UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Computer Science Curriculum

Tiit Pikma

Auditing of Secure Multiparty Computations

Master's Thesis (30 ECTS)

Supervisors: Jan Willemson, PhD
Sven Laur, PhD

Tartu 2014

Auditing of Secure Multiparty Computations

Abstract:

Secure multiparty computations allow independent parties to collectively analyze data without compromising their input's privacy. This data secrecy is guaranteed in some security model: in the passive model an adversary can only look at data visible to it, while in the active model adversaries can actively interfere in the computations. So from a security standpoint the active model is preferable, but carries a significant overhead.

This thesis proposes an intermediate model, which builds upon a system that is secure in the passive model, but audits the computations for active attacks. It carries the same guarantees provided by the passive model, while attempting to detect active attacks without being explicitly secure against them.

To facilitate auditing, systems produce audit logs, which can be examined to detect active malicious behaviour. Audit logging was added to Sharemind, an existing secure multiparty computation platform that operates in the passive model, and a prototype audit tool was created to make inspecting the logs during the auditing process accessible to human auditors with only basic knowledge of secure multiparty computations. To test the viability of this model, an Internet voting demo application was created using this modified Sharemind and possible attacks against this application were analyzed from an auditability standpoint.

The analysis unveiled that since the Sharemind audit logging implementation evaluates protocols without context, it is possible to automatically verify individual protocols and detect computational forgery, but the adversary can perform semantic forgery. The audit logs do not contain the semantics of the computation and this can be abused to modify inputs to protocols, effectively modifying the results. The need to include semantic information about the computation in the audit became apparent and is set as a goal for future work.

Keywords:

Auditing, secure multiparty computations, Sharemind, prototype.

Turvaliste ühisarvutuste auditeerimine

Lühikokkuvõte:

Turvalised ühisarvutused lubavad sõltumatutel osapooltel ühiseid andmeid analüüsida ilma sisendi privaatsust rikkumata. Andmete turvalisus tagatakse mõnes turvamudelis: passiivses mudelis saab vastane ainult vaadelda väärtuseid, mis talle nähtavaks tehakse, samal ajal kui aktiivses mudelis saab vastane aktiivselt ar-

vutustesse sekkuda. Seega turvalisuse seisukohalt on aktiivne mudel eelistatum, ent sellega kaasneb märkimisväärne arvutuslik lisakulu.

See lõputöö pakub välja uue vahepealse turvamudeli, mis võtab aluseks passiivses mudelis turvalise süsteemi ning auditeerib selle arvutusi leidmaks aktiivseid ründeid. Sellel mudelil on samad turvagarantiid mis passiivsel mudelil, kuid lisaks üritatakse tuvastada aktiivseid ründeid nende vastu ilmutatult kaitstud olemata.

Auditeerimise võimaldamiseks kirjutavad süsteemid auditlogisid, mida saab uurida aktiivse kuritahtliku tegevuse tuvastamiseks. Töö käigus lisati auditlogimine Sharemindi, olemasolevale turvaliste ühisarvutuste platvormile, mis töötab passiivses mudelis. Lisaks loodi auditi tööriista prototüüp, mis võimaldab inimaudiitoritel, kellel on ainult pinnapealne tutvus ühisarvutustega, logisid uurida. Väljapakutud mudeli testimiseks loodi valimiste demorakendus Sharemindis ja analüüsiti võimalikke ründeid selle rakenduse vastu auditeeritavuse vaatepunktist.

Analüüs tuvastas, et kuna Sharemindi auditeerimise implementatsioonis vaadeldakse protokolle ilma kontekstita, siis on võimalik individuaalsete protokollide automaatne verifitseerimine ja arvutusliku pettuse tuvastamine, aga vastane saab läbi viia n-ö semantilist võltsimist. Auditlogid ei sisalda semantilist informatsiooni arvutuste kohta ja seda saab ära kasutada protkollide sisendite muutmiseks ning selle kaudu arvutuste tulemuste muutmiseks. Selgus vajadus semantilise informatsiooni kaasamiseks auditeerimise protsessi ning see seati tulevaseks uurimissuunaks.

Võtmesõnad:

Auditeerimine, turvalised ühisarvutused, Sharemind, prototüüp.

Contents

Introduction

Many organizations have collected large amounts of data and wish to analyze it with regards to information that other, independent organizations hold. However, none of them wish to share their data with others due to business decisions, regulations, etc. There exists a need to collectively process data while keeping it secret. One possible solution for this problem is secure multiparty computation.

Secure multiparty computation is a cryptographic method that allows multiple independent parties to collectively analyze data they hold while keeping each party's private data secret. For this, a sharing scheme is applied, which shares the input parties' private data between multiple computing parties that do the analysis without a single one seeing the whole input.

However, since computations involve valuable private data, there is the risk that some parties attempt to gain unauthorized access to others' private input. Multiparty computations are observed in a security model, which describes the capabilities of the adversary trying to steal the data. If the described adversary is unable to attack the system, then the system is said to be secure in that model.

Two popular examples of security models are the passive and active model, but both have their drawbacks. A problem with the passive model is that the adversary it describes has some restrictions which might not apply in the real world, ignoring possible active attacks and increasing the risk of data leaking. The active model covers both passive and active attacks, but implementing security in it carries a significant computational and organizational overhead.

This thesis proposes an alternative intermediate security model which augments the passive model with human auditors. After a system which is secure in the passive model has finished computation, the actions of all computing parties are audited for active malicious behavior. If none of the audits detect anything irregular, then that decreases the risk of there having been any active attacks and increases the probability that no private data was leaked, adding only minimal additional overhead to the computation.

The thesis is based on the Sharemind multiparty computation platform, which operates in the passive model. Therefore the thesis uses the sharing scheme and computation protocols used in Sharemind. An additional issue with Sharemind in

particular is the fact that it is closed-source: essentially a black box. Therefore all parties using this system need to trust its implementation and that it does not leak any private data to either other parties or the implementers of Sharemind. Auditing the computations also verifies the operation of the software and attempts to raise trust in Sharemind.

The contribution of this thesis is the proposal of a new audited security model, implementation of components necessary for auditing Sharemind applications, creation of a prototype audit tool to ease auditing, an example electronic voting application in Sharemind to audit, and analysis of this example application to evaluate the viability of the new proposed model.

