile approximately how much memory a program needs before running it with the ORAM physical memory emulation.

10.5.2 The emulator's operating system

The "operating system" on the emulator is currently not separated from the actual emulator, and it supports only several system calls: `read`, `write`, `mmap`, `munmap` (only anonymous memory mapping, to support `dietlibc`'s implementation of `malloc`), and `exit`. These allow simple I/O and dynamic memory management. There is currently no distinction between the OS and the emulated hardware—a more principled approach would be to write the OS to run on the emulator, and interact with it only through the standard machine interface.

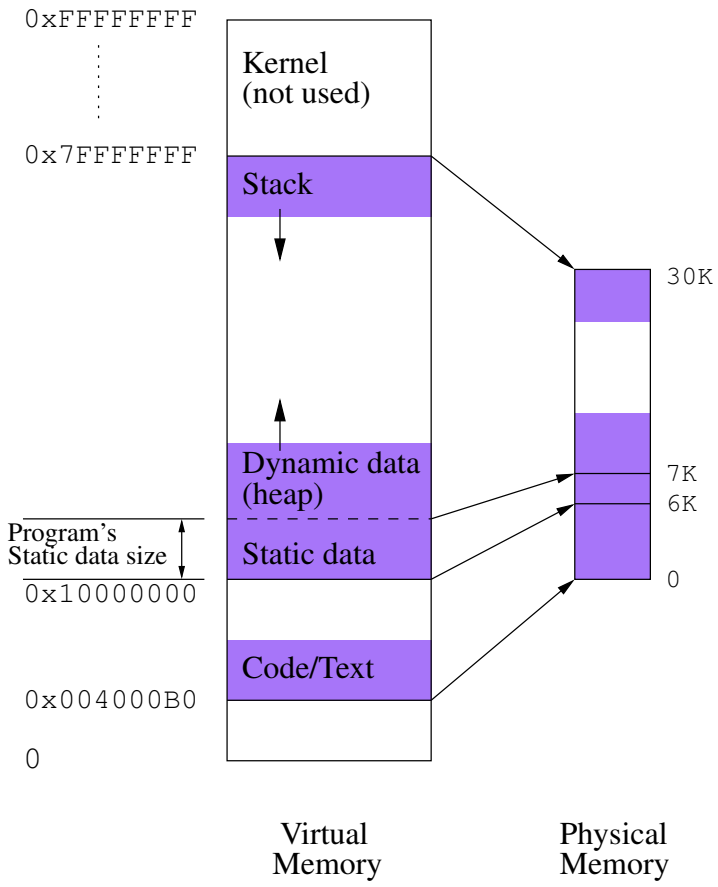


Figure 10.4: The memory layout of a MIPS machine, and an example mapping into a small amount of physical memory. Areas in use are shown shaded. Obviously large parts of the virtual memory are unused. Virtual addresses (VAs) falling outside the mapped range are flagged as errors. The stack is mapped to the top of physical memory, and the heap to the the bottom of the dynamic memory area, and they grow towards each other as usual. If the stack and heap collide, the program fails arbitrarily without warning.

10.5.3 MIPS emulator front-end

Our emulator is run using the program `runmips`, which takes a compiled and *statically linked little-endian MIPS executable in ELF format*, and:

- initializes a MIPS emulator instance,
  - includes initializing the emulator's memory with parameters like the various section sizes (text, static data, dynamic data),
- loads the MIPS executable (code and initialized static data) into the memory,

- loads `argv`, `argc` and `envp` into the top of the stack. It passes on the `argv` that it itself received on its command line.
- Starts the emulator CPU at the start address specified in the ELF file (the address of the `__start` symbol).

`runmips` uses the `bfd` library[7] to handle the ELF executable.

10.5.4 Development toolchain

We produce executables for this system in the normal way—from C source compiled with `gcc`, version 3.3, and statically linked against the compact `dietlibc`[8] C library. The use of library functions `malloc(3)`, `free(3)`, `read(2)`, `write(2)`, `exit(3)` works well. `printf(3)` and friends work too, but pull in several thousand instructions, which `dietlibc` actively discourages. The only limit to use of the C library is the limited syscall support. For example, no process creation is currently possible.

The same C code can be compiled for any other target (eg. the user's machine) and debugged there, as the MIPS emulator does not provide any application debugging functionality except copious logging of its activity, which is more useful for debugging the emulator itself.

10.6 Experiments and results

Most of our measurements and other experimental results are based on running a simple but complete implementation of Dijkstra's algorithm with heaps. We ran similar implementations on Faerieplay and on our Oblivious RAM prototype.

Comparing our system to Fairplay is more difficult as Fairplay a lot less scalable than the TTP-assisted approaches. Thus we had to resort to indirect experiments and measurements to extrapolate a comparison to our current setting of graph searching.

[7] The BFD library is part of GNU binutils; see http://www.gnu.org/software/binutils/, and http://sourceware.org/binutils/docs-2.17/bfd/index.html

[8] http://www.fefe.de/dietlibc/

10.6.1 Dijkstra implementation in SFDL

Our SFDL implementation of Dijkstra's algorithm represents the graph as shown in Figure 10.5. The graphs we ran it on ranged in size from 7 to 255 vertexes. They were generated randomly with $5 - 10$ outgoing edges per vertex.

```
1  type Vertex     = struct { Idx num,
2
3                              Idx edge_list_head,
4                              Word num_out_edges,
5
6                              Weight d, /* distance from source */
7                              Idx pi_idx, /* predecessor */
8
9                              Idx heap_idx
10                             };
11
12 type Edge       = struct { Idx dest_idx,
13                             Weight w };
14
15
16 type Graph      = struct { Vertex[V] Vs,
17                             Edge[E] Es };
```

Figure 10.5: The graph representation in the SFDL implementation of Dijkstra. The types `Word`, `Idx` and `Weight` are all integers. In the `Graph` structure, The `Es` array is a flat array of edges, which holds all the edges in the graph. The adjacency list (ie. the outgoing edges) of vertex `V` is located contiguously within `Es`—it starts at `Es[V.edge_list_head]`, and occupies the next `V.num_out_edges` elements in `Es`.

10.6.2 Against oblivious RAM

To compare the Faerieplay circuit implementation against an implementation using oblivious RAM, we wrote a simple C implementation of the algorithm, and ran it on the MIPS emulator using an oblivious RAM. The C implementation represents the graph as summarized in Figure 10.6. We ran it on the same graphs as the SFDL implementation (Section 10.6.1).

```
1  /* the graph */
2  typedef struct {
3
4      /* an array of vertices */
5      node_t * nodes;
6      size_t len;
7
8  } graph_t;
9
10 /* a graph vertex */
11 typedef struct node_t {
12     node_idx_t num;
13
14     list_t * out_edges;
15
16     /* SP algorithm fields: */
17     int d;              /* current distance estimate from source */
18     struct node_t * pi; /* the predecessor node in the shortest path */
19
20     /* and for the heap, if needed: */
21     index_t heap_idx;
22 } node_t;
23
24 /* a standard head-tail list of edges */
25 typedef struct _list {
26     edge_t head;
27
28     struct _list * tail;
29 } list_t;
30
31 /* an edge, with a weight and destination vertex */
32 typedef struct {
33     struct node_t * dest;
34     int w;
35 } edge_t;
```

Figure 10.6: Summary of the graph representation in the C implementation of Dijkstra, targeting our MIPS/ORAM emulator.

Experimental method

We ran both implementations on graphs of various sizes. The graphs were randomly generated, with a fixed number of vertexes, and a random outgoing edge list from each vertex: the number of out-edges for each vertex was normally distributed around a fixed mean, between 5 and 8, and each edges's weight and destination vertex were uniformly selected. A random pair of vertexes was used as the source and destination inputs to the shortest path algorithm.

The hardware setup was the same in both cases—an IBM 4758 as TTP, running the TTP portion of the Faerieplay CVM. The host was a Dell Optiplex desktop, with a single Intel Pentium IV 1.8

vertices V	7		15		63	
	F	O	F	O	F	O
Time in mins	4.1	174	8.9	300	53	25.9 hrs
instructions/gates	5.4K	10.5K	15.1K	18.7K	93K	90K
mem/array reads	640	1.2K	1.7K	2.4K	11.3K	13K
mem/array writes	510	1K	1.1K	1.8K	9.5K	8K
RAM size in words		4.8K		4.9K		6K

Table 10.1: Table of running times of Dijkstra on Faerieplay (F) and ORAM (O), with different graph sizes in terms of vertexes. See Section 10.6.1 for details of the graphs we used.

