A Domain-Specific Programming Language for Secure Multiparty Computation

Janus Dam Nielsen

PhD Progress Report

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS ·

Department of Computer Science
University of Aarhus
Denmark

Abstract

Creating tools with strong security guaranties which exploits the benefits obtained by combining confidential information without compromising it, is feasible and useful.

In this progress report we document the research carried out so far to establish the feasibility of constructing useful tools which makes it possible to take advantage of secret information from multiple sources without revealing the information. We focus on the *Secure Multiparty Computation Language* (SMCL) a domain-specific language for Secure Multiparty Computation (SMC).

We present the area of SMC along with a conceptual analysis highlighting the central concepts essential for a domain-specific language for SMC and present one realization of such a language SMCL. SMCL provides high-level abstractions and strong security guaranties to aid the programmer in producing programs for secure multiparty computation which do not reveal unintended information. We also provide a comprehensive survey of related work.

We hereby demonstrate the feasibility of constructing a useful programming language with strong security guarantees for writing SMC programs. Furthermore we present a number of ideas for future work including further developments of SMC and ideas for new tools which provide access to confidential information without compromising it.

Contents

Chapter 1

My Thesis

1.1 Introduction

Creating tools with strong security guaranties which exploits the benefits obtained by combining confidential information without compromising it, is feasible and useful. This progress report provides a status on our research into testing this thesis and our plans for future work. The thesis is relevant in an economic and a scientific perspective.

Information is a resource of huge importance and economic value to individuals, public administration, and private companies. This means that the confidentiality of information is crucial, but at the same time significant value can often be obtained by combining confidential information from various sources. Overcoming this fundamental conflict between the benefits of confidentiality and the benefits of information sharing leaves a huge yet unused potential for solving many problems of considerable economic value, like secure auctions where no information beside the final price is revealed, or information mining without revealing sensitive data. The combination of confidential information without revealing it has been know to be possible for quite some time using the technique of secure multiparty computation. However no tools have emerged which have provided any of the potential benefits. This is the challenge we have undertaken, to develop such tools.

Our first step towards providing tools which deliver the advantages of exploiting confidential data without revealing them is a domain-specific programming language [51] based on SMC. By providing a domain-specific language we allow people with a very limited understanding of SMC to solve problems like auctions and financial benchmarking using SMC. In this way we open for the easy development of new tools which exploit confidential information without revealing it.

We present the *Secure Multiparty Computation Language* (SMCL) a domain-specific language for SMC. The development of SMCL is based on an analysis of the SMC domain and experience with previous applications based on SMC. The language is carefully designed to provide high-level abstractions and strong security guaranties. The abstractions makes the application of SMC to many problems like auctions and benchmarking easy for newcomers to SMC. Security is a central issue in SMC, no information may be leaked unintended, and we identify and prevent a range of attacks including implicit flow and timing attacks. By applying a concept of checked annotations we raise the programmers awareness of which information is released.

Creating a programming language is only a first step and we envision many other tools which may provide advantages in various fields. In Chapter 6 we will present some ideas for future work mainly focusing on a idea for a database system based on secure multiparty computation, where organizations may jointly query their collective databases for interesting statistics and information without compromising their individual information.

In this report we are going to be using the term “secret” a lot, with a number of different semantics. The default meaning will be the most general, an entity is secret if it should not be revealed for some reason. An entity may also be “secret” if it is not known by some person, client, user, or organization of which we speak. The meaning should clear be from the context, if not we will make the meaning explicitly clear.

We expect the reader to be familiar with static analysis and basic cryptography on a level equivalent to the knowledge taught in most undergraduate courses on static analysis (or compiler construction) and cryptography [47]. Furthermore we expect the reader to have a rudimentary understanding of secure multiparty computation [17] and language-based security [40].

1.2 Structure

This progress report is mainly focused on our work on the SMCL language, and is thus organized to give a firm introduction to the language. In Chapter 2 we give an analysis of secure multiparty computation as domain and present the key linguistic concepts we have identified in the domain. Chapter 3 describes SMCL. Security of SMCL programs is discussed in Chapter 4 along with language semantics and we conclude our treatment of SMCL in Chapter 5 where we present and discuss various work related to SMCL. We conclude the report with an overview of our ideas for future work in Chapter 6.

Chapter 2

Secure Multiparty Computation as Domain

2.1 Introduction

In this Chapter we introduce Secure Multiparty Computation (SMC) as domain. We give a short introduction to SMC and SMCR a runtime for SMC. Based on this we provide a conceptual analysis to determine the central concepts of SMC.

2.2 SMC

The seminal example of SMC is the *Millionaire's problem* which involves a number of millionaires who want to find out which is richer, but all of them refuse to disclose their net worth. A conventional solution would involve an external trusted third party that could perform the comparisons and report the result. Using SMC it is possible to find the richest without involving a trusted third party. Yao [56] presented a solution for two millionaires that does not require an external party or any degree of trust between the two parties, and in Section 3.2 we present a solution to the Millionaire's problem expressed in SMCL.

A secure multiparty computation involves a number of parties that do not trust each other but still want to collaborate in performing a computation. In the abstract version, we have n parties $P_1, \ldots, P_n$ that wish to jointly compute the value of an integer function $f(x_1, x_2, \ldots, x_n)$, where party P_i only knows the input value x_i which must be kept secret from the other parties.

2.3 SMCR a Runtime Environment for SMC

SMCR (a further development of the system used in [11]) enables such computations to take place by allowing each party to make their input values secret, exchange them, and perform joint operations on such values. The final value of the function evaluation can only be revealed by collaboration from all parties.

Under standard cryptographic assumptions it can be proven that no party can obtain any extra information. SMCR is robust in the sense that the secrecy can only be compromised if a certain fraction of the parties decides to collude in *passive corruption*, where they pool all their secret values but continue to follow the protocol. The standard threshold is $n/2 + 1$ parties, but (more expensive) protocols exist where the threshold is $n - 1$, i.e. where each party trusts no other. SMCR is currently not robust against *active corruption* where the parties choose to sabotage the computation by not adhering to their individual part of the protocol, but such

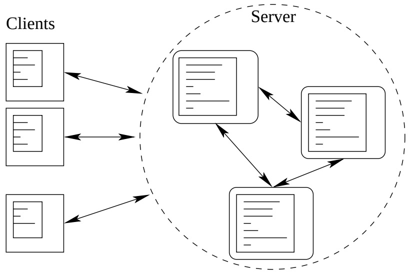


Figure 2.1: Conceptual and concrete view of clients and server

behavior is guaranteed to be detected and some (even more expensive) protocols can even tolerate a threshold of $n/3$ such parties.

SMC computations will generally involve complex protocols that involve many rounds of communication between all parties [11]. Thus, simple operations become several orders of magnitude more expensive than their non-cryptographic counterparts. Technically, the SMCR runtime is a Java API with support for public key encryption, secret sharing, primitive SMC operations, and distributed deployment and communication. We will not discuss the cryptographic challenges and technicalities in realizing the SMCR any further, but refer to [11].

2.4 Conceptual Analysis

Based on experiences with current and earlier versions of SMCR [12,26,48], we can identify a number of concepts that are used in describing realistic SMC computations.

First, a practical application will typically involve a number of *clients* that provide the inputs and receive some computed results. The computation itself is performed by a *server* which is conceptually a single machine that is realized through a number of separate parties that perform the SMC computations by running identical copies of the code in lock-step, see Figure 2.1. In a realistic example, involving the Danish commodities market for sugar beets, there are around 3000 farmers as clients and the server would be implemented by parties representing the buyers, the sellers, and a Government office. In general, there will be (possibly overlapping) one-to-many mappings from the various kinds of clients and the single server to physical machines.

Note that the clients are in principle unrelated to the parties mentioned in Section 2.2, as every secret client input is represented secretly on each of the server parties. Also, from the programmer's point of view, the server is a single entity.

Clients communicate with the server only and have no incentive to communicate directly with each other, since they generally do not trust each other. Potential attacks based on clients communicating directly and not through the server are captured in the adversary's capabilities as described in Chapter 4.

The division into clients and a single server separates public computations from secure computations respectively, in the sense that SMC computations are performed only on the server. Note that we actually have three kinds of values (and corresponding computations):

- *secret* values that reside on the server and are owned jointly by the server parties;
- *public* values that reside in plain view on the server; and

- *private* values that reside only on a single client.

Public and private computations are performed on ordinary values with a standard runtime representation. A secret value has a different runtime representation consisting of secret shares residing on the machines that physically realize the server parties, and the execution of primitive operations on such values will typically involve complex protocols with several rounds of communications. The server will have the ability to explicitly *open* a secret value which requires collaboration from all server parties. Careful limitations must be placed on the use of secret values as conditions in the control flow to avoid attacks that observe public side-effects of computations.

Clients and the server require secure and flexible communications: In some scenarios, a client only submits an input and does not need to wait for the result, whereas in other scenarios the interaction is more complex and ultimately requires a client to be connected to the server for the duration of the computation. For these purposes we identify the need for *tunnels* for asynchronous communication and *remote procedure calls* for synchronous communication.

The classical SMC applications compute a single integer function, which is similar to a straight-line program. While this is still at the core of large-scale applications, we will also allow the server to perform computations on public values and to perform iterations. As a motivating example we may consider the use of second-level protocols where a server repeatedly performs a sequence of secure auctions until some market equilibrium has been attained. Conceptually, the server will execute Turing-complete programs in which the data is separated into *public* and *secret* types. However, computations that involve only secret values still only corresponds to loop-free programs.

While the underlying cryptographic protocols are known to be provably secure, it is still a challenge to write reliable SMC applications, since confidential information may be propagated along non-obvious paths and may be leaked in subtle ways, thus an unbounded number of potential attacks exists. Implicit flows and timing attacks are classical examples [19, 28]. As another example, consider a variable x containing a secret integer value. Revealing the values of $x\%10$ and $x/10$ is sufficient to effectively reveal x itself. Thus, programmers must keep track of such value flow dependencies, which turns out to be a tedious task. However, since any non-trivial application is bound to reveal *something* about its input, the programmer must use careful judgment to determine what is acceptable. Thus, we are looking for a concept of *checked annotations* ensuring that a programmer has been made aware of all potential information leaks and has explicitly considered them.