Chapter 1 gives some background on secure multiparty computations, secret sharing, active and passive security models, and Sharemind. Chapter 2 proposes the new audited security model and details the necessary components of it. Chapter 3 introduces the prototype audit tool that was created to help human auditors. Chapter 4 describes an Internet voting application that demonstrates the audited model, and analyzes possible attacks against it and how auditing helps to detect those attacks. Chapter 5 draws conclusions. Appendix A presents the format of files used to store data necessary for auditing. Appendix B references source code of the prototype tool.

Chapter 1

Background

1.1 Secure multiparty computation

Secure multiparty computation [1] is a secure computation method for processing private data using several parties. Data secrecy is based on *secret sharing* [2]: the data being processed is divided into separate opaque *shares*, which seem random and are given to independent parties to hold. A k-out-of-n secret sharing scheme divides the private value into n shares and the original value can only be reconstructed from at least k shares.

However unlike with regular (non-homomorphic) encryption, these shares can still be used to compute some functions on secret values—this is the principle behind secure multiparty computations. Each party uses their shares as input, applies some algorithm to them, optionally communicates with the other parties, and as a result is left with a single share of the function's output. Combining the three result shares produces the output of the function without the parties knowing the actual input values.

This thesis refers to those functions as *protocols* and a number of them—including addition, multiplication and declassification, which are used in the following chapters—are given in [3]. These primitive protocols can be composed to construct more elaborate protocols.

1.2 Security models

When providing private data for secure multiparty computation, none of the parties can be sure that all other parties are honest and will not attempt to discover their secret information. It could also be that the software implementation of the protocols is intentionally backdoored or just has unintentional security flaws.

Whatever the situation, all possible adverse behaviour is said to be controlled by the *adversary*.

Because of this, secure multiparty computation systems are usually designed to be secure in a specific *security model*. The two most popular examples are *passive* and *active* models.

In the passive security model, the adversary is considered to be "honest, but curious", meaning that it will try to read as much data as it can and try to learn information about the secret values, but it will not intervene with the protocols. It will not modify any of the values, block communications, inject messages, etc.

In the active security model, the adversary is dishonest and does everything in it's power to attack the system. This can include taking control of one or all other parties, manipulating the network, etc.

1.3 Sharemind

Sharemind [4] is a complete solution for building practical data processing applications that use secure multiparty computations. *Sharemind Application Server* is the latest iteration of Sharemind. Although it supports multiple sharing schemes and security models, this thesis will look at the additive three party sharing scheme in the passive model. Protocols implemented in this configuration are secure if at most one computing party is under the control of a passive adversary.

The *additive secret sharing scheme* with three parties is a 3-out-of-3 scheme, where the private data is encoded as unsigned integers with fixed bit-length, e.g. 32-bit unsigned integers. Each integer is then additively shared using the following algorithm:

$$\begin{aligned} s_1 &\leftarrow \mathbb{Z}_{2^n} \\ s_2 &\leftarrow \mathbb{Z}_{2^n} \\ s_3 &= s - s_1 - s_2 \pmod{2^n}, \end{aligned}$$

where $x \leftarrow M$ denotes that x is a uniformly random value from the set M and n is the bit-length of the integer type.

The values s_1, s_2 and s_3 are the shares which will be given to three parties. These shares seem random to the holders, because s_1 and s_2 are uniformly randomly sampled, and s_3 is a value from which two random values have been subtracted. The original secret value can be retrieved by adding the shares together modulo 2^n:

$$s = s_1 + s_2 + s_3 \pmod{2^n}.$$

Having less than all three of the shares leaks no information about the original value.

Sharemind applications are written in *SecreC* [5], a programming language which is compiled into Sharemind executables containing bytecode for a low-level assembly language.

Two large components of Sharemind's framework are the *miners* and the *controller*. The miners are computing nodes, each held by a single party, and are responsible for interpreting the bytecode and executing the computation protocols. The controller is the node that instructs the miners which executable to load, passes them input parameters, and presents the results.

Among other controller types, Sharemind Application Server features a web controller module for Node.js [6]. It allows controlling miners from a Node.js web server in order to create web applications leveraging secure multiparty computations. This will be used in Chapter 4.

Chapter 2

Audited security model

2.1 Motivation

The *audited security model* is an attempt to augment the passive security model in order to introduce a new pseudo-model as a middle-ground between the passive and active models. It allows for the system to operate in the passive security model, but produces information that can be audited after the fact to detect any active malicious activity. So it does not prevent active attacks, but allows to detect them.

The audited security model should provide better guarantees than the passive model, but not as good as the ones in the active model. On the other hand, security in the active model is complicated to achieve and carries significant overhead—both computational and organizational—, while the audited model is essentially the passive model with an extra layer of verifiability on top, protecting against some real-world attacks that are not covered by the passive model.

This thesis explores viability of the audited security model on Sharemind.

2.2 Audit logs and party views

A key component of the audited security model are *audit logs*. These are files generated by the parties of the secure multiparty computation containing details about the computation protocols executed by the parties. The logs state what computation protocols were executed, with what input values, what random values were generated, what messages were sent and received, which party was the recipient or sender, and what were the results of the computations.

This information is later used to verify that all parties behaved correctly during the protocols and applied no detectable active attacks. Any protection against

eavesdropping is already guaranteed by the passive model that Sharemind operates in.

These audit logs are not public, as when the adversary has access to all three audit logs, it can just combine the shares contained in them and retrieve the secret shared values. So these log files must provide some value individually.

A *party's view* is the set of all messages it has sent to or received from remote computing parties during the execution of a protocol. This view is generated based on the information contained in the audit logs. These are compared between the parties to ensure that all messages that were sent were received unmodified or to detect cases where a party lies about sent or received messages.

Party views cannot be made public either, because seeing all communication between the three nodes leaks information about the shared values. For example the multiplication protocol [3] has parties passing their shares to the next one. So all three shares of a value can be combined from just two parties' views (as each party sees two shares).

2.3 Human auditors

The audit logs and party views cannot be made public, but still someone needs to check them for malicious activities. For this, *human auditors* are used. Three human auditors, independent of each other and independent of the computing parties, are each given access to the audit log of a single party. It is the auditors' job to verify that, based on the audit logs, all parties behaved correctly.

The first task is to make sure that all protocol computations are correct. To help with this, they use an audit tool which simulates secure multiparty computation protocols for them. A prototype of such an audit tool will be introduced in Chapter 3.

The next step is to ensure that the party views match. The views cannot be directly compared as then the auditors would see all shares. But since it is only necessary for them to match, hashes of the views can be compared instead, so as to not leak any information. Ensuring the views match guarantees that no party can lie about sent or received messages (given that they are not colluding with another party).