GHz CPU.

Results

We recorded several measurements from the experiments. Note that many of the measurements for the ORAM implementation are not specific to Oblivious RAM but apply in general to the RAM machine form of the function.

- The actual running times.
- How many operations the computation consisted of—gates for Faerieplay and instructions for RAM respectively,
- How many memory accesses, categorized into reads and writes, each implementation performed. For the RAM implementation this includes instruction fetches. For Faerieplay this includes just indirect array accesses, which are expensive, and not accesses to the circuit, which are cheap.

The measurements are shown in Table 10.1. The same measurements are presented graphically in Figure 10.7.

Commentary

The measurements we obtained are for the most not surprising. ORAM has a larger RAM than Faerieplay has indirect arrays, and thus it spends much more time on oblivious access.

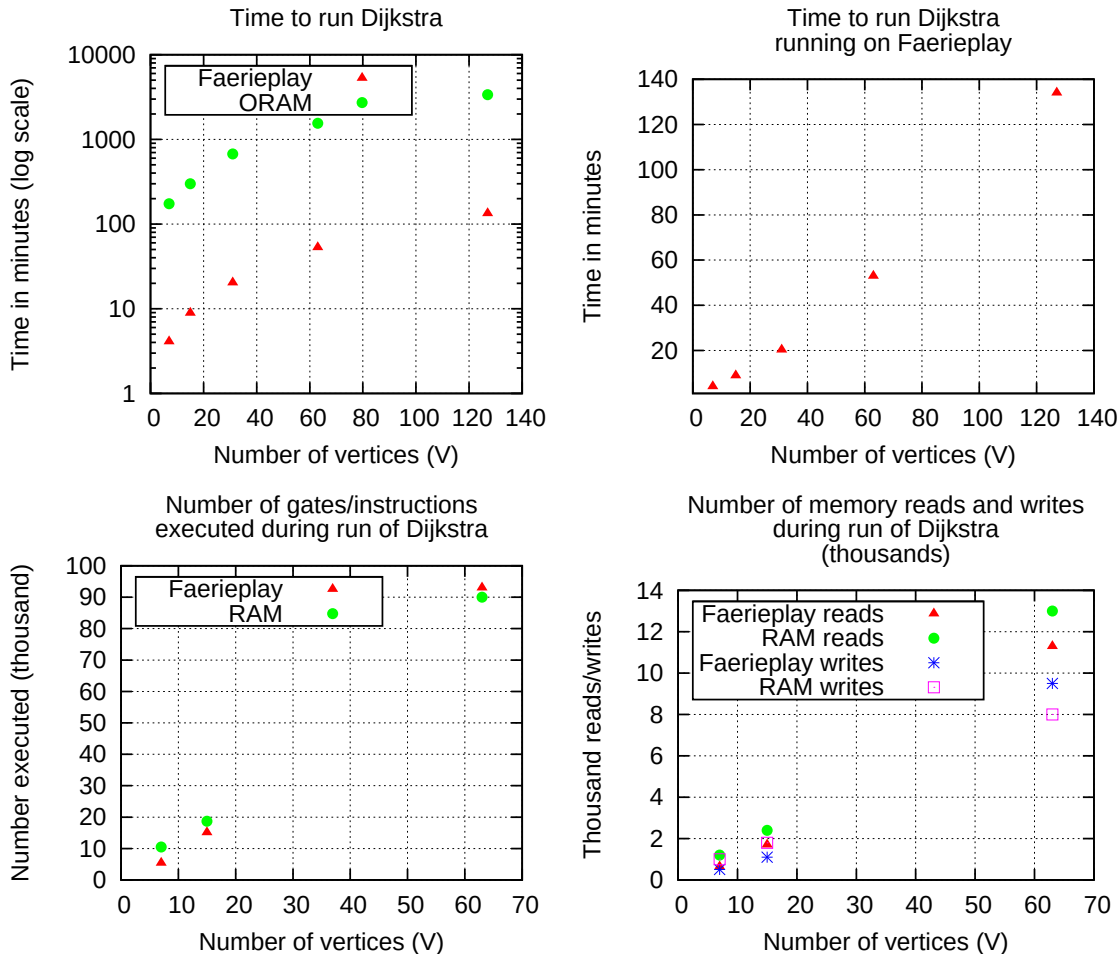


Figure 10.7: Graphs presenting the results of running a graph shortest path search using Dijkstra's algorithm, on both Faerieplay and our Oblivious RAM prototype. The first plot of running times is shown with a logarithmic scale as the difference between Faerieplay and ORAM is too big (about 30 times in this case) to show both on a single linear axis. The second plot shows Faerieplay times with a linear scale.

The number of operations in each case is quite similar, which suggests that the Faerieplay circuit version is not burdened with overhead when compared to the optimized RAM executable (we compiled it with `gcc` at `-O2` optimization)

One surprising result is that both implementations generate a similar number of memory accesses. I would have expected that Faerieplay would have fewer indirect accesses than ORAM, because its indirect indexing is limited to arrays which require it. One explanation is that the circuit implementation performs some array accesses inside non-selected conditional branches. Those accesses are "dummies", but they still incur the cost of the indirect array access.

10.6.3 Against Fairplay: Indirect comparisons

We could not run our Dijkstra SFDL code with Fairplay, as the compiler does not support enough of the language we currently use. Some amount of work on the Fairplay compiler could probably fix this, but we use several indirect ways to compare the performance of the two systems.

Pure indirect array accesses

We ran an SFDL program, shown in Figure 10.8, which sums N 16-bit numbers in an indirectly-addressed array, thus generating N indirect accesses to an N-element array. The example is of

```
1  const N = ...;
2  const WordSize = 16;
3
4  type Word = Int<WordSize>;
5
6  type PairT = struct {
7    Word x;
8    Word y;
9  };
10
11 var PairT [N] ins;
12 var Word sum;
13
14 // ...
15
16 // 'ins' is provided as a user input.
17
18 for (i = 0 to N-1) {
19     // sum the .y fields in order decided by the .x fields.
20     // The second lookup into 'ins' (to fetch the 'x' field)
21     // is indirect. The first lookup, for .y, is direct
22     // (note that the loop is unrolled)
23     sum = sum + ins[ ins[i].y ].x;
24 }
```

Figure 10.8: Program which generates only indirect accesses to an N-element array. Abridged slightly for brevity.

course artificial, as one would normally sum the array in a fixed order, which Fairplay does efficiently, treating each array member as a separate variable. We show this example as a pure illustration of the difference in indirect array access speed, which can then be used to predict relative performance in other more realistic programs, like our Dijkstra shortest paths example.

One run of the program used the Fairplay version 2 compiler and evaluation engine, and the other used the Faerieplay compiler and CVM in a 4758.

The hardware setup for the Fairplay runs was: Alice and Bob[9] each on a separate machine with a 2.7 GHz Xeon CPU, 4 GB memory, and SCSI drive, connected by gigabit Ethernet.

The results are shown in Table 10.2. They indicate that the TTP-based approach is indeed vastly more efficient than the blinded circuit approach, even with the latter on very fast machines.

N (array length)	FP gates	TTP gates	FP time	TTP time
64	193K	448	64	12
128	772K	896	255	26
256	3,085K	1792	1095	57
512	-	3584	-	142
1024	-	7168	-	372

Table 10.2: Circuit size and program execution time in seconds for different array lengths. The program simply summed the array of 16-bit integers, forcing the array addressing to be indirect, ie. it did N indirect accesses into an N-element array. The Fairplay implementation failed to evaluate its circuit for $N > 256$, due to insufficient memory (the JVM could use a maximum of 2.7 GB)[10]. Theoretically, we would expect the Fairplay running times to be $\mathrm{O}(N^2)$, and the TTP running times to be $\mathrm{O}(N\sqrt{N}\log N)$, which appears to be consistent with these results.

Scalar-only program

To give an idea of how Faerieplay and Fairplay compare when there is no indirect-addressing involved, we ran a program which performs N 32-bit additions. The results are in Table 10.3, and indicate that the two execution models are quite similar. The blinded circuit evaluation system has an edge through much faster hardware, whereas the TTP-based evaluation benefits from larger (and hence fewer) gates and wires.

