A Domain-Specific Programming Language for Secure Multiparty Computation

Janus Dam Nielsen [*] Michael I. Schwartzbach

BRICS, University of Aarhus, Denmark

{jdn, mis}@brics.dk

Abstract

We present a domain-specific programming language for Secure Multiparty Computation (SMC).

Information is a resource of vital importance and considerable 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. This fundamental conflict between the benefits of confidentiality and the benefits of information sharing may be overcome using the cryptographic method of SMC where computations are performed on secret values and results are only revealed according to specific protocols.

We identify the key linguistic concepts of SMC and bridge the gap between high-level security requirements and low-level cryptographic operations constituting an SMC platform, thus improving the efficiency and security of SMC application development. The language is implemented in a prototype compiler that generates Java code exploiting a distributed cryptographic runtime.

Categories and Subject Descriptors D.3.2 [*Domain-Specific Languages*]: Secure Multipart Computation

General Terms Languages, Design, Security

Keywords SMCL, design, analysis, implementation

1. Introduction

Information is a resource of vital importance and considerable economic value to individuals, public administration, and private companies. This means that confidentiality, i.e. the protection of confidential information from unwanted leakage, is an important security issue. At the same time, however, it is often possible to obtain significant added value by combining confidential information from different sources. The promise of Secure Multiparty Computation (SMC) is to get the best of both worlds: the advantages of information sharing without the risks of unwanted leakages.

The seminal example of SMC is the *Millionaries' Problem*, which involves a number of millionaires who want to find out which is richer, but all refuse to disclose their net worth. A conventional solution would involve an external trusted party that could perform the comparisons and report the result. Yao [46] presented a cryptographic solution for two millionaires that does not require an external party or any degree of trust between the two parties. The technique is essentially to perform computations on the data while it is encrypted and to control strictly when and how the resulting information is revealed. These techniques have subsequently been extended to cover comparisons, arithmetic, and bitwise operations [14].

There are significant benefits from eliminating the reliance on trusted parties. The only known method for making external parties trustworthy is essentially to compensate them so adequately that the temptation to betray other parties is minimized. This imposes a significant overhead that prohibits their involvement in many situations. In parody, trusted parties generally receive huge fees for opening envelopes and announcing the highest bids.

The SIMAP (Secure Information Managing and Processing) project aims to make SMC a practical and inexpensive technique for complex applications. Examples include distributed voting, private bidding and auctions, and business processes such as matchmaking and financial benchmarking [9]. The SIMAP project has three main components: 1) the development of an efficient cryptographic runtime system (SMCR) supporting the required primitive operations, 2) a domain-specific high-level language (SMCL) for specifying computations that are compiled into distributed applications based on SMCR, and 3) the development and deployment of large-scale applications in collaboration with industrial partners. This work presents the initial design and implementation of the SMCL language.

2. Contributions and Outline of the Paper

This paper explores the uncharted area of SMC programming and provides four main contributions:

- A conceptual analysis of the domain of SMC programming.
- A design and implementation of SMCL, a novel domain-specific programming language for SMC.
- A definition of the security properties that SMCL programs satisfy.
- A number of static analyses that ensure these security properties or boost efficiency of SMCL applications. We also present a simple language of checked annotations for describing potential information dependencies among variables in SMCL programs.

The rest of the paper is organized as follows. In Section 3 we briefly present the structure of the SMCR system. In Section 4 we perform a conceptual analysis of the linguistic elements of SMC. In Section 5 the various design decisions of SMCL are presented and

[*] Supported by the SIMAP Project under the Danish Strategic Research Council NABIIT-program

discussed. In Section 6 we discuss the security guarantees that are provided for SMCL programs and whose soundness is ensured by static analyses. In Section 7, we evaluate the cost of using SMCR and estimate how further static analyses may boost efficiency. A description of related work is given in Section 8, and we offer some concluding remarks in Section 9 and highlight a number of interesting areas for future work.

3. Secure Multiparty Computation

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 $P_1, \dots, P_n$ that wish to jointly compute the value of an integer function $f(x_1, x_2, \dots, x_n)$, where party P_i only knows the input value x_i which must be kept secret from the other parties.

SMCR (a further development of the system used in [8]) 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 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 [8]. Thus, simple operations become several orders of magnitude more expensive than their non-cryptographic counterparts, as seen in Section 7. 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 in this paper, but refer to [8].

4. Conceptual Analysis

Based on previous experiences with earlier versions of SMCR [9, 20, 39], 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 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 3, 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.

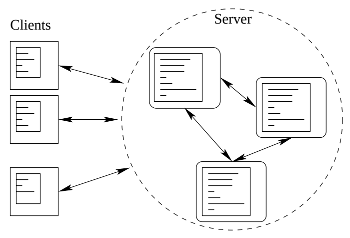


Figure 1. Conceptual and concrete view of clients and server

A given physical machine may run a client and a server party simultaneously which is useful if the owner of the machine do not trust anybody else, e.g. in game of poker.