The last step is to ensure that all parties adhered to the algorithm agreed upon in the form of an executable program. In Sharemind's context, this program is a SecreC script compiled into a Sharemind binary executable. This means that in addition to verifying that the protocol computations were correct and all sent and received messages match other parties views, it is necessary to assert that the party handled inputs, branching, intermediate results, etc. correctly.

If all these checks pass, it mitigates the possibility that an active adversary

modified the program and changed the computation process. This gives additional guarantees to Sharemind's additive three party passive domain, which normally operates in only the passive security model.

2.4 Audit logging for Sharemind

Since the Sharemind Application Server did not produce audit logs, it was necessary to add this feature. A very simplistic method for generating audit logs using log4cpp [7] was added to a small set of protocols—specifically addition, subtraction, summation, multiplication, and declassification. The source code of these modifications is not public as Sharemind is a closed source system and including the modifications is not essential for this thesis.

The log is structured as an Extensible Markup Language (XML) file, whose schema can be found in Appendix A. Sharemind parallelizes computations to save on network overhead, so all values are given in vectors, even if they are actually scalars.

The logs contain information about each invoked protocol separately, excluding information about how these protocols relate to each other and all other program semantics. Sharemind programs start out as SecreC scripts, which are compiled to a low-level assembly language stored in bytecode form in an executable, which in turn is later interpreted by a virtual machine that invokes the necessary protocols[4]. Including semantic information from the program in the audit logs would involve modifications to all these low-level components and to the Sharemind executable file format, which would have to store this extra information. This was not done in this thesis due to time and resource constraints: the implications of this are discussed in Section 3.4.

Chapter 3

Auditing tool prototype

3.1 Motivation and goals

Parsing the audit logs generated by computing parties manually is practically unfeasible. In order to help human auditors to review the log files, a prototype of an *auditing tool* was created. The aim of the tool is to parse the audit logs and present them in a more user-friendly way. In addition, it should automatically verify the computational correctness of protocols contained in the logs to reduce the workload of the human auditor.

A specific goal of the audit tool (be it a prototype or not) is to have public source code. This allows everybody to check the implementation of the automatic log verification and potentially increases trust in the tool. Alternatively, it can be used to aid in the creation of an independent audit tool.

The source code is referenced in Appendix B.

3.2 Application structure

The prototype auditing tool (nicknamed "Sharemind Player" or "SMPlayer") is a Python 3 application using the Kivy framework [8] for the graphical user interface.[1] It consists of the following Python packages:

smplayer The main Python package of the auditing tool, providing the graphical application intended to be used by auditors. It contains the description of the application's main window in KV language [9] and the logic for graphically selecting log files to load from the file system.

[1] There is also a non-interactive command-line interface, which just outputs the log's view and whether the automatic verification succeeded or not. This can be used to automatically process large batches of log files.

smplayer.widgets Contains the actual widgets used to construct the user interface. The main (and only exported) widget creates a tree view of header nodes, each corresponding to a single protocol in the audit log.

The header displays a short summary of the protocol it represents: the name of the protocol, what were the inputs, what was the logged output, and if it passes automatic verification (additionally displaying the simulation result if it does not). For simple, local protocols (e.g. addition) this is sufficient, as there are no extra details to display.

In case of more complex protocols that require communication between the parties (e.g. multiplication), the header node also contains a body subnode. The body displays a detailed description of the protocol, using the actual values in the audit log to show step-by-step computation.

smplayer.core Contains the core functionality of log parsing, automatically verifying the logs, and generating the party views.

Log parsing is done using the standard Python ElementTree XML API "xml.etree.ElementTree" [10]. Each element parsed is used to initialize a protocol instance.

Automatically verifying the logs is done by simulating all the protocol instances parsed from the logs. If all simulations yield the same results (output and sent messages) as in the audit logs, then automatic verification succeeds.

Generating party views is done by grouping all sent messages by recipient and hashing them together. The same is done for all received messages.

smplayer.core.protocols Contains the protocol classes and code for simulating protocol instances. More details on protocol simulation are given in Section 3.3.

`smplayer.core` and `smplayer.core.protocols` are independent of the other packages (i.e. the GUI) and can be used separately. An example of this is the command-line interface, which shares no code with the main graphical application except for those two packages.

The prototype uses Python's standard `distutils` package [11] for simple distribution and installation. It also includes a simple custom test command for running the included unit tests.

A screenshot of the audit tool can be seen in Figure 3.1.

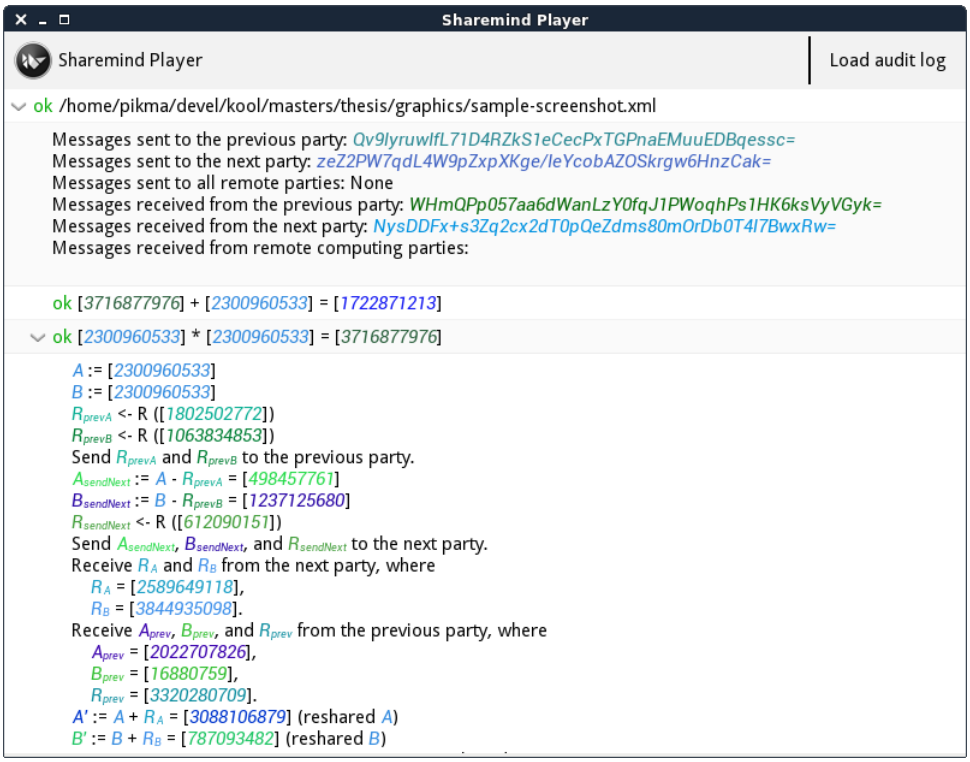

Figure 3.1: The Sharemind audit tool prototype displaying an addition and a multiplication protocol. The prototype actually uses Kivy's default color scheme of light colors on a black background, but a custom light color scheme is used for the screenshots to improve printability.