[9] Fairplay uses the more traditional character names

[10] It would be conceptually simple, though tedious, to make Fairplay evaluate its circuit without having the whole circuit in memory. Then the memory bound would go away, but that would be accompanied by an increase in running time due to all the extra disc activity.

N (additions)	FP gates	TTP gates	FP time	TTP time
512	33K	513	17	9
1024	66K	1025	33	17

Table 10.3: Circuit size and program execution time in seconds for N 32-bit additions. Fairplay circuit adders for b-bit integers are constructed (as usual) with b full-adders each, where each full adder consists of 2 3-input gates. Thus the difference in gate count is about a factor of 64.

Chapter 11

Conclusion

We have presented Faerieplay, our tiny TTP-based system for secure multiparty computation (SMC). It provides a much more efficient and scalable implementation of SMC than the traditional self-reliant protocols. Within the field of tiny TTP-based schemes, it provides higher performance than a solution based on the existing oblivious RAM technique.

11.1 Evaluation

Here we evaluate the various aspects of Faerieplay in light of our experience with the system.

11.1.1 Faerieplay circuits

Programming for a static circuit forces a particular approach to the representation of data, and implementation of algorithms. For example, linked lists are quite awkward to implement, so data representation leans towards using arrays. Iterative algorithms need to be terminated by statically known values, which means that an implementation has to specify worst-case parameters, and perhaps discard some iterations with a (dynamic) condition. Iterative approximation algorithms cannot simply iterate until they converge, but must have a fixed iteration count.

Many of these extra constraints require the programmer to be quite familiar with the algorithms in question, to be able to provide accurate worst-case bounds, to know where the static control

requirement means that an algorithm cannot be implemented efficiently in this manner, to be familiar with the convergence properties of approximation algorithms instead of simply waiting till they converge.

The benefits of the static nature of the circuit are that it allows the computation system to execute the circuit without worrying about hiding the access pattern to it from the adversary. The burdens on the programmer directly yield the benefit of rendering the execution control flow uninteresting to the adversary, and thus allowing at least half of what would be indirect memory accesses in a RAM program to become cheap direct accesses. The alternative approach to TTP-based secure computation—running a RAM program on an Oblivious RAM—allows running unmodified standard programs, but with considerably higher overhead as we saw in Section 10.6.2 on page 185.

11.1.2 Faerieplay programming environment

Programming is difficult in a good environment, and can become unmanageable in experimental environments. Faerieplay has both an experimental compiler and experimental execution environment, so nothing in the whole chain is proved, from a user (programmer) perspective. Several provisions helped to mitigate this situation: comprehensive static checking during compilation, support for "printf debugging", circuit emulation by the compiler, and circuit visualization tools.

Arguments constantly rage about the costs and benefits of statically-typed languages, but in this experimental setting, checking as much as possible about a program statically is essential. In the absence of good debugging and introspection support in the execution environment, a programmer would want to be as confident as possible about a program before starting to run it. Thus, the Faerieplay compiler is quite strict with enforcing type-safety rules.

The `print` statement we introduced to Faerieplay SFDL was essential in debugging, as without this there was no way to get a snapshot of the computation at the level of the source code. Without such a provision, it is still straightforward to observe the value of a single variable, as the compiler labels gates with the variable they correspond to, but then locating the contemporary values of other variables is not possible.

The programs which the Fairplay project developed were all simple single-page functions, which

can be easily confirmed correct by inspection. It would be even more challenging to provide debugging support for the boolean circuit which Fairplay uses, as that is more removed from the source code than an arithmetic circuit is.

11.2 Future work

11.2.1 Controlled partial leakage of information

As we have seen, ensuring that *no* information leaks to the adversary imposes considerable costs. If we could show that some dynamic parameter of an algorithm, like the iteration count until an approximation converges adequately, does not convey much useful information to the adversary, then we could allow a Faerieplay program to use such parameters in its control flow.

11.2.2 Checking the functionality for information leakage

With complex SFDL programs, it may not be obvious how information flows from the inputs to the outputs. Checking such information flow properties has been studied, with tools like JFlow which provides a way to check information flow in a Java program with ownership annotations on variables [63]. Integrating such functionality with Faerieplay could be useful, to complement the black-box execution environment provided by Faerieplay already. The ability of the Faerieplay compiler to accept different source languages would be helpful here—the compiler could be easily extended to support a subset of Java with JFlow annotations, and then the JFlow checker could be run on the same source used by Faerieplay.

11.2.3 Programmer conveniences

Faerieplay does not currently support pointers or references, which allow construction of linked structures like lists and trees. Providing some memory heap abstraction with some mechanism to reference objects in the heap could be convenient. A heap would be backed by an indirect array. This extension would require that the programmer specify the size of a heap, and should support multiple heaps, so that each one can be made as small as possible.

11.2.4 Oblivious array access algorithm

As we described in Section 5.1.2, the new oblivious access algorithm by Williams and Sion [91] may provide a much more efficient way to implement array gates in Faerieplay. The trade-offs to consider are its need for $\Omega(\sqrt{N})$ trusted memory, and the current lack of security assurance against an active adversary.

11.3 Conclusion

We have demonstrated that the incorporation of a trustworthy device into a two-party Secure Function Evaluation protocol brings about considerable efficiency gains. This is especially true for functionalities which are expensive in the circuit model, like indirect addressing of arrays. This efficiency comes at the cost of the protocol players having to trust a third party. The devices we considered are designed to warrant exactly this kind of trust, and through that they impose some challenges to their users, like small memory and slow processors, which we deal with.

We believe that it is important that potential users of secure protocols have a range of choices in the trade-off of performance vs. self-reliance, to meet their particular needs. Thus far the choices have largely erred on the side of self-reliance, and this work has attempted to provide a sample point in the performance corner. Investigation of this trade-off should continue.

Appendix A

Programming for a circuit machine

As the Fairplay project has demonstrated, programming for a circuit machine does not have to be very different from standard programming for a RAM machine. A compiler can target a boolean or arithmetic circuit virtual machine architecture just like it can target a RAM architecture. The Faerieplay compiler and CVM support most of the high-level language features that we are currently used to.

A.0.1 Limitations

The circuit format Faerieplay uses is a standard "one-pass" circuit, where values for each gate are computed once in some order which respects the gates' dependencies on each other. The Faerieplay CVM does not define any control flow constructs—it computes the whole circuit once[1]. The static nature of the circuit imposes some limitations on the high-level constructs available to the programmer. (In this discussion we use the standard term *static* to mean "known at compile time", ie. independent of any run-time inputs.)

- Loop iteration count must be statically determined.
- Recursion termination must be controlled by static values.
- Arrays must be a static size.

[1] We have designed an optimization to save unrolling of loops, and thus reduce circuit size, which is described in Section 9.9. This preserves the same functional and security semantics as a fully unrolled one-pass circuit.

Most of these limitations can be worked around, from a programmer's point of view. For example, a programmer can convert dynamically-controlled looping into a static loop with a conditional, as illustrated in Figure A.1.

```
var n;
n = ...;      // n could be dynamic
for (i = 0 to n)

    ...;
```

```
> const MAX_N = ...;
  var n;
  n = ...;      // n could be dynamic
| for (i = 0 to MAX_N)
>   if (i <= n)
      ...;
```

Figure A.1: Changes needed to implement the loop with a dynamic termination condition on the left. The non-static variable `n` is removed from the loop condition and replaced with a static value `MAX_N` which provides an upper bound. Then a condition is added to give the same behavior as the original loop.

Such a transformation could be done almost automatically– most likely the programmer will have to supply the value of `MAX_N`, and also keep in mind that the runtime complexity will be $O(\texttt{MAX\_N})$, and not directly related to `n`.

Dynamically-sized arrays can be simulated by using one fixed-size array as a heap from which smaller pieces of memory can be dynamically allocated, similarly to `malloc` and `free` in the C library. In this case, each access into a heap array will cost $O(\text{IndAddrTime}(\text{heap size}))$, instead of the (smaller) size of the actual array being referenced, and since $\text{IndAddrTime}(\cdot)$ is more than constant, such functionality should be kept to a minimum. (Recall that $\text{IndAddrTime}(N)$ is the cost of indirect indexing into an array of N elements; see Definition 1 in Section 7.4.)