2.5 Conclusion

To conclude, we have identified the following key concepts within the area of SMC programming:

- *Architecture*: The client-server view forms the fundamental computing paradigm of SMC, providing a separation between private, public, and secret computations and between logical and physical parties.

- *Values*: Values are either secret, private, or public, which also determines their runtime representation and separates the efficiency of primitive operations by several orders of magnitude.

- *Communication*: Clients communicate with the server only, either by using tunnels or by reacting to remote procedure calls from the server.

- *Expressiveness*: A general SMC framework must be able to perform any computation; i.e., it must be Turing-complete on private and public values.
- *Security*: Writing reliable SMC programs that do not leak unintended information, is a tedious and error-prone task that can benefit from automated assistance.

Chapter 3

SMCL Language Description

3.1 Introduction

Based on the key concepts identified in Chapter 2 we have designed a novel language called the *Secure Multiparty Computation Language* (SMCL). It is a high-level, domain-specific language [51], which allows programmers to express concepts such as clients, server, and operations on secret values directly using a special syntax and control structures tailored to the domain of SMC.

SMCL enjoys the classical advantages of being a domain-specific language as opposed to being a library API for a general-purpose language:

- The specialized syntax of SMCL closely matches the problem domain.
- A domain-specific compiler may generate more efficient code for SMCL.
- It is possible to perform domain-specific analyses that consider global properties of SMCL programs and provide stronger safety guarantees.

We start by introducing an example which gives a quick introduction to the look and feel of SMCL and then proceed to a more in-depth presentation of the basic concepts and continue with an elaboration of the example. Before concluding we shortly discuss the implementation of SMCL.

3.2 An Example

We present an example program written in SMCL in Figure 3.1, which shows an implementation of the solution to the Millionaires' Problem, generalized to an arbitrary number of millionaires.
The `Millionaires` client describes the actions of a millionaire and the `Max` server calculates and reports who is the richest. Each `Millionaires` client has a `main` function that initiates its execution. The other functions may either be invoked by the client itself (as in line `c6`) or by the server as a remote procedure call (as in line `s18`). In the example, each client submits its net worth via the `netWorth` tunnel (line `c10`). Tunnels are described in more detail in Section 3.3.4.

The server declares that `Millionaries` may belong to a group named `mills` (line `s2`). The group is processed in the `main` function of the server which describes the SMC application that is executed jointly by all the server parties. The `Max` server uses two secret variables `max` and `rich` to retain the current highest net worth and the identity of the corresponding millionaire (lines `s6` and `s7`). It then proceeds by using a `for` iterator to process each client in turn (line `s9`), updating `max` and `rich` if required. The update is guarded by the secret condition of the `if` command (line `s11`). To finish,

```
declare client Millionaires:

 tunnel of sint netWorth;

 function void main(int[] args) {
  ask();
 }

 function void ask() {
  netWorth.put(readInt());
 }

 function void tell(bool b) {
  if (b) {
   display("You are the richest!");
  }
  else {
   display("Make more money!");
  }
 }
```

```
declare server Max:
 group of Millionaires mills;

 function void main(int[] args) {

  sint max = 0;
  sclient rich;

  for (client c in mills) {
   sint netWorth = c.netWorth.get();
   if (netWorth > max) {
    max = netWorth;
    rich = c;
   }
  }

  for (client c in mills) {
   c.tell(open(c==rich|rich));
  }
 }
```

Figure 3.1: The generalized Millionaries' Problem in SMCL

the server reports to each client a boolean indicating whether or not that client is the richest. The `open` operator downgrades a value from secret to public.

3.3 Basic Concepts

We now present the basic concepts of SMCL and the role they fulfill within SMCL, how they are declared and what restrictions are put on their use.

3.3.1 Clients and Server

There is a clear distinction between the role of the server and the clients. The server does the computation and has no input or output besides communication with the clients. Clients take the input from the user, process it, provide it to the server, receive output from the server, and display it to the user. This can go on any number of times.

The server and client concept is central in the SMC world, it allows a wide range of scenarios from the paranoid self-trust scenario where the participants do not trust each other, to the more liberal scenario where some participants trust a specific party to do their part of the computation. Clients are declared using the reserved words `declare` and `client` followed by the name of the client and the client body. Similarly the server is declared using the reserved words `declare` and `server` followed by the server body. The client and server bodies are declared in the same way, as a number of functions interleaved with a number of field declarations.

3.3.2 Functions and Control

Functions

The server and clients may declare a number of functions, however at least one function, the `void main(int[] args)` function must always be present. Functions may return any kind of values and may take any kind and any number of arguments. Functions are declared using the reserved word `function`, prefix by a return type and followed by a comma-separated list of argument declarations, and finally a block command. An argument declaration is a type followed by a variable, the variable has scope in the entire block command, but may be shadowed by other variable declarations. Functions may be recursively defined.

Functions declared in a client may be invoked from the server. If the return type is secret it is treated as if the return type is public, if invoked from within the client. Otherwise the transmitted value is encrypted, and split into secret shares for the server parties.

In addition to user defined functions there are two primitive functions for input/output. The `readInt` and `display` functions are rudimentary primitives for communicating with the person controlling the client (in a future version, this will happen through a browser with support for appropriate GUI primitives). A third primitive function, `open(e|x,y,z)` is provided for declassification of secret values to public values. The operator computes and opens the secret expression `e` and declares that the programmer recognizes the simultaneous indirect leaking of some information about the secret variables `x`, `y`, and `z`. A program cannot be compiled unless it is *well-annotated*, meaning that the programmer has recognized all potential leaks (see Section 4.5 for further details).

Restrictions No recursion with secret stop conditional. No calls to functions with side-effects within conditionals guarded by secret values. Clients may no invoke functions defined outside the client.

Variable declaration

A variable is declared by first declaring its type and then its name. The name can be any Unicode character, but must begin with a letter. Variables may be declared by the server or client, as a field which is in scope throughout the server or client, but may be shadowed. Variables may also be declared in block-commands, with scope throughout the block, but may be shadowed.

While-commands

A while-command consists of the reserved word `while` followed by an expression (the condition) and a block command, for example,

```
int i = 7;
sint x = 84;
while (i < 42) {
  x += i;
  i++;
}
```

Restrictions No `while` loops may occur with secret typed condition. No `while` loops may occur inside conditionals guarded by secret values.

For-commands

A for-command is written with the reserved word `for` and then a variable declaration, followed by the reserved word `in`, followed an expression which evaluates to a group, and finally a block-command. The declared variable is in scope inside the block-command and draws a new value from the group in each iteration of the for-command. An example is:

```
S17:   for (client c in mills) {
S18:    c.tell(open(c==rich|rich));
S19:   }
```

Restrictions Only groups can be iterated using for-commands.

If-commands

An if-command is constructed from the reserved word `if`, a conditional expression followed by a command and an optional occurrence of the reserved word `else` along with another command, for example,

```
S11:    if (netWorth > max) {
S12:     max = netWorth;
S13:     rich = c;
S14:    }
```

Restrictions No I/O inside conditionals guarded by secret values. No assignments to public non-locally declared variables inside conditionals guarded by secret values. No function calls to functions which have side-effects inside conditionals guarded by secret values. No `return` commands inside conditionals guarded by secret values. No `while` loops inside conditionals guarded by secret values.

3.3.3 Types

The SMCL language supports the primitive datatypes `int` and `bool`. The identities of clients also form a datatype `client`. All of these have secret versions, denoted `sint`, `sbool`, and `sclient`. The types `sbool` and `sclient` are represented as secret integers at runtime, because the SMCR only manipulates public values and secret integers. A secret client is a client whose identity (IP-address) is secret shared; the total number of clients is always public. Furthermore, it is possible to construct records and multi-dimensional arrays of such primitive datatypes. Private, public, and secret datatypes support the same standard primitive operations, and the type system ensures that results are secret unless all arguments are public (this may involve implicit conversions to the runtime representation of secret values).

3.3.4 Tunnels

A tunnel is a mean for asynchronous communicating between a client and the server, and is declared in clients only. A tunnel is declared using the reserved words `tunnel` and `of`, followed by the type of values that can be placed in the tunnel, and terminated by the name of the tunnel, for example,

```
C3:  tunnel of sint netWorth;
```

A tunnel may be declared to contain data of any primitive type, `int`, `sint`, `bool`, `sbool`. When a client sends a secret value to the server, the transmitted value is not only encrypted, but it is also split into secret shares for the server parties, matching the runtime representation of secret values. When the server sends a secret value to a client, all server parties send encrypted version of the secret shares which are then assembled on the client to yield a private value. Public values sent are encrypted but not secret shared. A tunnel is equipped with functions for sending, `put`, and retrieving, `get` data and for inquiring the emptiness, `isEmpty` of the tunnel.

Currently a tunnel is only working as long as the client and the server are running. That is if one of them terminates, any information in the tunnel is lost and the remaining process is likely to terminate abruptly. In a future version we would like to introduce various different kinds of persistent tunnels, e.g. an XML-tunnel, database-tunnel, or a plain file-tunnel, where all values are serialized and stored persistently. The values can be store in different places. In a central computer if all values are encrypted using the public keys of the server parties. Another possibility is to store the values for each party at a computer owned by the organization running the server party.

```
client: Millionaires
 gates.microsoft.com   4001  0x85FFA494  mills
 ebenezer.scrooge.org  4001  0x5532BB72  mills
 ingvar.ikea.com       4001  0x2333DDCC  mills
 larry.google.com      4001  0x631DE7F2  mills
 sergei.google.com     4001  0x7587B5AF  mills

server
 gates.microsoft.com   4000  0x857722B7
 smcl.brics.dk         4000  0xF471BCA7
 survey.fortune.com    4000  0x66A7FF35
```

Figure 3.2: A map identifying the concrete participants

Restrictions Only primitive types can be sent through a tunnel.

3.3.5 Groups

A group is a collection of clients. Currently a group may only contain clients of the same kind, they have to be homogeneous. Groups may only be declared in the server. A group is declared using the reserved words `group` and codeof followed by the name of the clients and the name of the group, for example,

```
S2:  group of Millionaires mills;
```

The members of a group are specified externally by a mapping supplied to the SMCR runtime describing the concrete participants involved during runtime. Figure 3.2 shows a hypothetical example. Each participant is identified by an IP address, a port number, and a public encryption key. Note that the same machine may serve both as a client and as a server party. Clients may further be listed as belonging to a number of groups, in this case only the single group `mills` containing all clients.