3.3 Protocol simulation

As mentioned, the audit tool simulates the protocols contained in the audit logs to automatically verify computational correctness. This is done by reimplementing all supported Sharemind protocols in Python and invoking them with the values read from the log file instead of communicating with external parties. All protocol results are cached, so a protocol instance is simulated at most once.

For local protocols, this is simple: read the input to the protocol from the log file, compute all the steps of the protocol, and check whether the output of the simulation matches the value recorded in the log file. The currently implemented local protocols in the tool are addition, subtraction, and summing elements in a vector. These are all extremely trivial protocols, as in the additively shared scheme these are just adding or subtracting with a modulus.[2]

Simulating protocols involving communication between the parties and generating random values are more complex. In addition to input and output of the protocol, the audit log needs to contain all sent and received messages. For the currently implemented non-local protocols (multiplication and declassification) all generated random values are also sent to another party, so they can be extracted from the messages. For other protocols, some generated random values do not appear unmodified in sent messages and need to be logged separately.

The tool simulates a non-local protocol using the input and messages received from other computing parties to produce the output of the protocol and all messages sent to other parties. These are compared to the values in the log file to automatically verify computation.

An explicit goal for the Python reimplementations was to be simple and understandable. This way the source code of the reimplementation itself can be verified by a larger audience. To achieve this, the code is written in a high-level, functional style made possible by Python. It is possible to draw direct parallels between the protocol source code and algorithm descriptions in published papers (e.g. [3]).

3.4 Program semantics

The current format of the log file contains only the values used in individual protocol, excluding context and program semantics (see Section 2.4). This means that looking at the values input to a protocol without additional information, it is not possible to determine if this value is an external input to the program, the output of a previous protocol, or simply a constant. It is also difficult to map lines

[2]Currently all simulations in the audit tool are made modulo 2^{32}, but mechanisms are in place to simply change this either globally or per protocol instance.

from the audit log to expressions in the original program (in case of Sharemind, the SecreC script).

As a result of this, it is not possible for the automatic log verification to check if the output value of one protocol, which is supposed to be used as the input to another one, is done so without modifications. Take for example the simple expression

$$a + b - c,$$

where a, b, and c are secret shared scalar values. Evaluating this expression would create the following two audit log entries:

```
<add>
  <input>
    <vector>
      <value>
        <!-- share of a -->
      </value>
    </vector>
    <vector>
      <value>
        <!-- share of b -->
      </value>
    </vector>
  </input>
  <output>
    <vector>
      <value>
        <!-- share of a + b -->
      </value>
    </vector>
  </output>
</add>
```

and

```
<sub>
  <input>
    <vector>
      <value>
        <!-- share of a + b -->
      </value>
    </vector>
    <vector>
      <value>
        <!-- share of c -->
      </value>
    </vector>
  </input>
  <output>
```

```
    <vector>
      <value>
        <!-- share of a + b - c -->
      </value>
    </vector>
  </output>
</sub>
```

While automatic log verification can check that the individual protocols are indeed correct, it does not know to how check that the output value of the addition protocol must match the first input value of the subtraction protocol, or specifically, that `./add/output/vector/value` must match `./sub/input/vector[1]/value`. This enables semantic forgery, which will be discussed in Section 4.4.4.

Due to the same problem, one cannot distinguish external inputs to the program from values produced internally, which means the inputs cannot be included in a party's view and input correctness is not handled. This enables input forgery, which will be discussed in Section 4.4.3.

Therefore complete correctness of the log in its current format cannot be automatically analyzed and needs to be verified by a human auditor. Yet this is only viable for simple programs and small amounts of data, as all relations would need to be meticulously checked. The effects of this will be explored more closely in Section 4.4.

An alternative to including semantic information in the log file directly would be to give the audit tool access to the SecreC program that was compiled or the bytecode that was executed and let it extract semantic information from it. However, this would entail reimplementing the parsers for SecreC or the assembly language, which is arguably a larger task than modifying existing low-level components in Sharemind. An idea to simplify this would be to modify the SecreC compiler such that it writes the abstract syntax tree of the program to a file, and the audit tool parses this instead: this avoid the need for the audit tool to understand SecreC. Figuring out the best solution and implementing it is future work.

Chapter 4

Internet voting example

4.1 Overview

To provide a practical test scenario for the audited security model and audit tool prototype, a simple demonstration of Internet voting on the Sharemind Application Server was created. Key functions of this application are audited and checked for active attacks.

A web page presents voters with multiple candidates grouped into parties and they have the chance to cast a vote for a single candidate (see Figure 4.1). Parties are there just for information and don't affect results: votes are tallied per candidate. A separate web page presents the tally of votes cast so far (see Figure 4.2). Authenticating voters is not covered in this example as that can be achieved with external means.

The candidates on the web pages are not hard-coded, but are generated from a JavaScript Object Notation (JSON) file containing information about the candidates. The functions used to check if a ballot is valid and to calculate election results generate audit logs that can be viewed with the prototype audit tool.

On a technical level the Internet voting example is a web application. A Node.js web server instance serves static content such as the candidate list, the result view, and client-side JavaScript. The client's browser (directed by JavaScript) uses WebSockets [12] (via socket.io [13]) to connect to three different Node.js instances (not the one sharing static content), each belonging to a different party, and communicates using the client-server protocol given in Section 4.2.