Debugging Debugging programs is always a challenging issue, and especially so on an experimental platform. We present the provisions we make for a programmer to debug Faerieplay source code in Section 10.4.

The Language Sfdl

BNF-converter

July 7, 2008

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Sfdl

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded.

Literals

String literals ⟨*String*⟩ have the form "x", where x is any sequence of any characters except " unless preceded by \.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Sfdl are the following:

```
Boolean  Int    cast
const    else   enum
false    for    function
if       print  program
struct   to     true
type     var    void
```

The symbols used in Sfdl are the following:

```
{    }    ,
=    ;    (
)    :    <
>    <‘>  [
]    &    .
!    –    ~
*    /    %
+    <<   >>
<=   >=   ==
!=   ^    |
&&   ||
```

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of Sfdl

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Prog*⟩ ::= `program` ⟨*Ident*⟩ { ⟨*ListDec*⟩ }

⟨*ListDec*⟩ ::= ϵ
| ⟨*Dec*⟩ ⟨*ListDec*⟩

⟨*ListStm*⟩ ::= ϵ
| ⟨*Stm*⟩ ⟨*ListStm*⟩

⟨*ListIdent*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `,` ⟨*ListIdent*⟩

⟨*Dec*⟩ ::= `const` ⟨*Ident*⟩ `=` ⟨*Exp*⟩ `;`
| `type` ⟨*Ident*⟩ `=` ⟨*Typ*⟩ `;`
| `var` ⟨*Typ*⟩ ⟨*ListIdent*⟩ `;`
| `function` ⟨*Typ*⟩ ⟨*Ident*⟩ `(` ⟨*ListTypedName*⟩ `)` `{` ⟨*ListDec*⟩ ⟨*ListStm*⟩ `}`
| `function` ⟨*Typ*⟩ ⟨*Ident*⟩ `(` ⟨*ListTypedName*⟩ `)` `:` ⟨*Typ*⟩ `{` ⟨*ListDec*⟩ ⟨*ListStm*⟩ `}`

⟨*Typ*⟩ ::= `Int` `<` ⟨*SizeExp*⟩ `>`
| `Int` `<‘>`
| `Boolean`
| `void`
| `struct` `{` ⟨*ListTypedName*⟩ `}`
| `enum` `{` ⟨*ListIdent*⟩ `}`
| ⟨*Typ*⟩ `[` ⟨*Exp*⟩ `]`
| ⟨*Typ*⟩ `&`
| ⟨*Ident*⟩

⟨*TypedName*⟩ ::= ⟨*Typ*⟩ ⟨*Ident*⟩

⟨*ListTypedName*⟩ ::= ϵ
| ⟨*TypedName*⟩
| ⟨*TypedName*⟩ `,` ⟨*ListTypedName*⟩

⟨*Stm*⟩ ::= `{` ⟨*ListDec*⟩ ⟨*ListStm*⟩ `}`
| ⟨*LVal*⟩ `=` ⟨*Exp*⟩ `;`
| `print` `(` ⟨*String*⟩ `,` ⟨*ListExp*⟩ `)`
| `for` `(` ⟨*Ident*⟩ `=` ⟨*Exp*⟩ `to` ⟨*Exp*⟩ `)` ⟨*Stm*⟩
| `if` `(` ⟨*Exp*⟩ `)` ⟨*Stm*⟩
| `if` `(` ⟨*Exp*⟩ `)` ⟨*Stm*⟩ `else` ⟨*Stm*⟩
| ⟨*Stm*⟩ `;`

⟨*ListExp*⟩ ::= ϵ
| ⟨*Exp*⟩
| ⟨*Exp*⟩ `,` ⟨*ListExp*⟩

⟨*LVal*⟩ ::= ⟨*Exp*⟩

⟨*Exp13*⟩ ::= ⟨*Ident*⟩
| ⟨*Integer*⟩
| `true`
| `false`
| `(` ⟨*Exp*⟩ `)`

⟨*Exp12*⟩ ::= ⟨*Exp12*⟩ `[` ⟨*Exp*⟩ `]`
| ⟨*Exp12*⟩ `.` ⟨*Exp13*⟩
| ⟨*Ident*⟩ `(` ⟨*ListFunArg*⟩ `)`
| ⟨*Exp13*⟩

⟨*Exp11*⟩ ::= `!` ⟨*Exp12*⟩
| `-` ⟨*Exp12*⟩
| `~` ⟨*Exp12*⟩
| ⟨*Exp12*⟩

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ `*` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `/` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `%` ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp9*⟩ ::= ⟨*Exp9*⟩ + ⟨*Exp10*⟩
| ⟨*Exp9*⟩ - ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp8*⟩ ::= ⟨*Exp9*⟩ << ⟨*Exp9*⟩
| ⟨*Exp9*⟩ >> ⟨*Exp9*⟩
| ⟨*Exp9*⟩

⟨*Exp7*⟩ ::= ⟨*Exp8*⟩ < ⟨*Exp8*⟩
| ⟨*Exp8*⟩ > ⟨*Exp8*⟩
| ⟨*Exp8*⟩ <= ⟨*Exp8*⟩
| ⟨*Exp8*⟩ >= ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp6*⟩ ::= ⟨*Exp7*⟩ == ⟨*Exp7*⟩
| ⟨*Exp7*⟩ `!=` ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ `&` ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ `^` ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ | ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ `&&` ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp1*⟩ ::= ⟨*Exp1*⟩ || ⟨*Exp2*⟩
| ⟨*Exp2*⟩

⟨*Exp*⟩ ::= ⟨*Exp1*⟩

⟨*SizeExp3*⟩ ::= ⟨*Ident*⟩
| ⟨*Integer*⟩
| ⟨*Ident*⟩ `(` ⟨*ListSizeFunArg*⟩ `)`
| `(` ⟨*SizeExp*⟩ `)`

$$
\begin{array}{lcl}
\langle \mathit{SizeExp2} \rangle & ::= & \langle \mathit{SizeExp2} \rangle \ \mathtt{*} \ \langle \mathit{SizeExp3} \rangle \\
& | & \langle \mathit{SizeExp3} \rangle \\
\\
\langle \mathit{SizeExp1} \rangle & ::= & \langle \mathit{SizeExp1} \rangle + \langle \mathit{SizeExp2} \rangle \\
& | & \langle \mathit{SizeExp1} \rangle - \langle \mathit{SizeExp2} \rangle \\
& | & \langle \mathit{SizeExp2} \rangle \\
\\
\langle \mathit{SizeExp} \rangle & ::= & \langle \mathit{SizeExp1} \rangle \\
\\
\langle \mathit{FunArg} \rangle & ::= & \langle \mathit{Exp} \rangle \\
\\
\langle \mathit{ListFunArg} \rangle & ::= & \epsilon \\
& | & \langle \mathit{FunArg} \rangle \\
& | & \langle \mathit{FunArg} \rangle \ \mathtt{,} \ \langle \mathit{ListFunArg} \rangle \\
\\
\langle \mathit{SizeFunArg} \rangle & ::= & \langle \mathit{SizeExp} \rangle \\
\\
\langle \mathit{ListSizeFunArg} \rangle & ::= & \epsilon \\
& | & \langle \mathit{SizeFunArg} \rangle \\
& | & \langle \mathit{SizeFunArg} \rangle \ \mathtt{,} \ \langle \mathit{ListSizeFunArg} \rangle \\
\\
\langle \mathit{ListTyp} \rangle & ::= & \epsilon \\
& | & \langle \mathit{Typ} \rangle \ \langle \mathit{ListTyp} \rangle
\end{array}
$$

Appendix C

SFDL Samples

This appendix chapter contains several example programs illustrating the Faerieplay SFDL language.

```
// -*- c -*-
/*
 * Circuit compiler for the Faerieplay hardware-assisted secure
 * computation project at Dartmouth College.
 *
 * Copyright (C) 2003-2007, Alexander Iliev <sasho@cs.dartmouth.edu> and
 * Sean W. Smith <sws@cs.dartmouth.edu>
 *
 * All rights reserved.
 *
 * This code is released under a BSD license.
 * Please see LICENSE.txt for the full license and disclaimers.
 *
 */

// The millionaire's problem: compare two numbers.
program Millionaires {

    const WordSize = 32;
    type Word = Int<WordSize>;

    function Boolean sfdlmain (Word a, Word b) {
      sfdlmain = a < b;
    }
}
```