Restrictions Only homogeneous groups.

3.4 The Example Elaborated

```
for (client c in mills) {
  c.tell(open(c==rich|rich));
}
```

(A) public booleans, public receivers

```
for (client c in mills) {
  c.tell(c==rich);
}
```

(B) secret booleans, public receivers

```
for (sclient c in mills) {
  c.tell(open(c==rich|c,rich));
}
```

(C) public booleans, secret receivers

```
for (sclient c in mills) {
  c.tell(c==rich);
}
```

(D) secret booleans, secret receivers

Figure 3.3: The combinations of server knowledge

In generalizing the original Millionaries' Problem from two to many millionaires, we have in our solution chosen that while the net worth of each millionaire remains secret, it is actually public information which millionaire is the richest, see Figure 3.3(A). In a stricter version of the generalized problem we could also keep this information secret and only allow each millionaire to know his own status. In our program, we would then change lines `s17` through `s19` into the lines of Figure 3.3(B).

Here, we do not open the secret boolean before it is sent to the client. This means that the server parties send their shares representing the value of type `sbool` to the client which combine the shares into a value of type `bool`. An equivalent effect can be achieved by changing the iterator `c` to have type `sclient` and thus keep it secret while revealing the comparison result, Figure 3.3(C). Consequently, the invocation `c.tell(...)` is now implemented by sending to all clients the same message that can only be understood by the intended recipient (function invocations with illegible arguments are ignored by the clients). Since `c` is now also secret, the `open` operation must also recognize responsibility for compromising it (ever so slightly). In a yet stricter version, we may change the three lines into the lines of Figure 3.3(D).

For this particular example, however, this refinement makes no difference (since the server always sends one `true` value and a number of `false` values).

3.5 Implementation

SMCL is implemented by a prototype compiler (SMCLc) which produces Java code based on the SMCR API. A Java program is created for each kind of client and for the server. Deployment scripts can be used to install and start applications. Currently, all communication takes place through a coordinator process (that only sees encrypted information). The coordinator could itself be distributed using broadcast protocols.

3.6 Efficiency

Our experiences with SMCR show that Secure Multipart Computations are feasible in practice. However, secret computations are quite expensive as they are based on complex protocols that involve several rounds of communications between the server parties. To illustrate this, we consider a program which computes the sign of a polynomial given coefficients `a`, `b`, and `c` and a data point `x`. We provide three versions of this program shown in Figure 3.4. To enable proper timings, the client network communications have been replaced with simple assignments. The ideal version keeps everything secret until the output is revealed. The pragmatic version has `x` as a public value and chooses to allow the value of the polynomial to be public as well as its sign. The public version merely performs an ordinary computation.

In Figure 3.4, we show the timing results from running the compiled versions of these programs on SMCR with 3, 5, and 7 server parties distributed on an equal number of Intel P4 1,8 Ghz with 512 MB of memory (the timings are for one execution of the programs and do not include the time for *preprocessing*, which is a part of the protocols that SMCR uses for multiplications and comparisons). The time needed for preprocessing depends on the number of multiplications done in the computation. The SMCR can be instructed to preprocess a number of multiplications and furthermore use idle time to maintain a pool of preprocessed multiplications. The numbers 1, 2, and 3 denote the threshold that is used in the respective case. Our conclusion is that SMC primitives are expensive but feasible. The slowdown from the public to the pragmatic version is significant but many practical application exists where the slowdown is acceptable. An example is offline auctions where ample time is available for executing the auction. The slowdown from the pragmatic to the ideal version is stunning, but it is to a large extent an unavoidable price for obtaining the full invulnerability of our security properties. In practice, applications will be written in the pragmatic style—making a convincing case for automated proof support like our simple annotation language. It should also be noted that there are still many opportunities for optimizing SMCR.

```
sint x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
sint p = a*(x*x) + b*x + c;
sint sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = open(sign|p);
```

Ideal version

```
int x = 17;
sint a = 42;
sint b = -5;
sint c = 87;
int p = open(a*(x*x) + b*x + c|a,b,c);
int sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = sign;
```

Pragmatic version

```
int x = 17;
int a = 42;
int b = -5;
int c = 87;
int p = a*(x*x) + b*x + c;
int sign = 0;
int output;
if (p < 0) sign = -1;
if (p > 0) sign = 1;
output = sign;
```

Public version

	ideal	pragmatic	public
(3,1)	12 sec	30 ms	<1 ms
(5,2)	17 sec	65 ms	<1 ms
(7,3)	30 sec	132 ms	<1 ms

Figure 3.4: Three versions of the polynomial program and their timing results in SMCR

The SMCL compiler employs a range of static analyses to boost efficiency, and the timing results clearly show that the potential payoff can be dramatic. These analyses are all simple instances of the monotone framework [16] based on fundamental analyses described in [38], but they are interesting because they solve important domain-specific problems and thus illustrate the benefits of using a domain-specific language.

3.7 Conclusion

We have described the Secure Multiparty Computation Language, a high-level, domain-specific language, which allows programmers to express concepts such as clients, server, and operations on secret values directly. We have discussed the basic concepts of SMCL, how they are connect, their restrictions and use. We have presented the seminal Millionaire's problem in SMCL and how it may be generalized in different ways. Furthermore we have show results witnessing the feasibility of SMCL and SMCR.

Chapter 4

Security in SMCL

4.1 Introduction

As discussed in Chapter 5, security requirements are many and multidimensional. Also, the problems to be considered depend heavily on the capabilities that an adversary are assumed to possess [14, 29].

For SMCL, we are able to obtain quite strong security properties in the face of powerful adversaries due to two properties: 1) the use of strong cryptographic protocols in SMCR , and 2) a careful design of SMCL and its semantics.

To handle many specific but important modes of attack in a common framework, we will assume an unusually strong model of the adversary, which is able to observe the physical state of the server: At every clock cycle the entire layout of memory and the instruction pointer are available for inspection. However, the secret values are not visible to any adversary (unless more than the given threshold of the server parties have been corrupted in which case no guarantees are given) and neither are the private values of the clients. We assume that clients cannot corrupt other clients but clients may collaborate, e.g. share information. This is a strong adversary who is capable of many common attacks including e.g. simple eavesdropping and more complex attacks which are a function of the program trace, like interference, and timing [23, 28].

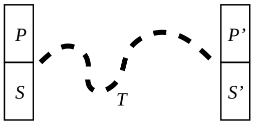
To formally define these notions, we have provided a small-step operational semantics of SMCL programs which we will describe in Section 4.3. We consider only *well-typed* programs as described in Section 4.4, which have the simple property that variables with public types can never contain secret values [46, 52, 54].

4.2 Adversary Traces

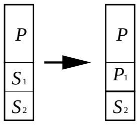
To formalize our security guarantees, we introduce a notion of *adversary traces*, which contain the information that is made available to an adversary. Such a trace consists of the entire sequence of system states (configurations in the small-step semantics) that is encountered during the evaluation of a program with three restrictions:

- secret values on the server and in tunnels are masked out;
- the private states of clients are not available; and
- no `open` operations are performed.

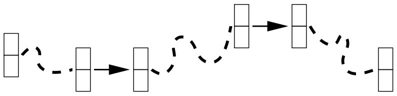
The capabilities of an adversary are then limited to observing these traces. We will use an illustration to show an adversary trace T from a state with public values P and secret values S to one with public values P' and secret values S':



Also, we use an illustration to show a transition where a part S_1 of the secret state is made public using the `open` operation to become the public state P_1:



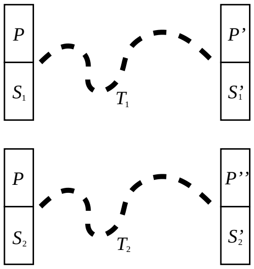
A complete computation that occasionally makes use of the `open` operation for downgrading is then described by an alternating sequence of adversary traces and these transitions:



The security guarantees of well-typed SMCL programs can now be expressed through two properties that will be ensured by the compiler.

The Identity Property

The *identity property* states that whenever we have the two situations

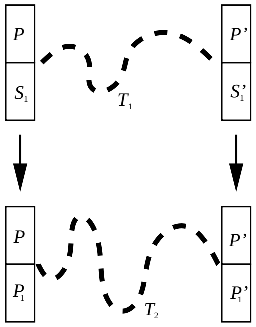


then $T_1 = T_2$ (and thus also $P' = P''$). This is a strong property stating that computations from initial states with the same public values will have *identical* observable traces from the point of view of the adversary. This implies the property of *noninterference*, which normally only requires that the resulting public values must be equal [40].

This property implies that SMCL programs are immune to a range of attacks that attempt to exploit information leaks, namely all of those where the leaked information is a function of the adversary trace. This includes timing attacks as discussed in [10, 28] and also more exotic attacks, e.g. based on measuring radiation from the server [13]. SMCL programs are even immune to stronger timing attacks, since we not only assume that computations have the same overall duration regardless of the secret values, but also that the instruction pointer of the server is independent of any secret conditionals at any point in time. Invulnerability to attacks of course hinges on the same properties holding for the basic operations on secret values, where the protocols are independent of the argument values.

The Commutativity Property

The *commutativity property* states that `open` operations and computations commute:



This property evidently expresses that the secret representation is sound. Note that T_1 and T_2 will in general clearly be different, but the commutativity property implies that T_1 terminates exactly when T_2 does.

Ensuring Security Properties

Validity of the two security properties hinges on two properties of the SMCL language:

- a runtime semantics where *both* branches of an `if` command with a secret conditional always terminate and are always evaluated in sequence; and
- static analyses of well-typed SMCL programs to verify that such branches have no public side-effects.

4.3 Timing and Termination attacks

In this section we describe how the semantics of SMCL are carefully crafted to prevent timing attacks and furthermore we describe how termination attacks are prevented.

Timing attacks exploits the information signaled through the time at which an action occurs or the time an action takes to complete. Similar, termination attacks exploit the information signaled through the termination or non-termination of a computation. Differences in execution time of an SMCL program may occur as result of the execution of a conditional command where one branch takes longer than the other. Or when a computation is redone as in `while` loops or after a conditional command, where the same computation was done in either branch (cache-timing attack [10]). Termination attacks are only relevant in connection to `while` loops and recursive functions.