The Node.js instances are running Sharemind web controller modules, which act as the servers in the client-server protocol. The controllers handle connecting to Sharemind Application Server miner instances, passing messages between them and the client, and running Sharemind executables on the miners. The three miners handle communicating with each other independently of the controllers

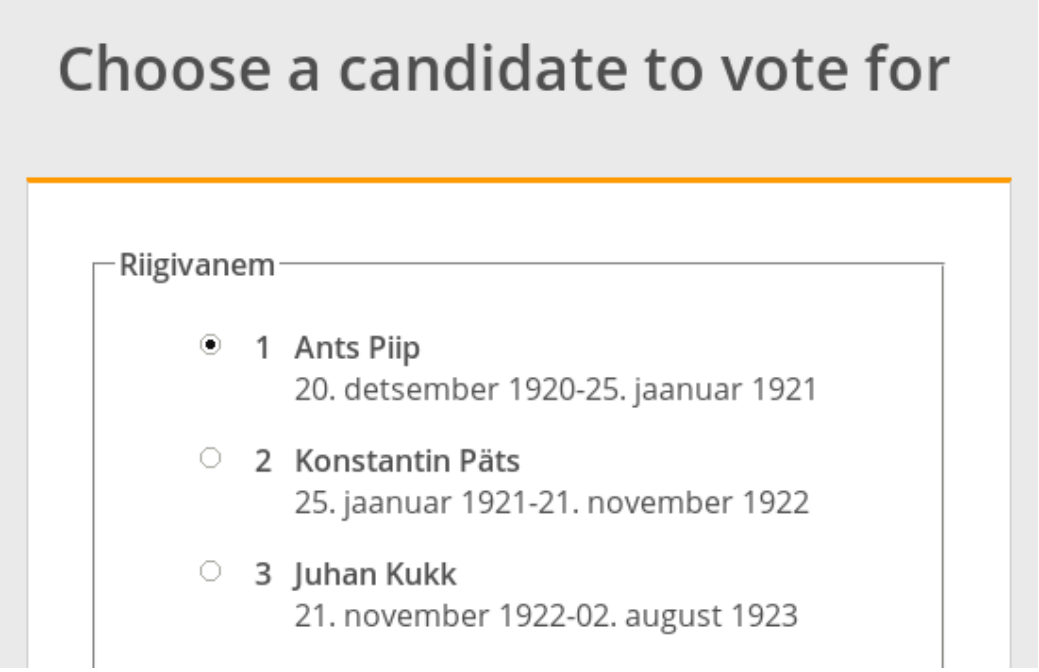

Figure 4.1: Web page displaying list of candidates grouped by parties (the only "party" that can be seen here is "Riigivanem").

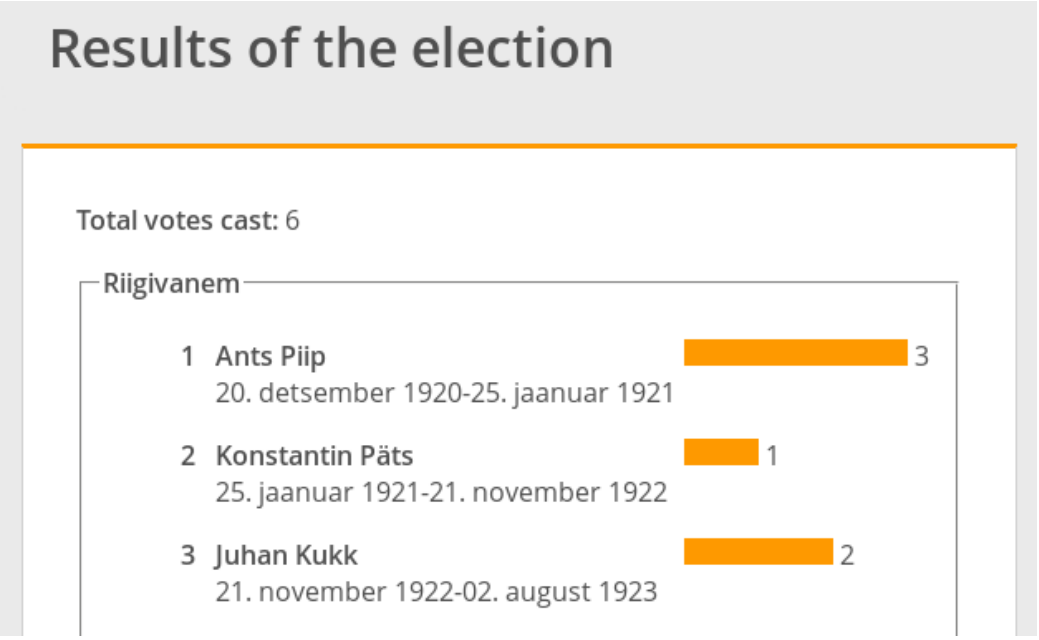

Figure 4.2: Web page displaying a tally of the votes cast.

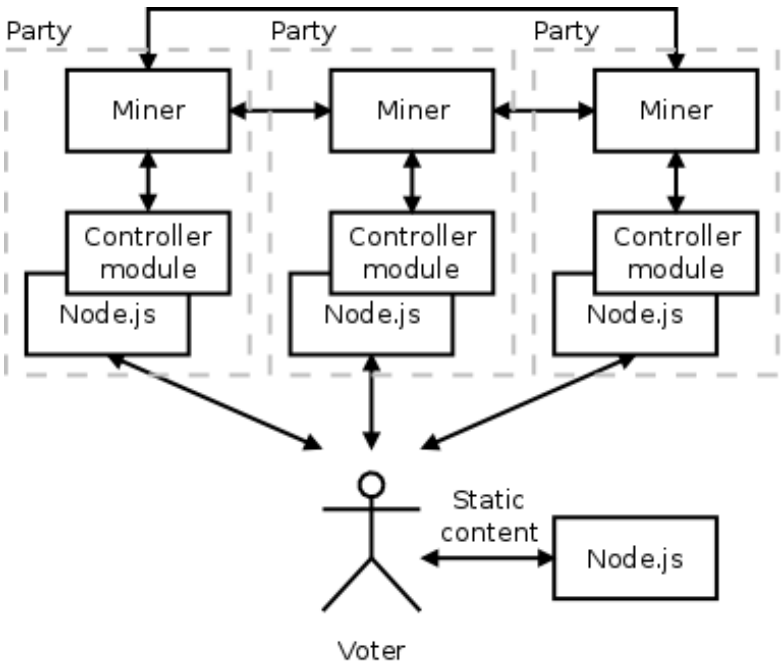

Figure 4.3: An overview of the components involved in the Internet voting example and communication between them.

(see Figure 4.3).

The source code of this demo application is not included with this thesis as it can not be used without Sharemind miners and the Node.js controller module, so it is not useful on its own.

4.2 Client-server protocol

The client-server protocol consists of three server functions that the client can invoke: generating cryptographically secure randomness, submitting the vote, and retrieving the election results.

Because the randomness generated by JavaScript is not specified to be cryptographically secure[1], the servers provide a function to ask for cryptographically secure randomness. This method was first used in [14].