Figure C.1: An SFDL "hello world" program: compare two numbers.

```
/*
 * Circuit compiler for the Faerieplay hardware-assisted secure
 * computation project at Dartmouth College.
 *
 * Copyright (C) 2003-2007, Alexander Iliev <sasho@cs.dartmouth.edu> and
 * Sean W. Smith <sws@cs.dartmouth.edu>
 *
 * All rights reserved.
 *
 * This code is released under a BSD license.
 * Please see LICENSE.txt for the full license and disclaimers.
 *
 */


program SimplestCond
{
    const WordSize = 32;
    type Word = Int<WordSize>;

    function Word sfdlmain (Word x, Word y)
    {
      print ("y = ", y);
      if (y != 0) {
        sfdlmain = x/y;
      }
      else {
        sfdlmain = x;
      }
    }

}
```

Figure C.2: Simple conditional in SFDL.

```
// -*- c -*-

/*
 * Circuit compiler for the Faerieplay hardware-assisted secure
 * computation project at Dartmouth College.
 *
 * Copyright (C) 2003-2007, Alexander Iliev <sasho@cs.dartmouth.edu> and
 * Sean W. Smith <sws@cs.dartmouth.edu>
 *
 * All rights reserved.
 *
 * This code is released under a BSD license.
 * Please see LICENSE.txt for the full license and disclaimers.
 *
 */


// correct answer, from a shell command (adjust x as needed)
/*
echo "x=3; (0 + 4*(2*x)) * (x^2)" | bc
*/

program TestLexicalScope {

    const WordSize = 32;
    type Word = Int<WordSize>;

    // correct result: (0 + 4*(2*x)) * (x^2)
    // possible incorrect result: (0 + 4*x) * (x^2)
    function Word sfdlmain (Word x) {
        var Word y, z;

        y = 0;
        z = x * 2;

        for (i = 0 to 1) {
          var Word x;
          x = z;

          for (j = 0 to 1) {
            // here we use the inner x
            y = y + x;
          }
        }

        for (i = 0 to 1) {
          // here use the outer x (param to main)
          y = y * x;
        }

        sfdlmain = y;
    }
}
```

Figure C.3: Demonstration of lexical scoping in SFDL.

```
/* -*- mode: c++; fill-column: 72; -*-
 */

program NestedLoops
{

    const WordSize = 32;
    type Word = Int<WordSize>;

    const N = 10;

    const X=9;
    const Y=7;

    // use addition and loops to achieve exponentiation: return X^Y.
    // note that we cannot have X and Y be (input) variables, as they
    // delimit the loops below, and so need to be static/constant.
    function Word sfdlmain () {

        var Word x;

        x = 1;

        for (i = 1 to Y) {
            //want:
            // x = x*X;
            // but do in a loop
            var Word x_1;
            x_1 = x;
            x = 0;
            for (j = 1 to X) {
                x = x + x_1;
            }

        }

        sfdlmain = x;

    }

}
```

Figure C.4: Loops in SFDL.

```
// -*- mode: c++; fill-column: 72; -*-
/*
 * Circuit compiler for the Faerieplay hardware-assisted secure
 * computation project at Dartmouth College.
 *
 * Copyright (C) 2003-2007, Alexander Iliev <sasho@cs.dartmouth.edu> and
 * Sean W. Smith <sws@cs.dartmouth.edu>
 *
 * All rights reserved.
 *
 * This code is released under a BSD license.
 * Please see LICENSE.txt for the full license and disclaimers.
 *
 */

program TestRef
{

    const WordSize = 32;
    type Word = Int<WordSize>;

    const REPEAT = 10;

    // Function with a reference parameter.
    // Need the dummy return value, as there is now no way to call a
    // void function.
    function Word accumulate (Word & accum, Word x) {
        accum = accum + x;
        accumulate = 0;
    }

    // result should be:
    // a + REPEAT*b
    // important point is that it's done via a reference parameter.
    function Word sfdlmain (Word a, Word b) {
        var Word accum;

        accum = a;

        for (i = 1 to REPEAT) {
            var Word dummy;
            // FIXME: we do not support statements which do not assign a
            // result, so need this dummy for now.
            dummy = accumulate (accum, b);
        }

        sfdlmain = accum;
    }

}
```

Figure C.5: References in SFDL.

The Language Fcpp

BNF-converter

April 12, 2009

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Fcpp

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String*⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Fcpp are the following:

```
Integer  bool    cast
const    else    enum
false    for     function
if       int     print
program  return  struct
to       true    type
typedef  var     void
```

The symbols used in Fcpp are the following:

```
(    )    ;
,    =    [
]    {    }
<    >    &
+=   -=   *=
/=   %=   ++
--   .    !
-    ~    *
/    %    +
<<   >>   <=
>=   ==   !=
^    |    &&
||
```

Comments

Single-line comments begin with `//`, `#`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of Fcpp

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Prog*⟩ ::= `program (` ⟨*Ident*⟩ `) ;` ⟨*ListDec*⟩

⟨*ListDec*⟩ ::= ϵ
| ⟨*Dec*⟩ ⟨*ListDec*⟩

⟨*ListStm*⟩ ::= ϵ
| ⟨*Stm*⟩ ⟨*ListStm*⟩

⟨*ListIdent*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `,` ⟨*ListIdent*⟩

⟨*Dec*⟩ ::= `const int` ⟨*Ident*⟩ = ⟨*Exp*⟩ `;`
| `typedef` ⟨*Typ*⟩ ⟨*Ident*⟩ `;`
| `var` ⟨*Typ*⟩ ⟨*Ident*⟩ `[` ⟨*Exp*⟩ `]` `;`
| `var` ⟨*Typ*⟩ ⟨*ListIdent*⟩ `;`
| `function` ⟨*Typ*⟩ ⟨*Ident*⟩ `(` ⟨*ListTypedName*⟩ `)` `{` ⟨*ListDec*⟩ ⟨*ListStm*⟩ `}`

⟨*Typ*⟩ ::= `int`
| `Integer` < ⟨*SizeExp*⟩ >
| `bool`
| `void`
| `struct` `{` ⟨*ListTypedName*⟩ `}`
| `enum` `{` ⟨*ListIdent*⟩ `}`
| ⟨*Typ*⟩ `[` ⟨*Exp*⟩ `]`
| ⟨*Typ*⟩ `&`
| ⟨*Ident*⟩

⟨*TypedName*⟩ ::= ⟨*Typ*⟩ ⟨*Ident*⟩

⟨*ListTypedName*⟩ ::= ϵ
| ⟨*TypedName*⟩
| ⟨*TypedName*⟩ `,` ⟨*ListTypedName*⟩

⟨*Stm*⟩ ::= `{` ⟨*ListDec*⟩ ⟨*ListStm*⟩ `}`
| ⟨*AssStm*⟩ `;`
| `return` ⟨*Exp*⟩ `;`
| `print` `(` ⟨*String*⟩ `,` ⟨*ListExp*⟩ `)` `;`
| `for` `(` ⟨*Ident*⟩ = ⟨*Exp*⟩ `;` ⟨*Exp*⟩ `;` ⟨*AssStm*⟩ `)` ⟨*Stm*⟩
| `if` `(` ⟨*Exp*⟩ `)` ⟨*Stm*⟩
| `if` `(` ⟨*Exp*⟩ `)` ⟨*Stm*⟩ `else` ⟨*Stm*⟩

⟨*ListExp*⟩ ::= ϵ
| ⟨*Exp*⟩
| ⟨*Exp*⟩ `,` ⟨*ListExp*⟩

⟨*AssStm*⟩ ::= ⟨*LVal*⟩ ⟨*AssOp*⟩ ⟨*Exp*⟩
| ⟨*LVal*⟩ ⟨*PostFixOp*⟩
| ⟨*PostFixOp*⟩ ⟨*LVal*⟩

⟨*AssOp*⟩ ::= =
| +=
| −=
| *=
| /=
| %=

⟨*PostFixOp*⟩ ::= ++
| −−

⟨*LVal*⟩ ::= ⟨*Exp*⟩

⟨*ConstExp*⟩ ::= ⟨*Exp*⟩

⟨*Exp13*⟩ ::= ⟨*Ident*⟩
| ⟨*Integer*⟩
| `true`
| `false`
| `(` ⟨*Exp*⟩ `)`

⟨*Exp12*⟩ ::= ⟨*Exp12*⟩ `[` ⟨*Exp*⟩ `]`
| ⟨*Exp12*⟩ `.` ⟨*Exp13*⟩
| ⟨*Ident*⟩ `(` ⟨*ListFunArg*⟩ `)`
| ⟨*Exp13*⟩