By timing the execution of a secret guarded conditional (or `while` loop) with standard semantics an adversary may reveal the secret information. We cannot of cause allow this possibility, thus a solution is needed. A solution cannot be based on running either of the branches since we assume the adversary has access to the instruction pointer. This observation also rules out the solution by Agat [3] where branches are cross-padded with dummy instructions. However the approach by Agat can be taken to an extreme which is the execution of both branches in the store obtained after evaluating the condition of the conditional no matter what the condition is. Then if we combine the results of the two branches in the end based on the condition we get the correct result as if only the right branch had been evaluated.

To formalize this we now introduce the formal semantics of SMCL. We will be needing a good deal of terminology which we introduce as we go along. A complete list can be found in Appendix A.

SMCL is a concurrent imperative language where a number of clients contribute data to and consume data from a server. Concurrency means that the server and clients may be executing at the same time. The server and clients may be executed with an arbitrary interleaving of commands and expressions, and only need to synchronize for communication. SMCL consists of a number of functions containing commands, which are executed sequentially. Commands may contain expressions which potentially have side effect on the state of the program.

An SMCL program consists of a server, σ and a set of clients, χ. The execution of an SMCL program is modeled using a number of transition systems with configurations $\gamma \in \Gamma$, terminal configurations $\mathrm{T} \subseteq \Gamma$, and transition relation $\Rightarrow \subseteq \Gamma \times \Gamma$. The server and clients are both evaluated using the concepts of commands and expressions, but the possible computations and semantics are different on the server and the clients. We model this difference by using different transition systems for client-side expressions, client-side commands, server-side expressions, and server-side commands. The transition systems are fairly standard and we will not go into detail with these transition systems here, but refer to [37], the non-standard parts will be discussed below to some extent. The transition systems are used in the *SC* transition system described below. The SC system is an overall system describing the interaction and communication between the server and clients.

$$\begin{aligned}
\Gamma_{SC} &= \Gamma^1_{COM_{cl}} \times \ldots \times \Gamma^n_{COM_{cl}} \times COM_{sv} \times \mathrm{State}_{sv} \\
&\cup \; \Gamma^1_{COM_{cl}} \times \ldots \times \Gamma^n_{COM_{cl}} \times \mathrm{State}_{sv} \\
\mathrm{T}_{SC} &= \mathrm{T}^1_{COM_{cl}} \times \ldots \times \mathrm{T}^n_{COM_{cl}} \times \mathrm{State}_{sv}
\end{aligned}$$

A configuration in the SC transition system consist of a number of configurations from the respective transition systems, one for each client process and one for the server. We write configurations, $G \vdash \|_{i \in \chi} \langle \kappa_i : C \rangle \parallel \langle \sigma : C, \sigma.S \rangle$, in the SC system as a number of concurrent configurations for the clients which are in the set χ, similar to [50], and an additional configuration for the server. $\kappa_i : C$ means that client κ_i is about to evaluate the command C and similarly for the server, σ.

We define the *Global client store* G in Figure 4.1 to be a tuple containing the client stores, $\kappa_i.S$, and all the tunnels, $\kappa_i.\theta_j$. We write $G[O \mapsto U]$ for the store G where the component O is updated to hold U. The server and each client, ξ, have a local store, $\xi.S$ defined as a map from variables to values: $\xi.S = Var \mapsto \mathrm{Value}$. The local stores also work as environments. We will not describe it further since it works as one would expect. A *preliminary* store is a store when the computation begins. The preliminary local store is initialized with bindings from function names to function bodies. Also bindings from field variables to an initial value which depend on the type of the field are added. From here on fields are treated as ordinary variables which may be shadowed in the current scope.

$$G \quad : \quad [\kappa_1.S, \ldots, \kappa_n.S, \kappa_1.\theta_1, \ldots, \kappa_1.\theta_j, \ldots, \kappa_n.\theta_1, \ldots, \kappa_n.\theta_k]$$

Figure 4.1: The global client store of an SMCL computation

The transition rules of the SC system are described in Figure 4.2 where the first rule describe how clients are evaluated and the second rule describes how the server is evaluated. In this way a client or server may be evaluated one step using the appropriate transition system. We will not present all of the rules here but focus solely on the evaluation of conditionals on secret values, because they are the focus point of timing attacks. The rest are mainly standard.

$$\frac{G \vdash \langle \kappa_i : C \rangle \ \rightarrow_{COM_{cl}} \ \langle \kappa_i : C' \rangle}{G \vdash \|_{i \in \chi} \langle \kappa_i : C \rangle \parallel \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{SC} \ \|_{i \in \chi} \langle \kappa_i : C' \rangle \parallel \langle \sigma : C, \sigma.S \rangle} \ \big(\textsc{client-eval}\big)$$

$$\frac{G \vdash \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{COM_{sv}} \ \langle \sigma : C', \sigma.S' \rangle}{G \vdash \|_{i \in \chi} \langle \kappa_i : C \rangle \parallel \langle \sigma : C, \sigma.S \rangle \ \rightarrow_{SC} \ \|_{i \in \chi} \langle \kappa_i : C \rangle \parallel \langle \sigma : C', \sigma.S' \rangle} \ \big(\textsc{server-eval}\big)$$

Figure 4.2: One-step evaluation of an SMCL computation

We present a part of the semantics related to evaluation of condition on secret values because it is essential to ensure security from timing attacks [3]. Timing attacks may be possible if the computation of the two branches does not take the same amount of time. Furthermore we operate in a scenario where the code is executed on an untrusted computer, thus an adversary may inspect the program pointer and compare values in registers and memory and maybe discover some correlation he should not have.

An adversary who is in control of the program pointer knows which branch is being evaluated thus we must develop a scheme that lets us compute the correct result, but in a way that does not reveal which branch was taken in a conditional. The solution is based on the idea of *conditional assignment*: `x = b*y+(1-b)*z`. The variable `x` is assigned the value of `y` or `z` based on the value of condition `b`. Applying this technique after the execution of both branches in the same initial store, where `y` and `z` are the values of x resulting from the `then` and `else` branch respectively gives the correct value. Furthermore it removes timing attacks as a security threat if values are immutable and if two representations of a secret value are not equal. Fortunately the two last requirements are properties common in secure multiparty computation and thus already present in SMCR.

In Figure 4.3 we present two of the seven transition rules concerning evaluation of conditional commands on secret values. The other 5 rules are mainly concerned with evaluation of the command in each branch to obtain the forms shown in Figure 4.3 and will not be presented here. We use $\boxed{v}$ to denote a secret value (we cannot see it, it is inside a box).

The rule IF-SBOOL-ELSE is the result of evaluating command C_1 as far as possible. C_1 has been evaluated in a local store, U_{then} which is created as a copy of the original store S which is threaded along. Rule IF-SBOOL-ELSE says that in order to evaluate C_2, we should evaluate C_2 with the store S (effectively a copy of S) in the server-command transition system. The evaluation results in a store S' which we save as part of the new state of the conditional as the store U_{else}. The evaluation of C_2 now proceeds in a number of steps using the store U_{else}, and eventually the system end up in a configuration as the one in rule IF-SBOOL-PHI. The last step of evaluating a conditional command is the combination of stores from each branch. The result is a store $\sigma.S'$ which is the same as the original store S but updated for each variable which has been assigned to during the execution of a branch. We update a variable `x` by looking up `x` in both U_{then} and U_{else} and combine the values using conditional assignment. If a variable has not been updated in a branch its value is unchanged compared to the original store and conditional assignment ensures that the result is correct.

Evaluation of both branches removes timing attacks because the execution time is independent of the condition. Cache-timing attacks are also eliminated because which branch gets executed is independent of the condition, and thus the state of the cache contains no information about the value of the condition.

Evaluation of both branches is however not sufficient to ensure the security properties. Since we always execute both branches, we need to make sure that they will both always terminate. To this end, the SMCL compiler performs a static analysis that conservatively checks the branches for termination (using simple syntactic criteria in the present implementation). Furthermore no return commands, I/O, or function calls with side-effects are not allowed.

`while` loops on secret conditionals and calls to recursive functions which recur based on secret conditions, cannot be treated in the same way as conditionals on secret values. In conditionals we only needed a finite number of additional stores, in a loop or recursion and unbounded number is needed. Currently we can see no better alternative than to disallow `while` loops and recursive functions based on a secret condition. Iteration through a group of clients is possible using a `for` iterator, and if the identities of the clients are secret then the iteration is performed through a secret random permutation of the clients computed at the time of use to avoid revealing any secret information.

$$\frac{G \vdash \langle C_2, S\rangle \;\rightarrow_{COM_{sv}}\; \langle C_2', S'\rangle}{G \vdash \langle \mathbf{if}(\boxed{v})\ \{\}\ \mathbf{else}\ \{C_2\}, U_{then}, S\rangle \;\rightarrow_{COM_{sv}}\; \langle \mathbf{if}(\boxed{v})\ \{\}\ \mathbf{else}\ \{C_2'\}, S', U_{then}, S\rangle}$$

$\left(\textsc{If-sbool-else}\right)$

$$\frac{\begin{array}{c}\sigma.S' = S[x \mapsto \boxed{v} * U_{then}(x) + (1 - \boxed{v}) * U_{else}(x)] \\ \forall x \in S | U_{then}(x) = v \;=\; U_{else}(x) \vee U_{then}(x) = \boxed{v'} \;\wedge\; U_{else}(x) = \boxed{v''}\end{array}}{G \vdash \langle \mathbf{if}(\boxed{v})\ \{\}\ \mathbf{else}\ \{\}, U_{else}, U_{then}, S\rangle \;\rightarrow_{COM_{sv}}\; \langle \sigma.S'\rangle}$$

$\left(\textsc{If-sbool-phi}\right)$

Figure 4.3: Sample of server-side semantics for secret conditional commands

4.4 Hoistability

In this Section we introduce the concept of hoistability and describe a type-system based approximation. As described before merging the two branches of conditionals removes timing leaks. However it does not prevent implicit flow [19]. This includes assignments to public variables with scope outside the branches, function calls, IO, and communication with clients. To this end, the SMCL compiler performs a static analysis that conservatively checks that all public side-effects can be hoisted out of the two branches without changing the semantics; specifically, this includes non-local assignments, function calls, and communication.