The client sends a public integer to the parties, containing the number of random bytes it wishes. Each miner generates the number of cryptographically secure

[1]There is currently a W3C working draft proposing a web cryptography API for browsers (`http://www.w3.org/TR/WebCryptoAPI/`), which would also include cryptographically secure pseudo-random number generators, but this is not widely supported yet.

random bytes requested and sends them to the client. The client then combines the three different random byte vectors to obtain secure random bytes. It uses these bytes to initialize a local JavaScript pseudo-random number generator based on counter-mode AES, which will be used as the source for all further randomness.

After the client has selected a candidate, the client-side JavaScript forms a ballot. The ballot is a vector of 32-bit unsigned integers[2] which represents the list of candidates: the index corresponding to the candidate the client voted for contains the value 1, while all other elements in the vector have a value of 0.

So for example, if the election contained five candidates named Candidate$_i$, where $i = 1 \ldots 5$, then the vector $[0, 0, 0, 1, 0]$ would be a ballot containing a vote for Candidate$_4$. When additively sharing vectors, each element is shared separately, so a possible additive sharing of this ballot would be

$$
\begin{aligned}
s_1 &= [3050374495, \quad 3715785323, \quad 1716321425, \quad 774040719, \quad 2595299194], \\
s_2 &= [1677317558, \quad 3563153009, \quad 1672686238, \quad 1096563566, \quad 1245860233], \\
s_3 &= [3862242539, \quad 1310996260, \quad 905959633, \quad 2424363012, \quad 453807869].
\end{aligned}
$$

The ballot is additively shared for three parties and each share is sent to a different server. The servers collectively verify that the vote on the ballot is valid (using the function described in Section 4.3.1) and store their individual shares in their respective databases. The client is notified of success. If the vote was not valid, the client is sent an error message.

When the client asks for the results of the elections, the parties sum up the votes given to each candidate, declassify the result to make it known to all parties, and send it to the requesting client.

Protocol progress is displayed to the voter on the web page per miner. See Figure 4.4.

4.3 Audited functions

The two interesting functions to audit in this example are the vote validity checking and election result calculation functions.

4.3.1 Checking votes

As mentioned in the Section 4.2, a ballot is a vector with length equal to the number of candidates in the election, exactly one value 1 representing the candidate

[2]Although is is possible to use 8-bit integers or even plain bit strings here, that would mean that the shares need to be cast to a larger data type when calculating results later. Starting off with 32-bit integers avoids that.

Figure 4.4: Protocol progress of miner 1 reported to the client during voting (left) and displaying results (right).

voted for, and all other values 0. Since calculating the results is just summing up the votes the candidates received, a malicious client could have more than one 1 on the ballot, effectively voting for multiple candidates, and/or have values larger than 1, giving multiple votes to a single candidate.

Therefore it is necessary to validate the votes on cast ballots. A vector of 32-bit unsigned integers contains a valid vote if it satisfies the following conditions:

The vector's length is equal to the number of candidates in the election.
This is trivial to check publicly as the length of an additively shared vector is not secret. Therefore this check does not need to be audited.

All values in the vector are either 0 or 1.
Although Sharemind supports comparing secret shared values to constants, that is an expensive protocol compared to the following alternative: check that each value x in the vector satisfies

$$x^2 - x \equiv 0 \pmod{2^{32}},$$

as this holds iff $x = 0 \vee x = 1$.

Proof.

$$\begin{aligned} x^2 - x &\equiv 0 \pmod{2^{32}} \\ x(x-1) &\equiv 0 \pmod{2^{32}} \quad \Leftrightarrow x(x-1) \vdots 2^{32} \end{aligned}$$

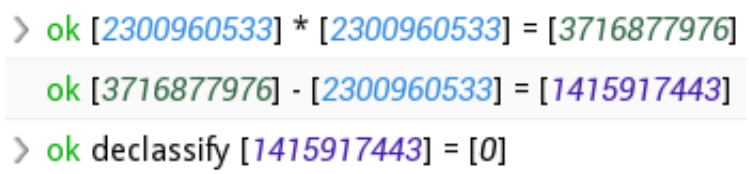

Figure 4.5: Checking if a shared value is zero or one. Note that having the end result equal to zero does not mean that the shared value was zero, but is a confirmation that it was either zero or one.

x and $x-1$ are two consecutive integers, so one must be even and the other one odd. All factors of 2^{32} are even, so

$$x \vdots 2^{32} \vee x - 1 \vdots 2^{32},$$

depending on if x is even or $x-1$ is even.

If $x \vdots 2^{32}$, then $x \equiv 0 \pmod{2^{32}}$. If $x-1 \vdots 2^{32}$, then $x \equiv 1 \pmod{2^{32}}$. □

The value $x^2 - x$ is calculated on the shares, keeping the input and all intermediate values secret, the result is declassified and publicly compared to 0.[3]

An example of how this calculation looks in the audit tool prototype can be seen in Figure 4.5.

There is exactly one value 1.

Since it was already confirmed that all values in the vector are 0 or 1 and realistically there are less than 2^{32} candidates, it is possible to verify that there is exactly one value 1 on the ballot by summing up the vector, declassifying the result, and publicly checking that the result is 1.

If any of the above conditions is not satisfied, then the honest parties will reject the ballot.

4.3.2 Calculating results

The votes are stored in a simple database containing a single table with a column for each candidate and rows containing the votes cast. When requested to tally the votes, the parties retrieve a column from their databases and sum it up to get the number of votes the candidate corresponding to that column received. This is

[3]If the declassified result is not 0, then that leaks information about the input, but this is not a problem since the system does not worry about keeping the inputs of malicious clients secret.

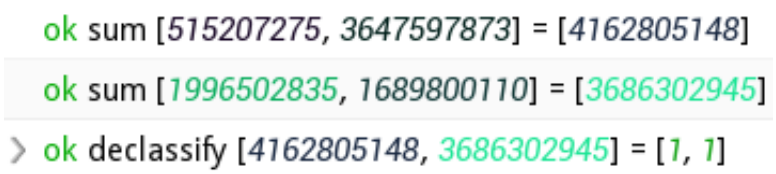

Figure 4.6: Calculating the result of two votes given to two candidates. The end result means that both candidates received a single vote.

repeated for all candidates and the results are stored in a vector in the same order as the candidates were in the votes cast. This vector is then declassified to make the election results public.

In the audit log, this will appear as a summing operation per candidate, each having as input a vector with as many elements as there are votes cast, and a single declassification operation. A minimal example with two candidates and two total cast votes can be seen in Figure 4.6.