⟨*Exp11*⟩ ::= `!` ⟨*Exp12*⟩
| − ⟨*Exp12*⟩
| `~` ⟨*Exp12*⟩
| ⟨*Exp12*⟩

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ `*` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `/` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `%` ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp9*⟩ ::= ⟨*Exp9*⟩ + ⟨*Exp10*⟩
| ⟨*Exp9*⟩ − ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp8*⟩ ::= ⟨*Exp9*⟩ << ⟨*Exp9*⟩
| ⟨*Exp9*⟩ >> ⟨*Exp9*⟩
| ⟨*Exp9*⟩

⟨*Exp7*⟩ ::= ⟨*Exp8*⟩ < ⟨*Exp8*⟩
| ⟨*Exp8*⟩ > ⟨*Exp8*⟩
| ⟨*Exp8*⟩ <= ⟨*Exp8*⟩
| ⟨*Exp8*⟩ >= ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp6*⟩ ::= ⟨*Exp7*⟩ == ⟨*Exp7*⟩
| ⟨*Exp7*⟩ `!=` ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ `&` ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ `^` ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ `|` ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ `&&` ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp1*⟩ ::= ⟨*Exp1*⟩ `||` ⟨*Exp2*⟩
| ⟨*Exp2*⟩

⟨*Exp*⟩ ::= ⟨*Exp1*⟩

⟨*SizeExp3*⟩ ::= ⟨*Ident*⟩
| ⟨*Integer*⟩
| ⟨*Ident*⟩ `(` ⟨*ListSizeFunArg*⟩ `)`
| `(` ⟨*SizeExp*⟩ `)`

⟨*SizeExp2*⟩ ::= ⟨*SizeExp2*⟩ `*` ⟨*SizeExp3*⟩
| ⟨*SizeExp3*⟩

⟨*SizeExp1*⟩ ::= ⟨*SizeExp1*⟩ + ⟨*SizeExp2*⟩
| ⟨*SizeExp1*⟩ − ⟨*SizeExp2*⟩
| ⟨*SizeExp2*⟩

⟨*SizeExp*⟩ ::= ⟨*SizeExp1*⟩

⟨*SizeFunArg*⟩ ::= ⟨*SizeExp*⟩

⟨*ListSizeFunArg*⟩ ::= ϵ
| ⟨*SizeFunArg*⟩
| ⟨*SizeFunArg*⟩ `,` ⟨*ListSizeFunArg*⟩

⟨*FunArg*⟩ ::= ⟨*Exp*⟩

⟨*ListFunArg*⟩ ::= ϵ
| ⟨*FunArg*⟩
| ⟨*FunArg*⟩ `,` ⟨*ListFunArg*⟩

⟨*ListTyp*⟩ ::= ϵ
| ⟨*Typ*⟩ ⟨*ListTyp*⟩

Appendix E

Faerie C++ Samples

This appendix chapter contains several example programs illustrating the Faerie C++ language.

```
// -*- c++ -*-
// Faerie C++

// define empty macros for 'program', 'var' etc.
#include "fcpp-crutches.h"

// The millionaire's problem: compare two numbers.
program (Millionaires);

function bool sfdlmain (Integer<32> a, Integer<32> b) {
    return a < b;
}
```

Figure E.1: A "hello world" program: compare two numbers.

```
// -*- c++ -*-
// Faerie C++

// define empty macros for 'program', 'var' etc.
#include "fcpp-crutches.h"

// Test program with structures.

// this program is affected by two compiler bugs:
// - the field decl separator for structs is now a comma and not semicolon as in
// C/C++, and
// - GenHelper_C.gen_arg_parser can only handle int params to mainsfdl
program (Structs);

// Faerieplay compiler only supports this style of struct definition.
typedef struct {
    int x,
    int y
} point_t;

// the 'function' keyword is required
function int sfdlmain (int p1_x, int p1_y,
                       int p2_x, int p2_y)
{
    // the 'var' keyword is required
    var point_t p1, p2;
    p1.x = p1_x;
    p1.y = p1_y;
    p2.x = p2_x;
    p2.y = p2_y;

    return p1.x > p2.x && p1.y > p2.y;
}
```

Figure E.2: Program to illustrate struct definition and use.

```
// -*- c++ -*-
// Faerie C++

// define empty macros for 'program', 'var' etc.
#include "fcpp-crutches.h"

program (NestedLoops);

const int WordSize = 32;
typedef Integer<WordSize> Word;

const int X=9;
const int Y=7;

// use addition and loops to achieve exponentiation: return X^Y.
// note that we cannot have X and Y be (input) variables, as they
// delimit the loops below, and so need to be static/constant.
function Word sfdlmain () {

    // NOTE: declarations must be at the beginning, as in C.
    var Word accum;
    var int i;

    accum = 1;

    for (i = 1; i <= Y; ++i) {

        //want:
        // accum = accum*X;
        // but do in a loop
        var Word accum_snapshot;
        var int j;

        accum_snapshot = accum;
        accum = 0;
        for (j = 1; j <= X; ++j) {
            accum += accum_snapshot;
        }

    }

    return accum;
}
```

Figure E.3: Loops in Faerie C++.

The Language Json

BNF-converter

August 2, 2008

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Json

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String*⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + '.'⟨*digit*⟩ + ('e'-?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Json are the following:

```
false  null  true
```

The symbols used in Json are the following:

```
{   }   ,
=   [   ]
```

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of Json

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*TopLevelT*⟩ ::= ⟨*ListObjectT*⟩

⟨*ListObjectT*⟩ ::= ϵ
| ⟨*ObjectT*⟩ ⟨*ListObjectT*⟩

⟨*ObjectT*⟩ ::= { ⟨*ListAssocT*⟩ }

⟨*ListAssocT*⟩ ::= ϵ
| ⟨*AssocT*⟩
| ⟨*AssocT*⟩ , ⟨*ListAssocT*⟩

⟨*AssocT*⟩ ::= ⟨*Ident*⟩ = ⟨*Value*⟩

⟨*Value*⟩ ::= ⟨*SListT*⟩
| ⟨*ObjectT*⟩
| ⟨*Number*⟩
| ⟨*String*⟩
| ⟨*Literal*⟩

⟨*Number*⟩ ::= ⟨*Integer*⟩
| ⟨*Double*⟩

⟨*SListT*⟩ ::= [⟨*ListValue*⟩]

$$
\begin{array}{lcl}
\langle \mathit{ListValue} \rangle & ::= & \epsilon \\
& | & \langle \mathit{Value} \rangle \\
& | & \langle \mathit{Value} \rangle \texttt{ , } \langle \mathit{ListValue} \rangle \\
\langle \mathit{Literal} \rangle & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & \texttt{null}
\end{array}
$$

Bibliography

[1] Masayuki Abe and Fumitaka Hoshino. Remarks on mix-network based on permutation networks. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 317–334. Springer, 2001. 28

[2] Dakshi Agrawal, Selçuk Baktır, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using ic fingerprinting. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE Press, 2007. 92

[3] Rakesh Agrawal, Dmitri Asonov, and Ramakrishnan Srikant. Enabling sovereign information sharing using Web Services. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 873–877. ACM Press, 2004. 86

[4] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 86–97. ACM Press, 2003. 86

[5] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997. Versions exist for C and Java as well. 168

[6] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, May 2004. 65

[7] Dmitri Asonov and Johann-Christoph Freytag. Almost optimal private information retrieval. In R. Dingledine and P. Syverson, editors, *Privacy Enhancing Technologies*, pages 209–223, San Francisco, CA, April 2002. Springer. LNCS 2482. 24

[8] Dmitri Asonov and Johann-Christoph Freytag. Private information retrieval, optimal for users and secure coprocessors. Technical Report HUB-IB-159, Humboldt University, 10099 Berlin, Germany, June 2002. 39