Note that *hoistability* is a general (and undecidable) concept that is implied by conventional requirements for noninterference [46,52,54]. Instead of fixing a specific decidable requirement, we will allow the implementation of the SMCL compiler to perform any sound approximation of this property.

In our current implementation, hoistability is approximated by a type system based on a type system by Volpano and Smith [52, 54] which includes *effects* in the style of Jouvelot and Gifford [27]. We extend this type system by tracking all public side-effects and allowing assignments to locally defined public variables in secret value conditionals.

We use a lattice of security levels as introduced and used by many others before [19–21, 52, 54]. The lattice has two security levels, the secret or high-level, S, and the public or low-level, P, where $\mathsf{P} \sqsubseteq \mathsf{S}$. The lattice describes the highest security level at which a variable has been *read*, and we call it the Read-lattice. In order to record the other public side-effects we introduce two more lattices. First, the Write-lattice describing the least security level at which a variable has been *written* to and whether the variable is declared in the current scope. PG (public, global written) $\sqsubseteq$ PL (public, local written) $\sqsubseteq$ SG (secret, global written) $\sqsubseteq$ SL (secret, local written). Second, the I/O-lattice, a lattice describing whether I/O has occurred IO (I/O occurred) $\sqsubseteq$ NIO (no I/O occurred).

The partial ordering, $\sqsubseteq$, of the lattices gives rise to straightforward reflexive, transitive, and anti-symmetric subtyping relations $\leq$ when we view the elements of the three lattices as types. The least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operator on the lattices give rise to the least common super type $\veebar$ and the greatest common subtype $\barwedge$. The operations naturally extends from the binary to n-ary case, $\veebar_{i=1}^{n}\rho_i$ and $\barwedge_{i=1}^{n}\rho_i$ and similar for the other types.

We adapt the approach by Volpano and Smith [54] and define four kinds of type. The *variable types*, (τ, ρ)-var, describes the kind, τ, of values which may be assigned to the variable and the security level of the value, $\rho \in \text{Read} - \textit{lattice}$. *Expression types* $(\tau, \rho, \nu, \iota o)$-exp are assigned to expressions. Intuitively an expression has a given type $(\tau, \rho, \nu, \iota o)$-exp if the value computed by the expressions has type τ, there will be a read from variables of at most security level ρ, assignments to variables of at least ν, and ιo describes whether I/O will be made during the evaluation of the expression. *Command types* are assigned to commands. A command with type $(\nu, \iota o)$-cmd is expected to maintain the invariant that no assignments are made to a variable in the command of type lower than ν. ιo describes whether I/O is going to occur during the evaluation of the command. *Function types* $(\nu, \iota o)$-ρ-fun$(\rho_1, \ldots, \rho_n)$are assigned to functions.

The typing context, $\Gamma_t = [\Sigma, \eta, \mu, \varpi]$, is a tuple of the server declaration Σ, the var-typing η, the write-typing μ, and the return type of the current function ϖ. The var-typing and the write-typing are a finite maps from variables to variable-types and write types (from the write-lattice) respectively.
A typing judgement has the form $\Gamma_t \vdash C \ : \ (\nu, \iota o)$-cmd for commands. The judgment means that command C has type $(\nu, \iota o)$-cmd, assuming that $\Gamma_t$prescribes the server declaration, types for any variable in C and a return type for the current function. Similar for variable, expression, and function types. The server declaration contains information from which we can easily construct the type of tuples and functions (arguments and return type).

We now present the three most interesting type rules of the type system. The other type rules are mainly as one would expect, for a complete treatment see [37]. In Figure 4.4 we present the type rule for assignment. The rule is as one would expect, except for the use of the write function. We would like to allow assignments to public variables if they are declared within the branch in which the assignment occurs. We use a *local reaching definitions* analysis which for each program point computes the set of variables declared in the same scope to decide this. The write function uses the result of the local reaching definitions analysis to decide which element from the write-lattice should be returned.

In Figure 4.5 we present the typing rules for conditional commands. The first rule Tif-secret define the type of conditional commands with a secret condition and the second rule Tif-public the type of conditionals with public condition. The rule Tif-public just propagate the side-effects of the condition and the branches, whereas the rule Tif-secret is more interesting. Here we require no I/O in the branches and assignments

$$\frac{[\Sigma,\eta,\mu,\varpi] \vdash \mathtt{x} : (\tau,\rho)\text{-var} \qquad [\Sigma,\eta,\mu,\varpi] \vdash e : (\tau',\rho',\nu',\iota o')\text{-exp} \qquad \tau' \leq \tau \quad \nu = \text{write}(\mathtt{x},\eta,\mathtt{x} = \text{e})}{[\Sigma,\eta,\mu,\varpi] \vdash \mathtt{x} = e : (\tau, \rho \mathbin{\underline{\vee}} \rho', \nu \mathbin{\overline{\wedge}} \nu', \iota o')\text{-exp}}$$

(tAssign)

Figure 4.4: Typing rule for assignment

must be of type at least public and local. This rules out implicit flow to local variables declared outside the branches, but allows assignment to variables declared inside the branches, a formal proof of soundness is work in progress. The I/O of the condition determines the I/O of the conditional. The variables written to is the greatest common suptype of those in the branches, PL, and in the condition. A

$$\frac{\Gamma_t \vdash e : (\text{bool}, \mathsf{S}, \nu, \iota o)\text{-exp} \quad \Gamma_t \vdash C_1 : (\mathsf{PL}, \mathsf{NIO})\text{-cmd} \qquad \Gamma_t \vdash C_2 : (\mathsf{PL}, \mathsf{NIO})\text{-cmd}}{\Gamma_t \vdash \textbf{if } (e)\ \{C_1\} \textbf{ else } \{C_2\} : (\mathsf{PL} \mathbin{\overline{\wedge}} \nu, \iota o)\text{-cmd}}$$

(tIf-secret)

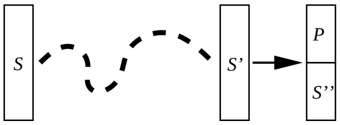
$$\frac{\Gamma_t \vdash e : (\text{bool}, \mathsf{P}, \nu_0, \iota o_0)\text{-exp} \quad \Gamma_t \vdash C_1 : (\nu_1, \iota o_1)\text{-cmd} \qquad \Gamma_t \vdash C_2 : (\nu_2, \iota o_2)\text{-cmd}}{\Gamma_t \vdash \textbf{if } (e)\ \{C_1\} \textbf{ else } \{C_2\} : (\overset{2}{\underset{i=0}{\overline{\wedge}}}\ \nu_i,\ \overset{2}{\underset{i=0}{\underline{\vee}}}\ \iota o_i)\text{-cmd}}$$

(tIf-public)

Figure 4.5: Typing rules for conditional commands

4.5 Semantic Security

The security properties provide some basic guarantees about the behavior of SMCL programs. With these guarantees, any computation (without `while` loops) can be made invulnerable to attack by being structured as an *ideal computation*:



Here, all information is kept in secret variables and only at the very end are the outputs P made public. However, as shown in [36], computations on secret values are quite expensive. Thus, a pragmatic computation will keep information in public variables as much as possible without compromising the overall security requirements. The commutativity property ensures that the ideal computation and the pragmatic computation will produce the same output, but the programmer now has the burden of (manually) proving that these two computations will only reveal the same relevant secret information. Since such proofs are difficult to construct, the SMCL compiler provides a simple annotation language to aid the programmer.

The `open` operation may be annotated with the names of some secret variables: `open(e|x,y,z)`. The meaning of this annotation is that the programmer recognizes responsibility for compromising the secret values of these variables, and the compiler should check that all compromised variables are mentioned, so the programmer is fully aware of his proof obligations. A program is then only accepted as *well-annotated* if all potential semantic information leaks are explicitly allowed by such annotations. To be conservative, which is a good attitude when security is concerned, any secret variable whose value may have influenced the opened value is viewed as potentially compromised. Thus, for any `open` operation the SMCL compiler computes the set of secret variables that have ever contained a value that may have influenced the value currently being opened. From this set of potentially compromised secret variables we subtract the corresponding sets from all previously executed `open` operations whose values have not since changed. The resulting set of newly compromised secret variables must explicitly be mentioned in the `open` operation. The set of variables which must be mentioned may grow fast thus for ease of annotation we also subtract the variables which may have influence any already mentioned variable. The corresponding analysis is a mixture of a def-use analysis, a liveness analysis, and an available expressions analysis [38]. A simple constant folding analysis also takes care of cases such as multiplying a secret value by the constant zero. This is essentially a bookkeeping procedure where we try to reduce the annotation burden as much as possible. Of course, little is gained if the programmer blindly use these annotations to accept responsibility for the behavior of the pragmatic computation: The idea is that it will be easier to prove equivalence to the ideal computation when the compiler has verified that the program is well-annotated.

4.6 Conclusion

To conclude we have discussed the security guaranties related to SMCL programs. We operate with a particular strong model of adversary which we assume even has access to the physical machines executing the server. We provide security against any attack which is a function of the trace due to the notion of adversary traces. This include among others explicit flow, implicit flow, and timing attacks. To obtain these security guaranties we have carefully designed the semantics to avoid timing attacks and provided an extension to the well known type system based approach to noninterference which also allows assignment to variables defined in the same scope.

Chapter 5

Related Work

5.1 Introduction

To the best of our knowledge, SMCL is the first imperative programming language for general Secure Multiparty Computation. We discuss its relation to two other languages for SMC, and we survey the areas of language-based information-flow security and cryptography and explain their relationship to our work.

5.2 Languages for SMC

Closely related is the Fairplay project [31], which has developed a DSL for secure two-party computation (that is the special case of SMC where the number of parties is restricted to two). The Fairplay system consists of a compiler from the Secure Function Definition Language (SFDL) to one-pass boolean circuits described in the Secure Hardware Definition Language (SHDL). SFDL is a procedural DSL where all values are secret boolean, integer, or enumerations. SFDL also support arrays and the usual logic and arithmetic operations on booleans and integers except for multiplication and division on integers. The restriction to two parties and the use of boolean circuits as target greatly reduces the complexity of the runtime and the compilation. In contrast to the SFDL, SMCL allows both public/private and secret values which may potentially boost efficiency and allows general loops and recursive functions on public/private values. SMCL leaves the main burden of generating sound and efficient code to the compiler. Also, SFDL is restricted to the two-party scenario.