A better approach for counting votes would be to iterate over the database rows, setting the first row as the initial accumulator and adding the other rows to it element-wise. The result would be the same as described above, yet this approach requires less memory and allows (re-)verifying the votes before counting without reading all columns into memory. But at the time of creating this example, the Sharemind database module did not support reading rows from the database—only columns—so this was not possible without modifications to the database module, which were delayed to a later time.

4.4 Possible attacks and detection

This section will discuss the potential types of attacks that the adversary could use to alter the election results and how the proposed audit logs and tools are able to detect them.

4.4.1 Computation forgery

The simplest way to forge some result is just to lie about the computation result. Since the ballots are additively shared vectors of integers, they can be easily modified: if a value is added or subtracted from a share, then that will modify the shared value by the same amount in the same direction. One possible attack vector would be to change the sum of votes given to a specific candidate. At this point assume that everything is logged properly, i.e. the inputs in the log entry of the summing operation are correct, but the output is a lie.

Figure 4.7: An example of computation forgery, where the result is incremented by one.

When a candidate has received only one vote from two cast ballots, but a party lies that it received both, the audit log has the following entry:

```
<sum>
  <input>
    <vector>
      <value>
        <!-- share of 1 -->
      </value>
      <value>
        <!-- share of 0 -->
      </value>
    </vector>
  </input>
  <output>
    <vector>
      <value>
        <!-- (share of 1 + 0) + 1 -->
      </value>
    </vector>
  </output>
</sum>
```

The same party would also have to lie that another candidate got one vote less, otherwise it would be obvious that there is an extra vote.

The audit logging model and audit tool were devised to solve this specific problem. The tool parses the audit log and simulates the protocol to automatically verify that the computation is correct. If the simulated result does not match the logged result, then a failure notice is displayed, making it obvious that something is wrong immediately after loading the file. Figure 4.7 shows how the above example looks in the prototype audit tool.

4.4.2 Audit log forgery

An obvious idea for augmenting computation forgery is to lie in the audit logs, i.e. modify the output value as discussed in the previous section, but log it and all following computations without the modification. Thus when inspecting that party's audit log alone, everything seems correct and no forgery can be detected.

This is where party views come in: as soon as a protocol involving communication between parties is encountered, there is a discrepancy between the different

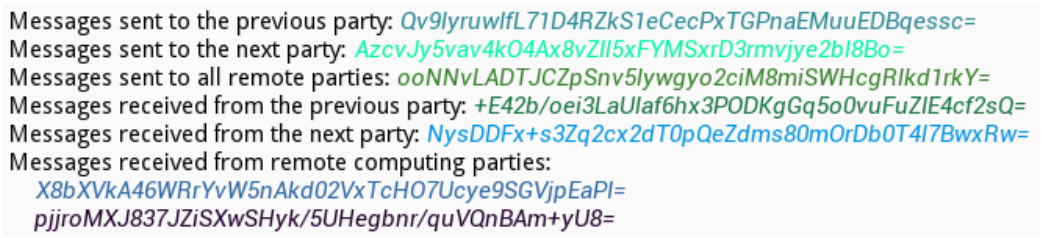

Figure 4.8: Hashes of messages sent and received are displayed at the top of the application's main window.

parties' logs. The value the modified log claims to have sent does not match with the one the receiving party's log claims to have gotten (given that the receiving party is not colluding with the forger). So it is obvious that one of those logs has a forged entry. In the Internet voting example, the last protocol of both audited functions is declassification, so there is always communication in the end to trigger this situation.

In the audit tool all messages sent to a party are concatenated and hashed, as are all messages received from a party. These hashes are displayed at the top of the application's main window (see Figure 4.8) and can be freely compared between audit logs without leaking information about the messages sent or received. It is the human auditors' job to verify that hashes of the parties' log files match.

A separate issue here is determining which log is lying—the one that sent a value or the one that received a different value—, but that is not explored here. The important part is that the fact of audit log modification is apparent and actions can be taken, e.g., discard all votes and start over with a a set of three new parties.

4.4.3 Input forgery

While modifying the output of a protocol is not possible without being caught by automatic verification or a simple comparison of view hashes, it is possible to modify the inputs to the program. All parties receive their shares of the ballot directly from the client without anybody else seeing them and as such have the opportunity to modify them.

As discussed in Section 3.4, with the current audit logging solution it is not possible to automatically identify external input values, because the semantics of the values are not known. As a result of this there is no way to automatically protect against input forgery. However, if the audit tool would have access to semantic information, it could include the external inputs in the party's view and enable the following simple example scheme to detect input forgery.

Before sending the secret shares to the parties, all clients sign the shares. The

receiving party verifies the signature, logs the share, strips the signature from it, and then passes it to the Sharemind miner. Then when later auditing the Sharemind logs, the auditor can check that the external inputs to the audited function match the shares logged by the receiver. Of course this signature verification, stripping, and logging would also have to be audited.

This signature scheme helps detect forged inputs to the miners. However—in the current voting demo implementation—after vote verification is done, a party can forge inputs to the tallying function. To detect this, the set of valid votes from the vote verification function needs to be compared to the input to the tallying function. If we have access to semantic information, we know which values in the verification function are valid votes: we just sort them, concatenate them, and hash them together. Then we do the same with the inputs to the tallying function[4] and have the auditor verify that these hashes match. If they do, then the inputs to the audited functions could not have been forged.

4.4.4 Semantic forgery

Because the protocols in the audit log are automatically examined without context, it is possible to carry out *semantic forgery*, i.e. not adhere to the agreed upon executable program. The adversary is not completely free to do what it wants, as it still needs to invoke the same protocols in the same order to maintain compatibility with the honest miners executing the proper program, but it still has some options. A simple example was already given in Section 3.4 when calculating $a + b - c$. A more relevant example is counting the votes a candidate received and declassifying them.

A malicious party sums up the votes given to a candidate correctly (to not get caught by automatic verification), but modifies the result values before declassifying them: some votes are removed from some candidates and some votes are added to others. All protocol computations are correct, but the output set from the summing protocols does not match the input set of declassification.

Without extra semantic information about which outputs of protocols should be used as inputs to other protocols, and which values are external inputs or even constants, it is impossible to automatically detect this forgery. As with input forgery, the attack can be detected by adding semantic understanding of the executed program to the audit tool.

One option for adding semantic information (presented in Section 2.4) is to include it in the log file, making it the miner's responsibility to log extra semantic information about the program it is executing. Although there is a possibility here

[4]This means reading the vote rows from the database, which requires improvements to the database module as discussed in Section 4.3.2.

for the miner to forge the logs, this information would have to match for all three parties, since they are all executing the same program. Therefore a forgery could be easily detected by comparing the semantic information of the three different logs.