[9] Mikhail Atallah, Marina Blanton, Vinayak Deshpande, Keith Frikken, Jiangtao Li, and Leroy Schwarz. Secure collaborative planning, forecasting, and replenishment (SCPFR). In *Proceedings of 10th INFORMS Conference on Manufacturing and Service Operations Management (MSOM)*, Evanston, Il, USA, June 2005. 86

[10] Mikhail J. Atallah and Wenliang Du. Secure multi-party computational geometry. In *WADS '01: Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 165–179, London, UK, 2001. Springer-Verlag. 86

[11] Kenneth Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Atlantic City, NJ, USA, April 1968. Thomson Book Company. 52

[12] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Jean-Francois Raymond. Breaking the $\mathrm{O}(n^{1/(2k1)})$ barrier for information-theoretic private information retrieval. In *Proc. of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 261–270, 2002. 23

[13] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the server's computation in private information retrieval: Pir with preprocessing. In M. Bellare, editor, *Crypto 2000*, volume 1880 of *LNCS*, pages 55–73. Springer-Verlag, 2000. 23

[14] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 547–557. Springer-Verlag LNCS 435, 1989. 73

[15] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. `http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html`, May 2005. URL live in April 2007. 79

[16] Mladen Berekovic, Helge Kloos, and Peter Pirsch. Hardware realization of a java virtual machine for high performance multimedia applications. *J. VLSI Signal Process. Syst.*, 22(1):31–43, 1999. 146

[17] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In *RSA Data Security Conference*, volume 1872 of *LNCS*. Springer, February 2002. 50

[18] Matt Blaze, Joan Feigenbaum, and Moni Naor. A formal treatment of remotely keyed encryption (extended abstract). In *EUROCRYPT '98*, volume 1403 of *LNCS*, pages 251–265, Espoo, Finland, May 1998. Springer-Verlag. 84

[19] Stefan Brands. *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy*. The MIT Press, August 2000. 39

[20] Stefan Brands. A technical overview of digital credentials. At `http://www.credentica.com/technology/technology.html`, Feb 2002. 39

[21] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM Press. 67

[22] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *WPES '07: Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, pages 21–30, New York, NY, USA, 2007. ACM. 39

[23] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Eurocrypt 1999*, Prague, Czech Republic, 1999. Springer-Verlag. LNCS 1592. 22, 23

[24] Jan Camenisch and Els Van Herreweghen. Design and implementation of the IDEMIX anonymous credential system. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 21–30, Washington, DC, USA, 2002. ACM Press. 39

[25] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, November 1998. 22

[26] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003*, volume 9 of *LNCS*, pages 188–207. Springer-Verlag, 2003. 134

[27] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms*, chapter 27. In [28], second edition, 2001. Problem 27-3 on permutation networks. 27

[28] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001. 54, 100, 221

[29] Intel Corp. Intel Trusted Execution Technology architectural overview. `http://www.intel.com/technology/security/downloads/arch-overview.pdf`, September 2006. 83

[30] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 2, Oakland, CA, USA, 2003. IEEE Computer Society. 81

[31] Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD, December 1985. This is better known as the "Orange Book," due to the color of the cover on the hardcopy. 63

[32] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983. 115

[33] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols (extended abstract). In *FOCS*, pages 350–357. IEEE, 1981. 115

[34] Marlena Erdos and Scott Cantor. Shibboleth architecture. Available from `http://shibboleth.internet2.edu/`, May 2002. Version 5. 17

[35] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28:637–647, 1985. 73

[36] Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. Privacy preserving mining of association rules. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–228, New York, NY, USA, 2002. ACM Press. 86

[37] Milan Fort, Felix C. Freiling, Lucia Draque Penso, Zinaida Benenson, and Dogan Kesdogan. TrustedPals: Secure multiparty computation implemented with smart cards. In *ESORICS 2006*, pages 34–48, Hamburg, Germany, September 2006. Springer. LNCS 4189. 82

[38] Keith B. Frikken and Mikhail J. Atallah. Privacy preserving route planning. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 8–15, New York, NY, USA, 2004. ACM Press. 86

[39] Simson Garfinkel. *Database Nation: The Death of Privacy in the 21st Century*. O'Reilly Media, Inc, 2001. 3

[40] Oded Goldreich. *Foundations of Cryptography Volume 2: Basic Applications*, chapter 7: General Cryptographic Protocols, pages 599–764. Cambridge University Press, May 2004. 78

[41] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. 13, 24, 39, 42, 43, 44, 46, 56, 68, 97, 109, 137, 138

[42] Philippe Golle. Revisiting the uniqueness of simple demographics in the US population. In *WPES '06: Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 77–80, Alexandria, Virginia, USA, 2006. ACM. 3, 19

[43] Mahadevan Gomathisankaran and Akhilesh Tyagi. Arc3d: A 3d obfuscation architecture. In *International Conference on High-Performance Embedded Architecture and Compilation: HiPEAC 2005*, pages 184–199. Springer-Verlag Berlin, nov 2005. LNCS 3793. 84

[44] Mahadevan Gomathisankaran and Akhilesh Tyagi. Architecture support for 3d obfuscation. *IEEE Trans. Comput.*, 55(5):497–507, 2006. Member-Akhilesh Tyagi. 84

[45] John Hennessy and David Patterson. *Computer Organization and Design: The Hardware/Software Interface*, appendix A: Assemblers, Linkers and the SPIM Simulator. Morgan Kaufmann, 3rd edition, 2004. Available at `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`. 181

[46] Alex Iliev and Sean Smith. Privacy-enhanced directory services. In *2nd Annual PKI Research Workshop*, pages 109–121, Gaithersburg, MD, April 2003. NIST. iv, 15

[47] Alexander Iliev and Sean Smith. Private information storage with logarithmic-space secure hardware. In *I-NetSec '04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pages 201–216, Toulouse, France, August 2004. IFIP, Kluwer. iv, 41, 55

[48] Alexander Iliev and Sean Smith. Protecting client privacy with trusted computing at the server: Two case studies. *IEEE Security and Privacy*, 3(2):20–28, March 2005. iv

[49] Bob Jenkins. Minimal perfect hashing. `http://burtleburtle.net/bob/hash/perfect.html`, 2003. 27

[50] Franz Kafka. *The Trial (Der Process)*. Verlag Die Schmiede, Berlin, 1925. 2

[51] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–8, Berkeley, CA, USA, 2008. USENIX Association. 92

[52] Thomas F. Knight. Implementation of a list processing machine. Master's thesis, MIT, 1979. advisor: Marvin L. Minsky. 146

[53] Sherri A. Koenig. Power generation scheduling by Lagrangian relaxation. Master's thesis, Cornell University, 1975. 104

[54] Yaping Li, J. D. Tygar, and Joseph M. Hellerstein. Private matching. In D. Lee, S. Shieh, and J. D. Tygar, editors, *Computer Security in the 21st Century*. Springer, June 2005. To appear. 86

[55] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS 2000*, pages 168–177, November 2000. 83

[56] Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. *J. Cryptology*, 22(2):161–188, 2009. 74

[57] M. Luby and C. Rackoff. How to construct pseudo-random permutations from pseudo-random functions. *SIAM Journal on Computing*, 17(2):373–386, 1988. 48

[58] Stefan Lucks. Faster luby-rackoff ciphers. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 189–203, London, UK, 1996. Springer-Verlag. 49

[59] Philip MacKenzie, Alina Oprea, and Michael K. Reiter. Automatic generation of two-party computations. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 210–219, New York, NY, USA, 2003. ACM Press. 82

[60] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, pages 287–302. USENIX, August 2004. 13, 77

[61] N. Modadugu, D. Boneh, and M. Kim. Generating RSA keys on the PalmPilot with the help of an untrusted server. In *RSA Data Security Conference and Expo*, 2000. 84

[62] John A. Muckstadt and Sherri A. Koenig. An application of Lagrangian relaxation to scheduling in power-generation systems. *Operations Research*, 25(3):387–403, May 1977. 104

[63] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, San Antonio, Texas, United States, 1999. ACM Press, New York, USA. 85, 90, 194

[64] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 590–599, New York, NY, USA, 2001. ACM Press. 85, 95

[65] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139, New York, NY, USA, 1999. ACM. 73

[66] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology*, 12(1):29–66, 1999. 49

[67] Janus Dam Nielsen and Michael I. Schwartzbach. A high-level dsl for secure multiparty computation. Linked at `http://www.brics.dk/~mis/papers.html`, 2006. 82