Another closely related language is the SMC language [45]. The language is a declarative language for SMC based on constraint programming. A public program is distributed among the parties in the computation along with an interpreter, each party inputs his secret values and the interpreter calculates the result. Computations are specified as arithmetic circuits and lacks branches on secret values. The computer of each party is considered secure in contrast to SMCL where the computation is done at the server parties, which we do not consider secure. SMCL is more expressive, offers stricter security guarantees, and provides a higher abstraction level.

5.3 Language-Based Security

Language based information-flow security aims at developing language mechanisms for protection against deliberate or accidental release of information. A thorough survey of language-based security is given by Sabelfeld and Myers in [40]. To SMCL the protection of confidential information is of vital importance and SMCL applies information-flow control to enforce security. Below we discuss areas of related work relevant to language-based security.

5.3.1 Noninterference

Denning [19] was the first to present a solution to the noninterference problem in terms of a static analysis which prevents explicit as well as implicit flow of information, however Denning did not provide a formal argument of noninterference.

Volpano and Smith [54] recast the work of Denning in terms of a type system for a simple imperative language and prove that well-typed programs obey the noninterference property. The type system is based on the lattice proposed by Denning. The partial ordering on the lattice extends naturally to a subtype relation. Types are divided into security levels and variable- and command types. A variable has a type τ-var if it contains values of security level τ or lower, while a command has type τ-cmd if no assignments occur in the command to a variable of security level lower than τ. The command-types are similar to the program counter label used by others like Sabelfeld in [57]. The work by Volpano and Smith has ignited a wide variety of work on the use of security types for enforcing noninterference. The term security types has been coined by Sabelfeld and Myers in [40] to denote annotation based approaches, e.g. type systems, to noninterference. The work has been extended in various directions to languages with first-order procedures [52], multiple threads [43, 46], and concurrent programs [53]. The type system based approach has been applied to wide range of settings like the calculi SLam [25] and DCC [1], the functional language FlowCaml [39], and recently VHDL [50].

Another approach to noninterference is the use of an “information-flow logic”. Amtoft et al. propose a Hoare-like logic [5,6] on top of which they present an interprocedural and modular information-flow analysis where noninterference is enforced as an end-to-end guarantee in object-oriented programs and programs with pointers. The logic supports programmer assertions that specify more precise information-flow policies. The technique has increased precision compared to previous type-based approaches like the ones by Volpano and Smith [52] and Zdancewic [57].

SMCL is a security-typed language which is firmly based on the work by Denning and by Volpano and Smith and is in line with the work done by others [23,40,50,57]. SMCL basically employs a two-level lattice of security levels, a type system based on [52, 54] (in the current implementation), which together with a semantics where the trace is independent of secret values to enforce noninterference.

In the decentralized label model (DLM) of [34], information is marked by labels. A label is a set of components consisting of an owner section and a reader section. The purpose of labels is to protect the confidentiality of the owner principals that may grant other principals the right to read their values. The DLM guarantees that the privacy of principals is never compromised. SMCL is also concerned with protecting the privacy of client input, and one could easily imagine that the DLM would be suitable for SMCL to guarantee that the values from some kind of clients do not flow to certain other clients. SMCL already has the notion of groups of clients and it seems like the combination of groups and DLM make an interesting match. We leave research into their synergies as future work.

5.3.2 Declassification

The noninterference property is often too restrictive in practice. Any practically interesting program leaks some kind of acceptable information, e.g. a password checker even leaks information when rejecting a candidate password. To accommodate this intentional leak of information, a way to lower or declassify the security level is needed. Allowing declassification without unintentional release of information has been the focus of recent attention and the paper by Sabelfeld and Sands [44] provides a good survey of declassification. According to the survey downgrading may be classified according to *what* information is revealed (The PER model [42], delimited release [41], relaxed noninterference [30], and quantitative abstractions [15]), by *whom* (The DLM and robust declassification [58]), *where* (non-disclosure [33]), and *when*.

Partial information flow analysis regulates what may be downgraded. The PER model [42] uses partial equivalence relations (a PER on a domain is an equivalence relation on a subset) to model the adversary's ability to distinguish between values. The PER model is powerful and captures a wide variety of approaches like delimited release [41], relaxed noninterference [30], and quantitative abstractions [15]. Downgrading in SMCL can quite possibly be formulated in a PER model. The programmer is alerted by the compiler of the possible implicit leaks which may result from a downgrade. An interesting future direction of research is to relate the warnings to the quantitative approach and deduce how much of the information is released.

Who is downgrading the data is important since an adversary may use the declassification mechanism to reveal more secure information than intended. The DLM prevents this by only allowing the owner to downgrade information as specified in the data security labels. This is resembling SMCL where a downgrade can only occur if all server parties (or at least a number of parties equal to the threshold) agree. Another approach is the *robust declassification* by Zdancewic and Myers. [58], where declassification may only be carried out by the designated owner of the data. In a later paper by Myers et al. [35], owner-ship information is used as integrity information, and declassification is deemed safe in areas of high integrity. The connection between integrity and owner-ship is further explored by Zdancewic, who use the DLM extended with integrity labels to determine robust declassifications in [57]. In SMCL all program points are of high integrity since an adversary may alter the computation completely and try to open exactly the secret value he wishes. He will not succeed unless he can corrupt a sufficiently large subset of server parties, in which case he would learn the secret anyway.

The approach by Mantel and Sands [32] based on intransitive noninterference and the non-disclosure approach by Matos and Boudol [33] are similar to downgrading in SMCL. The usual noninterference property does not hold in the presence of declassification, but as observed by both Mantel and Sands and Matos and Boudol the property may be enforced in maximal paths along which there is no downgrading, and then restart the bisimulation game in the context of any new low-equivalent stores. Our notion of adversary traces achieves the same goal using similar techniques: localization of declassification and enforcing the noninterference property between downgrades. The trivial information flow relation is left implicit in SMCL since we only have a two-level security lattice.

Information may be downgraded over time. The SMCR is based on computational security, so the values transmitted from clients to the server are encrypted using public-key cryptography. Thus an adversary may reveal these values if he has sufficient patience to break these cryptographic systems.

Recently there has been some effort to combine the four dimensions of declassification. Askarov and Sabelfeld [7] propose a *localized delimited release* as a combination of the *what* and *where* dimension.

5.3.3 Timing Attacks

Timing channels can present a serious threat. The problem of preventing timing attacks has received significant attention, and we will only consider those approaches closely related to SMCL.

Volpano and Smith [53] propose a notion of protected branches with atomic execution time. Their approach guarantees absence of timing leaks observable in the program, but does not prevent external timing leaks and forbid the use of loops in secret conditionals. Agat [3] observed that branches of secret conditionals must have the same timing characteristics in order to prevent timing attacks. Agat proposed to use transformation as a tool to remove timing attacks. In secret conditionals time parameters from the semantics are used to guide a cross padding of instructions with dummy instructions to ensure the same execution time of the two branches. The technique has inspired others like Barbosa and Page [9] who analyze functions (branches) to find the least set of dummy assignments that make their execution time equivalent. In some sense we employ the simplest possible variant of this approach: execute both branches in sequence and join the effects on the store. Our approach is potentially a lot slower than the approach by Barbosa and Page, but we cannot apply dummy assignments because the adversary may inspect the instruction pointer and thus learn which branch is being executed, so to eliminate this possibility we must execute both branches.

Tolstrup and Nielson [49] consider VHDL programs for which they define a semantic definition of security against timing attacks based on bisimulation and use a type system to enforce the condition. In SMCL there is no need for transformations and a type system is only needed to prevent loops on secret values. The lack of timing channels is vacuously true due to the semantics of SMCL. The model of Köpf and Basin [29] is a general and abstract semantic model based on automata for observable input and output, which is suitable for many situations but not entirely for SMCL, since the capability of the adversary is not just a function of the input and output, but also of the instruction pointer and state of the computation at any time.

5.4 Information-Flow Aware Languages

A number of general-purpose programming languages have been extended with support for information-flow security. The decentralized label model has been used as basis for the Java extension *JIF* [34] where the Java type system is extended with labels and principals. Other examples are FlowCaml [39] which is an extension of Caml with security-types, and information-flow inference for ML [39]. SMCL is similar to these in the sense of employing security types, but the goal of SMCL is to make it easy to write secure SMC programs.

In [59] Zdancewic proposes secure program partitioning as a means of allowing mutual distrusting hosts to execute a program. A program is partitioned into a number of slices according to security types and trust declarations. Confidentiality of information is obtained by restricting the computation on values to the host who owns the values or any host trusted by the owner. This has some resemblance to SMCL since both operate in a scenario of untrusted hosts, but whereas program partitioning is aiming at removing the need for a universally trusted host, SMCL realizes such a host through a combination of Secure Multiparty Computation and language-based security. In SMCL the need for trust can be totally eliminated by having each host execute a server party. A limitation of program partitioning seems to be that functions on confidential values similar to the Millionaires' Problem are not possible to compute without revealing the total net worth to the other millionaires.

The InCert project [18] suggests to develop a programming language that enables the development of secure applications operating on multiple data sources controlled by different principals without violating the security policies of the involved principals. SMCL may be viewed as realizing part of this ambitious goal for the special case of the strictest possible security policy where no principal must learn anything about any other.

5.5 Validation of Cryptographic Protocols

Much effort has been done in the area of validating cryptographic protocols [2, 22], and a domain-specific language for verifying such protocols has been proposed by Gordon and Jeffery [24].

A language which tackles similar security threats as SMCL is CAO [8]. CAO is a DSL for cryptographic software and it would be possible to implement the SMCR in CAO, thus ensuring the absence of timing attack.

5.6 Conclusion

We have presented an overview and discussed the relation between SMCL and related work. SMCL is an imperative programming language for general Secure Multiparty Computation in contrast to Fairplay which is only for two-party computations and SMC which lacks branches on secret values. SMCL applies techniques from within the area of language-based information-flow security in a novel way. Furthermore we have found no apparent relationship between SMCL and languages for verification of cryptographic protocols, besides that they may be used to verify the implementation of SMCR.