Semantic forgery is very similar to input forgery: the difference being that the inputs are not from external sources, but constants and/or outputs from previous protocols. Therefore being able to detect semantic forgery also helps detecting input forgery. However, since input forgery deals with external sources, it still has the added question of verifying if the inputs match what the voters sent.

Chapter 5

Conclusions

5.1 Viability of the audited security model

As shown in Sections 4.4.1 and 4.4.2, with the current implementation of Sharemind audit logging and the prototype audit tool, the audited security model is able to detect cases of computation forgery and log file forgery.

However, that is not enough to detect all attacks an active adversary can carry out. In Sections 4.4.3 and 4.4.4 input forgery and semantic forgery were discussed, respectively, which allow to modify the results of the multiparty computation without being practically detected.

Thus in its current implementation, the audited security model does not offer any additional benefits to the passive model if the adversary is aware that the log files will be audited, which it realistically is. On the other hand, if the log files or audit tool would be augmented with semantic information as discussed in Sections 2.4 and 3.4, then it would be possible to carry out automatic detection of input and semantic forgery. This would make the audited security model viable and offer the benefit of being able to detect active adversaries in the passive model. However, including semantic information in Sharemind's audit logs was not done in this thesis due to resource constraints and the amount of work required not only by this thesis' author, but also by the developers of Sharemind.

5.2 Future Work

Obviously, the first future step would be to introduce semantic information to the current logging implementation. This would allow demonstrating automatic detection of active adversaries and make the audited security model practically useful.

Currently audit logging was introduced to Sharemind on a per-protocol basis: each protocol had to be modified to write out logging information. An improvement would be to generate this information automatically. In addition to providing logging information for all protocols, this would keep protocol code cleaner as it would not contain any audit logging specifics. A promising way to do this is the protocol domain-specific language being developed for Sharemind [15].

An additional future question is the detection of the malicious party in case some parties' views do not match. If the views do not match, then that reveals that a value was not modified, but not which party modified it. This can be trivial for some protocols, because a value must be sent to multiple parties and the majority wins, but is more difficult in other cases, where only two parties see a value.

References

[1] David Chaum, Ivan Damgård, Jeroen van de Graaf, *Multiparty computations ensuring privacy of each party's input and correctness of the result*. CRYPTO, 1987.

[2] Adi Shamir, *How to share a secret*. Communications of the ACM 22, 1979.

[3] Dan Bogdanov, Margus Niitsoo, Tomas Toft, Jan Willemson, *High-performance secure multi-party computation for data mining applications*. International Journal of Information Security 11(6), 2012.

[4] Dan Bogdanov, *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[5] Roman Jagomägis, *SecreC: a Privacy-Aware Programming Language with Applications in Data Mining*. Master's thesis, University of Tartu, 2010.

[6] *Node.js*, `http://nodejs.org/`. Last accessed May 26, 2014.

[7] *Log for C++ Project*, `http://log4cpp.sourceforge.net/`. Last accessed May 26, 2014.

[8] *Kivy: Crossplatform Framework for NUI*, `http://kivy.org`. Last accessed May 26, 2014.

[9] *Kv language*, `http://kivy.org/docs/guide/lang.html`. Last accessed May 26, 2014.

[10] *`xml.etree.ElementTree`—The ElementTree XML API*, `https://docs.python.org/3/library/xml.etree.elementtree.html`. Last accessed May 26, 2014.

[11] *`distutils`—Building and installing Python modules*, `https://docs.python.org/3/library/distutils.html`. Last accessed May 26, 2014.

[12] Ian Fette, Alexey Melnikov, *RFC 6455: The WebSocket Protocol*. IETF, 2011. `http://tools.ietf.org/html/rfc6455`. Last accessed May 26, 2014.

[13] *Socket.IO: the cross-browser WebSocket for realtime apps*, `http://socket.io/`. Last accessed May 26, 2014.

[14] Riivo Talviste, *Deploying secure multiparty computation for joint data analysis—a case study*. Master's thesis, University of Tartu, 2011.

[15] Peeter Laud, Alisa Pankova, Martin Pettai, Jaak Randmets, *Specifying Sharemind's Arithmetic Black Box*. Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies, 2013.

Appendix A

Log format schema

The XML schema of the audit logs generated by the modifications introduced to the Sharemind Application Server and parsed by the prototype audit tool.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- All data is presented in vectors to enable SIMD. -->
  <xs:complexType name="Vector">
    <xs:sequence>
      <xs:element name="value" type="xs:long"
        minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- An information block of a protocol instance.
       Just a sequence of Vectors. -->
  <xs:complexType name="Block">
    <xs:sequence>
      <xs:element name="vector" type="Vector"
        minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- A Block containing messages sent to or received
       from other nodes. -->
  <xs:complexType name="Messages">
    <xs:complexContent>
      <xs:extension base="Block">

        <!-- Who sent/received this block of messages? -->
        <xs:attribute name="node">
          <xs:simpleType>
            <xs:restriction base="xs:string">
```

```
                <!-- Sent to/received from the next party. -->
                <xs:enumeration value="next"/>

                <!-- Sent to/received from the previous party. -->
                <xs:enumeration value="prev"/>

                <!-- Sent to all remote nodes. -->
                <xs:enumeration value="remote"/>

                <!-- Received from all computing nodes. -->
                <xs:enumeration value="computing"/>

              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>

      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- A protocol instance. -->
  <xs:complexType name="Protocol">
    <xs:sequence>
      <xs:element name="input"  type="Block"/>

      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="send" type="Messages"/>
        <xs:element name="recv" type="Messages"/>
      </xs:choice>

      <xs:element name="output" type="Block"/>
    </xs:sequence>
  </xs:complexType>

  <!-- Root element of the audit log. -->
  <xs:element name="audit">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="add"        type="Protocol"/>
        <xs:element name="sub"        type="Protocol"/>
        <xs:element name="sum"        type="Protocol"/>
        <xs:element name="mult"       type="Protocol"/>
        <xs:element name="declassify" type="Protocol"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Appendix B

Source code

The source code for the audit tool prototype is publicly available on GitHub: `https://github.com/sharemind-sdk/computation-audit`.

Non-exclusive licence to reproduce thesis and make thesis public

I, Tiit Pikma (date of birth: 12.07.1989),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1. 1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1. 2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   "Auditing of Secure Multiparty Computations"
   supervised by Jan Willemson and Sven Laur.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 26.05.2014