[68] Robert O'Harrow. *No Place to Hide*. Free Press, 2005. 3

[69] Rafail Ostrovsky and Victor Shoup. Private information storage. In *ACM Symposium on Theory of Computing*. ACM, 1997. 47

[70] Jacques Patarin. Luby-Rackoff: 7 rounds are enough for $2^{n(1-\varepsilon)}$ security. In *Advances in Cryptology–CRYPTO 2003*, pages 513–529. Springer-Verlag, Oct 2003. 49

[71] Benny Pinkas. Fair secure two-party computation. In *Advances in Cryptology—Eurocrypt '2003*, volume 2656 of *LNCS*, pages 87–105. Springer-Verlag, May 2003. 77

[72] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard, 1981. 73

[73] A. Sadeghi and C. Stuble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop*, September 2004. 67

[74] Len Sassaman, Bram Cohen, and Nick Mathewson. The Pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES '05: Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pages 1–9, New York, NY, USA, 2005. ACM Press. 81

[75] Bruce Schneier and John Kelsey. Unbalanced Feistel networks and block cipher design. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 121–144, London, UK, 1996. Springer-Verlag. 49

[76] Shibboleth® federated Single Sign-on software. http://shibboleth.internet2.edu/. 17

[77] Sean Smith. Outbound authentication for programmable secure coprocessors. In *7th European Symposium on Research in Computer Science*, October 2002. 21, 65

[78] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999. 29

[79] S.W. Smith and D. Safford. Practical server privacy using secure coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001. (Special Issue on End-to-End Security). 23

[80] Daniel Solove. Privacy and power: Computer databases and metaphors for information privacy. *Stanford Law Review*, 53:1393–1462, 2001. Currently (Aug 2008) available at http://ssrn.com/abstract=248300. 2

[81] Evan R. Sparks. A security assessment of Trusted Platform Modules. Technical Report TR2007-597, Dartmouth College, Computer Science, Hanover, NH, June 2007. http://www.cs.dartmouth.edu/reports/abstracts/TR2007-597/. 72

[82] National Institute Of Standards and Technology. Security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-1/fips1401.pdf, Jan 1994. FIPS PUB 140-1; URL current in June 2005. 29, 65

[83] National Institute Of Standards and Technology. Security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, May 2001. FIPS PUB 140-2; URL current in June 2005. 65

[84] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ISCA 2005*, Madison, WI, USA, June 2005. 83

[85] Latanya Sweeney. Uniqueness of simple demographics in the U.S. population. Technical Report LIDAP-WP4, Carnegie Mellon University, Laboratory for International Data Privacy, Pittsburgh, PA, 2000. http://privacy.cs.cmu.edu/dataprivacy/papers/LIDAP-WP4abstract.html. 3, 19

[86] Trusted Computing Group. https://www.trustedcomputinggroup.org/home, May 2005. 67

[87] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable anonymous credentials: Blocking misbehaving users without TTPs. In *ACM CCS*, pages 72–81, 2007. 39, 103

[88] Eli Upfal. A permutation network. http://web.archive.org/web/20060919091912/www.cs.brown.edu/courses/cs253/slide/class2.ps, 2000. Course Lecture Notes: Brown University CS 253: Design and Analysis of Communication Networks. 28

[89] Abraham Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, Jan 1968. 27, 28, 44

[90] S. Wang, X. Ding, R. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS 2006*, September 2006. LNCS 4189. 55, 80, 137, 138

[91] Peter Williams and Radu Sion. Usable PIR. In *NDSS 2008*, February 2008. 39, 47, 81, 195

[92] Peter Winkler. Electronic trusted party. US Patent 5117358, 1992. 83

[93] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998. 104

[94] A. C. Yao. How to generate and exchange secrets. In *FOCS 1986*, pages 162–167. IEEE, 1986. 73

[95] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994. 21, 65

[96] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In W. Fumy, editor, *Advances in Cryptology–EUROCRYPT '97*, volume 1233 of *LNCS*, pages 62–74, 1997. 92

[97] Adam Young and Moti Yung. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley & Sons, 2004. 92

[98] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: secure program partitioning. *SIGOPS Oper. Syst. Rev.*, 35(5):1–14, December 2001. 88

[99] Tao Zhang, Santosh Pande, and Antonio Valverde. Tamper-resistant whole program partitioning. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Languages, compilers, and tools for embedded systems*, pages 209–219, New York, NY, USA, 2003. ACM Press. 85

[100] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 72–84, Boston, MA, USA, 2004. ACM Press. 84

Index

Glossary

array gate *Gate which abstracts secure and efficient indirect array access in a circuit* Each array access is represented in the circuit by just one array gate.

array representation *The state of an indirect array in a circuit machine implementation* The CVM is free to implement indirect arrays as it chooses; the CF only specifies (opaque) references to arrays, which are passed into and out of array gates. The array representation is the actual state used by the CVM to store arrays.

Container *An abstraction of an array on the host* The host provides the TTP with services like reading and writing values to elements of the container; creating, moving and deleting containers; and a read/write container header. The TTP must provide for all privacy and integrity, eg. by encrypting and MAC-ing all values written to a container—it can assume that the adversary controls all containers; see Section 10.3.2.

Graphviz *Graph visualization program.* Produces a static visualization of a graph, in multiple formats and with a choice of formatting algorithms. Our circuit compiler supports exporing a circuit in the Graphviz format. Open-source. See `http://www.graphviz.org/`.

Indirect array *Array access where the index is dynamic* An array, or more precisely an array access, where the index is a variable. Thus, one access to RAM is required to obtain the index, and another access to obtain or update the array element. In a direct array access, the index is a constant, and does not itself require a RAM access.

Intermediate Format *abstract program representation used by a compiler* The IF is a format which is ideally independent of both the concrete syntax (language) and of the executable format (eg. MIPS assembly) being targeted.

switch bit *The control bit for a switch in a Beneš network* If set, the switch crosses its inputs.

uDraw(Graph) *Graph visualization program.* Allows some manipulation of the graphs, and is quite useful for interactively exploring a graph, for example following all the parents of a given node. The Faerieplay circuit compiler supports exporing a circuit in the uDraw-Graph format. Not open-source. See http://www.informatik.uni-bremen.de/uDrawGraph/en/uDrawGraph/.

List of Acronyms

AA . . . . . Attribute Authority

CBC . . . . . Cipher Block Chaining A common encryption **mode of operation!**, which chains adjacent ciphertext blocks using an XOR operation. The first block does not come from the plaintext, and is called an Initialization Vector (IV).

COTS . . . . . Commercial off the Shelf

CPA . . . . . chosen-plaintext attack An attack against an encryption scheme where the adversary can learn the encryption of plaintexts of his choosing.

CVM . . . . . Circuit Virtual Machine The virtual machine which evaluates Faerieplay circuits on particular inputs. The CVM is designed to run on a tiny hardware TTP, assisted by a standard untrusted host for storage of bulk data.

HS . . . . . Handle Service A **shibboleth!** component which generates opaque handles for users.

ISO . . . . . Independent System Operator ISOs operate most of the US electric power grid.

IV . . . . . Initialization Vector The block of bits XOR-ed with the initial ciphertext block in CBC encryption.

LDAP . . . . . Lightweight Directory Access Protocol

LHS . . . . . Left-hand side The left-hand side of an assignment statement. This is usually a variable whose value is modified by the assignement.

MAC . . . . . message authentication code Message authentication code.

PPIR . . . . . Practical Private Information Retrieval

RHS . . . . . Right-hand side The right-hand side of an assignment statement. This is usually an expression whose value is to be assigned to the variable at the LHS.

SCOP . . . . . Secure coprocessor A computer with physical protections designed to protect it from observation or tampering by a motivated adversary with physical access. Usually attached to a host computer. The IBM 4758 is an important example.

SCS . . . . . Secure Computation Server The computation server of Faerieplay, which receives users' inputs, performs the computation securely, and sends the results back.

SHAR . . . . . Shibboleth Attribute Requester **shibboleth!** target site component which requests attributes about a user requesting access.

SHIRE . . . . . Shibboleth Indexical Reference Establisher

SSO . . . . . Single sign-on An authentication system which stores a user's identity information in order to avoid prompting the user for credentials (like a password) repeatedly.

TCB . . . . . trusted computing base The set of components in a computer system which must be trusted to enforce the system's (or a user's) security policy.