Chapter 6

Future Work

6.1 Introduction

This Chapter is devoted to ideas for future work. Some of the ideas are just rough thoughts that have emerged one or two times during the last year and some of them have had substantially more work put into them and are thus more developed. We will state the ideas and try to describe why they are interesting from the point of view of our thesis and computer science in general and how we may persue to develop them into scientific results.

6.2 SMCL

In Chapter 3 we described the current state of SMCL. Future development of SMCL is possible in a number of directions which falls into either of the categories: language design or security. In this Section we have not argued directly how the work mentioned supports our thesis, since it should be obvious.

Within the area of language design it is still an open answer how to handle compound data types and even arrays are not obvious. Thus a direction of future research would be to figure out how to make it possible to handle arrays. There are basically two parameters which can be secret regarding arrays (and most other data types) the size of the array and the content of the array. An array of secret length may be represented by padding with some random number of extra cells. The question then becomes if this is secure and how iteration or lookup should be performed. An array of secret values is easily implemented as in the current implementation, but how does one index into an array with a secret number is not obvious. Arrays of secret length and secret indexing are potentially useful and so increase the usefulness of SMCL.

SMCL would benefit from some kind of component system, which would allow already defined functions to be reused. This is a general problem for any programming language when in use and the programs written becomes complex. Such a mechanism posses additional challenges to SMCL besides those enjoyed by most other programming languages. E.g. a component containing functions on secret values should they be allowed to be used in clients and what should the semantics be? Furthermore if we chose to use the object oriented paradigm then a whole new set of possibilities for security must be taken into account [6].

Currently SMCL supports the notion of groups. Groups are currently static sets of clients. We would like to extend the concept to dynamic sets where a client may enter or leave a group based on the flow of the program. An extended group concept may be combined with an object oriented paradigm where different kinds of clients may share a common super type. Allowing groups to vary dynamically will enhance the power of SMCL and makes a more fine grained reasoning about who contributed with which values possible, and restrict the flow of values to and from clients. However such an extension also raise new security issues, e.g. is the current treatment of secret groups adequate.

How to support iteration though a collection of data has recently received some attention, starting with the enhanced for loop construction introduced in Java 5. Before that a lot of work has been done in the Haskel community where comprehensions based on monads [55] has proven to be a great tool. Comprehension based language constructs for iterating a collection of values has recently found its way into object oriented programming languages like Fortress [4]. Fortress has introduced the concept of generators which in essence are comprehensions. SMCL is generator-ready but lacks the machinery for user defined generators. Enhancing SMCL with generators based on a monads similar to Fortress would be an interesting direction for future work.

A more thorough formal treatment of SMCL especially a formalization of our notion of adversary traces would allow us to formally prove various security properties of SMCL, e.g. no timing leaks or noninterference. We would also like to prove the soundness of the typesystem sketched in this report and described in more detail in [37]. A formal treatment will give us solid testimony for the strong security of SMCL. We do not yet know how to adequately describe adversary traces formally or how they relate in details to other approaches. On the other hand it should be rather straightforward to prove the absence of timing leaks and prove the soundness of the typesystem.

Declassification is an essential mechanism in SMCL and the control of declassification is important. We have mainly been focusing on the *where* dimension of declassification but there seems to be advantages in combining other dimensions as well as suggested by Mantel and Sands [32]. As we noted in Section 5.3.2 one could in particular imagine that a combination of the *where* and *who* dimension (in the incarnation of the DLM) could be a nice match for SMCL. It is an open challenge to combine the four dimensions of declassification. Until recently [7], no one has studied the combination of dimensions and it would be interesting not only to SMCL but to area of language-based security in general to study the combinations of dimensions.

6.3 Secure Multiparty Computation for Relational Databases (SecRas)

Most data in modern society is contained in large databases, owned by companies or government organizations who might be reluctant to share or combine the information with others because it is too valuable or they are restrained from doing so because it would invade the privacy of clients. Many interesting examples of these scenarios exists e.g. company benchmarking without revealing crucial information to competitors or insurance agencies wanting to prevent security fraud by combining client databases, but are disallowed to do so because it would invade the privacy of all the law obeying costumers.

We propose to apply SMC to the database domain to obtain a solution which would allow organizations to combine databases in such a way that it becomes possible to answer queries on the joint data without revealing the data it self.

We imagine a use scenario where a number of users interact on behalf of the organizations they represent. Each organization may posses one or more databases containing secret or sensitive information. Each organization has some interest in combining the information in the databases, but do not want to reveal their own databases in their entirety. The organizations could be security agencies who wish to find a (hopefully) limited number of customers who are cheating in relation to their insurance, Or it might be a scientists who want to correlate information about illness with information about occupation and income. A user creates a query which involves secret information from a number of databases, the query is feed to a database system based on SMC, and a result is returned. If the database system is based on the relational model, then the result is a relation containing some tuples which contains public as well as secret cells of information. Any user may create any query involving any relation in any database to which he has been granted access, but in order to reveal the contents of secret cells the other users must accept this. An example could be the insurance agencies who would run a number of standard "fraud detection queries" on their combined data, but in order to reveal the relevant information of potential fraudsters they must have the accept from a government office. The insurance agencies may of cause meet in secrecy and exchange the information, but this is prohibited by Danish law, and thus the results are void.

We would like to develop a database system based on SMC to make the above scenario possible. Creating such a system with adequate security guaranties would be another testimony to our thesis, creating tools with strong security guaranties which exploits the benefits obtained by combining confidential information from various sources is feasible and useful. Furthermore it solves a practical problem which has potential for a huge economic impact on society as well as a great tool for data mining highly confidential or sensitive information. The development of such a tool is not trivial. A customized query language has to be developed where great care must be put into the design in order to ensure that any constructable query, except declassification queries, do not reveal too much information, for a suitable definition of too much.

A possible realization of a secure multiparty database system can be based on a relational algebra for Secure Multiparty Database Computation. The choice of the relational model as a basis has been taken because it is well studied and forms the foundation for many real world database systems. We are currently doing research on how to design a variant of E.F. Codd's relational data model suitable for a database system based on SMC. In particular there are a number of performance and security issues we must address. Relations should be shared among a number of users efficiently, we plan on using the SMCR for this. The security guaranties are multi-level. On the lowest level we must ensure that a user cannot singlehandedly reveal information he does not already own. This is a classic application of SMC. On the higher levels we must ensure things like indirect information release, e.g. from the size of the result set or from combining the results of different queries.

6.4 SVM

As part of the SIMAP project the SMCR has been developed as the foundation for SMC applications. The SMCR is currently implemented as Java API. The SMCR handles most aspects of SMC, like setting up communication between clients and server parties, handling client groups, and doing the actual secret computations. A more streamlined interface to the SMCR would be desirable and a possible direction of future work would be to turn the SMCR into a minimal virtual machine, with it's own interpreted language. The SMCR is the foundation on which our tools are build, and a more convenient interface in the form of a low-level SMC language would be preferable because the virtual machine could focus on making the basic SMC operations fast without having to consider concepts like client groups which is purely a SMCL concept. Design a low-level interpreted language suitable for various applications of the virtual machine is not trivial and would most likely be of interest in it's own right. Before we start the design process for such a language it would be an advantage to gain more understanding of how the virtual machine is to support various applications of SMC. We already have some knowledge of this from SMCL and we may gain additional knowledge from SecRas as described above. One of the challenges is going to be the definition of the interface between the virtual machine and the various applications, the knowledge from SMCL and SecRas will help us here. Furthermore we need to strip down the SMCR and enhance the SMCL compiler and SecRas system similarly.

6.5 SPL

During the development of the SMCR runtime it has been noted that there is a large gap between how the cryptographic protocols are described in research papers and their implementation. A possibility for future research is to develop a domain-specific language for cryptographic SMC protocols. Such a DSL would make it easier to implement and thus try out new protocols, which may increase efficiency of the runtime. Any improvement in the efficiency of the runtime immediately translates into more efficient tools for SMC. A number of domain-specific language for cryptographic protocols in general do already exist, but they are too general and we believe that a number of abstractions relevant only for SMC protocols can be introduced with the usual benefits [51]. By analyzing the domain of SMC protocols and the current literature on general languages for cryptographic protocols we should gain a solid foundation on which we should be able to identify the needed abstraction for a SMC protocol language.

6.6 Conclusion

We have listed a number of different direction for future work. Some of the ideas are easily realized other take more work and yet others must be left for others to finish due to restricted time. We are currently pursuing the enhancement of SMCL and the realization of SecRas. This does however not mean that the rest are left unattended but they are not our main priority.

Chapter 7

Conclusion

In this progress report we have documented the research carried out so far. We have among other things documented that it is feasible to create a domain-specific language for secure multiparty computation, which can be used to write programs which combines confidential information without compromising it.

We have created the Secure Multiparty Computation Language which provides high-level abstractions for secure multiparty computations along with strong security guaranties which protects against a broad spectrum of security threats. However we have also discussed a number of ideas for future research which document that there is ample work to be done yet.

We conclude that we are well under way to document our thesis: It is feasible and useful to create tools with strong security guaranties which exploits the benefits obtained by combining confidential information without compromising the information.

Bibliography

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[3] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53. ACM Press, 2000.

[4] Allen, Chase, Hallett, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt. The Fortress Language Specification. Technical report, SUN Microsystems, 2007.

[5] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow analysis of pointer programs. Technical Report CIS TR 2005-1, Kansas State University, July 2005.

[6] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 91–102, New York, NY, USA, 2006. ACM Press.

[7] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, June 14 2007.

[8] M. Barbosa, R. Noad, D. Page, and N. P. Smart. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005.

[9] M. Barbosa and D. Page. On the automatic construction of indistinguishable operations. In *IMA Int. Conf.*, pages 233–247, 2005.

[10] D. J. Bernstein. Cache-timing attacks on AES, 2004.

[11] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications. Technical Report RS-05-18, BRICS, June 2005. 37 pp.

[12] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Proc. of Financial Cryptography*, volume 4107 of *LNCS*. Springer-Verlag, 2006.

[13] D. Brumley and D. Boneh. Remote timing attacks are practical. *Comput. Networks*, 48(5):701–716, 2005.

[14] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 136–145, 2001.

[15] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *J. Theoretical Computer Science*, 59(3):1–14, Jan. 2004.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[17] R. Cramer and I. Damgård. Multiparty computation, an introduction, 2004.

[18] K. Crary, R. Harper, F. Pfenning, B. C. Pierce, S. Weirich, and S. Zdancewic. Manifest security for distributed information. White paper, http://www.cis.upenn.edu/˜ bcpierce/papers/incertproposal06.pdf, Mar. 2006.

[19] D. E. R. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[20] D. E. R. Denning. *Cryptography and Data Security.* Addison-Wesley, Boston, MA, USA, 1982.

[21] D. E. R. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[22] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Sci. Comput. Program.*, 50(1-3):73–99, 2004.

[23] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[24] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. CSFW 15*, 2002.

[25] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[26] T. Jakobsen and S. From. Secure multi-party computation on integers. Master's thesis, Department of Computer Science, DAIMI, University of Aarhus, Denmark, July 2005.

[27] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 303–310. ACM Press, 1991.

[28] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. International Cryptology Conference on Advances in Cryptology*, volume 1109 of *LNCS*, pages 104–113, London, UK, 1996. Springer-Verlag.

[29] B. Köpf and D. A. Basin. Timing-sensitive information flow analysis for synchronous systems. In *Proc. European Symp. on Research in Computer Security*, pages 243–262, 2006.

[30] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[31] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *Proc. of USENIX Security Symposium*, pages 287–302, 2004.

[32] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *Proc. of the ASIAN Symposium on Programming Languages and Systems*, volume 3303 of *LNCS*, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[33] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society Press.

[34] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[35] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, 2006.

[36] J. D. Nielsen and M. I. Schwartzbach. A Domai-specific Programming Language for Secure Multiparty Computation. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, June 14 2007.

[37] J. D. Nielsen and M. I. Schwartzbach. The SMCL Language Specification. Technical Report RS-07-9, BRICS, Apr. 2007.

[38] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[39] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, 2002.

[40] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21, 2003.

[41] A. Sabelfeld and A. Myers. A model for delimited information release. In *Proc. of the International Symposium on Software Security*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[42] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symposium on Programming*, pages 40–58, 1999.

[43] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, page 200, Washington, DC, USA, July 2000. IEEE Computer Society Press.

[44] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society Press.

[45] M. C. Silaghi. SMC: Secure Multiparty Computation language, 2004. `http://www.cs.fit.edu/msilaghi/SMC/tutorial.html`.

[46] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, New York, NY, 1998.

[47] D. R. Stinson. *Cryptography Theory and Practice, Third Edition.* Chapman & Hall/CRC, 2006.

[48] T. Toft. Progress report - Secure Integer Computation with Applications in Economics., July 2005.

[49] T. K. Tolstrup and F. Nielson. Analyzing for Absence of Timing Leaks in VHDL. In D. Gollmann and J. Jürjens, editors, *Proc. International Workshop on Issues in the Theory of Security*, Mar. 2006.

[50] T. K. Tolstrup, F. Nielson, and H. R. Nielson. Information Flow Analysis for VHDL. In V. E. Malyshkin, editor, *Proc. International Conference on Parallel Computing Technologies*, volume 3606 of *LNCS*, pages 79–98. Springer-Verlag, Sept. 2005.

[51] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[52] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of Theory and Practice of Software Development*, pages 607–621, 1997.

[53] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1998.

[54] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[55] P. L. Wadler. Comprehending monads. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 61–78, New York, NY, 1990. ACM Press.

[56] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 160–164, 1982.

[57] S. Zdancewic. A type system for robust declassification. In *Proc. of the Mathematical Foundations of Programming Semantics*, Mar. 2003.

[58] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, June 2001.

[59] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. Technical Report 2001-1846, Cornell University, 2001.

Appendix A

Syntax and Terminology

The various constructs of the abstract syntax tree (AST) are typeset using different fonts:
terminals, **nonterminals**, *metavariables*, `variables`, and sets.

In the following we present the symbols we use for various concepts in the language.

- Program: P
- Client: κ
- Client declaration: K
- Server: σ
- Server declaration: Σ
- Server or client: ξ
- Store: The server and each client has a local store, the stores are denoted $\sigma.S$ and $\kappa.S$ respectively. Furthermore we use S to range over stores when we do not care which specific server/client store it is related to. We write $\sigma.S[\mathtt{x} \mapsto v]$, $\kappa.S[\mathtt{x} \mapsto v]$, or $S[\mathtt{x} \mapsto v]$ for the stores where the location denoted by $\mathtt{x}$ is updated with the value of v, where $\mathtt{x}$ is a variable in the current scope.
- Global store: G
- Field: $\mathtt{d}$
- Field declaration: D
- Function names: $\mathtt{f}$
- Function declaration: F
- Command: C
- Expression: e
- Values: Values are separated into public and secret values. Secret values, the values in the sets SBoolean, SClient and SInteger, are denoted by $\boxed{v}$ and public values are denoted by v. A value which is either public or secret is denoted by w.
- Variable: $\mathtt{x}$

- Tunnels: θ. The operator @ is used for concatenation of two tunnels. Values may be taken out of the tunnel in two ways: θ.get() is non-blocking and if the tunnel is empty returns the special value Null. θ.take() is blocking and if the tunnel is empty waits until a value becomes available. Values may be placed in the tunnel using θ.put(v).

- Record name: `r`

- Record declaration: R

- Labels: $\ell \in \mathcal{L}_R$ denotes the set of labels in the kind of tuples declared by R.

- Types: τ denote types, defined in [37],

- Integers: i, n.

- Generator: ω, a generator is a value which may produce some other values when the next() function is applied and more values are available in the generator. Furthermore a generator may be detected to be empty using the empty() function.

- Operators: *op*. +,-,*,/,&&,||,=,<>,>=,<= with the usual semantics.

- SBoolean: contains the two integer constants 0 and 1. Where 0 denotes false and 1 denotes true. In the concrete and abstract syntax, 0 and 1 are written false and true, respectively.

In Figure A.1 we present the overall abstract syntax of SMCL. The client-side and server-side abstract syntax is presented in Figure A.2 and A.3 respectively.

$$\text{Program} \quad ::= \quad K^{+}\ \Sigma$$

Figure A.1: SMCL abstract syntax and values

$$\begin{array}{lll}
K & ::= & \textbf{Declare Client}\ \texttt{name}\textbf{:}\ D^*\ F^+ \\
F & ::= & \textbf{function}\ \tau\ \texttt{f}\ (\tau_1\ x_1,\ldots,\tau_n\ x_n)\ \{\ C\ \} \\
D & ::= & \tau\ \texttt{x} = e \\
 & | & \textbf{Tunnel of}\ \tau\ \texttt{x} \\
\mathrm{C} & ::= & \tau\ \texttt{id} \\
 & | & \tau\ \texttt{x} = e \\
 & | & \textbf{while}\ (\ e\ )\ \{\ C\ \} \\
 & | & \textbf{if}\ e\ \{\ C_1\ \}\ \textbf{else}\ \{\ C_2\ \} \\
 & | & C_1; C_2 \\
 & | & \textbf{return}\ e \\
 & | & \textbf{return} \\
e & ::= & \mathrm{Value} \\
 & | & \texttt{x} \\
 & | & \texttt{f}(e_1,\ldots,e_n) \\
 & | & \textbf{fun-app}(C) \\
 & | & \theta.\mathrm{put}(e) \\
 & | & \theta.\mathrm{get}(e) \\
 & | & \theta.\mathrm{take}(e) \\
 & | & \textbf{display}(e) \\
 & | & \textbf{readint}() \\
 & | & \mathrm{new}\ \texttt{id}(e_1,\ldots,e_n) \\
 & | & \mathrm{new}\ \tau[e] \\
 & | & e_1\ \mathit{op}\ e_2 \\
 & | & e_1\ ?\ e_2\ :\ e_3 \\
\mathrm{op} & ::= & + \mid \texttt{-} \mid * \mid / \mid == \mid \&\& \mid ||\ |=|>|>=|<=|<
\end{array}$$

Figure A.2: Client-side SMCL abstract syntax and values

$$
\begin{array}{lcl}
\Sigma & ::= & \textbf{Declare Server } \texttt{name}\textbf{:}\ D^{*}\ F^{+} \\
F & ::= & \textbf{function}\ \tau\ \texttt{f}\ (\tau_1\ x_1,\ldots,\tau_n\ x_n)\ \{\ C\ \} \\
D & ::= & \tau\ \texttt{x} = e \\
 & | & \textbf{Group of}\ \texttt{name}\ \texttt{x} \\
C & ::= & \tau\ \texttt{id} \\
 & | & \tau\ \texttt{x} = e \\
 & | & \textbf{while}\ (\ e\ )\ \{\ C\ \} \\
 & | & \textbf{for}\ (\ \tau\ \texttt{x}\texttt{:}\ e\ )\ \{\ C\ \} \\
 & | & \textbf{if}\ e\ \{\ C_1\ \}\ \textbf{else}\ \{\ C_2\ \} \\
 & | & C_1; C_2 \\
 & | & \textbf{return}\ e \\
 & | & \textbf{return} \\
e & ::= & \text{Value} \\
 & | & \texttt{x} \\
 & | & \texttt{f}(e_1,\ldots,e_n) \\
 & | & \textbf{fun-app}(C) \\
 & | & e.\texttt{f}(e_1,\ldots,e_n) \\
 & | & \textbf{open}(e|\texttt{x}_1,\ldots,\texttt{x}_n) \\
 & | & \theta.\text{put}(e) \\
 & | & \theta.\text{get}(e) \\
 & | & \theta.\text{take}(e) \\
 & | & \text{new}\ \texttt{r}(e_1,\ldots,e_n) \\
 & | & \text{new}\ \tau[e] \\
 & | & e_1.e_2 \\
 & | & e_1\ \mathit{op}\ e_2 \\
 & | & e_1\ ?\ e_2\ :\ e_3 \\
\mathit{op} & ::= & +\ |\ \text{-}\ |\ *\ |\ /\ |\ ==\ |\ \&\&\ |\ ||\ |{=}|{>}|{>=}|{<=}|{<}
\end{array}
$$

Figure A.3: Server-side SMCL abstract syntax and values