Clients communicate with the server only and have no incentive to communicate directly with each other, since they generally do not trust each other.

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 correspond 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 a classical examples [15, 21]. 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.

In summary, 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.

5. Secure Multiparty Computation Language

Based on the above key concepts we have designed a novel language called the *Secure Multiparty Computation Language* (SMCL). It is a highlevel, domain-specific language [42], 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.

SMCL will be presented based on the example in Figure 2, 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`). A tunnel supports asynchronous communicating that is encrypted using the public key of the receiver. 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).

The server declares that `Millionaries` may belong to a group named `mills` (line `S2`). The member of a group is specified externally by a mapping supplied to the SMCR runtime describing the concrete participants involved during runtime. Figure 3 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.

The `main` function of the server describes the SMC application that is executed jointly by all the server parties.

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 inte-

```
 C1: declare client Millionaires:
 C2:
 C3:  tunnel of sint netWorth;
 C4:
 C5:  function void main(int[] args) {
 C6:   ask();
 C7:  }
 C8:
 C9:  function void ask() {
C10:   netWorth.put(readInt());
C11:  }
C12:
C13:  function void tell(bool b) {
C14:   if (b) {
C15:    display("You are the richest!");
C16:   }
C17:   else {
C18:    display("Make more money!");
C19:   }
C20:  }
```

```
 S1: declare server Max:
 S2:  group of Millionaires mills;
 S3:
 S4:  function void main(int[] args) {
 S5:
 S6:   sint max = 0;
 S7:   sclient rich;
 S8:
 S9:   for (client c in mills) {
S10:    sint netWorth = c.netWorth.get();
S11:    if (netWorth > max) {
S12:     max = netWorth;
S13:     rich = c;
S14:    }
S15:   }
S16:
S17:   for (client c in mills) {
S18:    c.tell(open(c==rich|rich));
S19:   }
S20:  }
```

Figure 2. The generalized Millionaries' Problem in SMCL

```
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. A map identifying the concrete participants

gers 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).

In clients, the types of tunnels and return types of functions may be secret (as in line `C3`). 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.

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.

In line `S11` a secret boolean is used as condition in an **`if`** statement. This is an instance of implicit flow [16] and could potentially leak the secret value, if the two branches could be observed to behave differently, e.g. by timing the execution of each branch. Consequently, we enforce a number of requirements for conditionals on secret values which we describe in Section 6. Similarly unintended information may leak from **`while`** loops with secret conditionals and recursive functions which recur based on secret conditions. **`while`** loops cannot be handled in the same way as conditionals and are thus not allowed. Recursive functions may take secret values as argument but cannot recur based on secret conditionals due to the semantics of SMCL similar calls to recursive functions are not allowed within secret conditionals. Conditionals and **`while`** loops on private and public values are allowed without any restrictions.

To finish, 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. The operator is generally used as **`open`**`(e|x,y,z)` which 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`. In line `S18`, opening the value of the comparison `c==rich` may leak some knowledge about the value of `rich` as described below. A program cannot be compiled unless it is *well-annotated*, meaning that the programmer has recognized all potential leaks (see Section 6 for further details).

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 4(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 4(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 4(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

```
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 4. The combinations of server knowledge

recognize responsibility for compromising it (ever so slightly). In a yet stricter version, we may change the three lines into the lines of Figure 4(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).

Figure 5 shows another example program which implements a so-called clock auction [23], where producers of electric power bid for shares of a total number of MegaWatts that a consumer requires. This example highlights the benefit of being able to iterate through what is essentially a sequence of individual SMC applications. A number of other examples such as double auction, multi-round auction, the stable marriage problem, the Miller-Rabin primality test and the k-means-clustering algorithm have all been implemented using SMCL.

The prototype compiler produces Java code using the SMCR API, for each kind of client and for the server parties. 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.

6. Security

As discussed in Section 8, security requirements are many and multidimensional. Also, the problems to be considered depend heavily on the capabilities that an adversary are assumed to possess [11, 22].

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

```
declare client Producer:

 function void main(int[] args) {}

 function sint getBid(int price) {
   display(price);
   return readInt();
 }

 function void result(int price) {
   display(price);
 }

declare client Consumer:

 function void main(int[] args) {}

 function void result(int price) {
  display(price);
 }

declare server Auction:

 function void main(int[] args) {
  group of Producer supply;
  group of Consumer demand;

  int totalMW = 7;
  int shares = 14500000;
  int price = 10;

  bool done = false;
  while(!done) {
   sint supply = 0;

   for (client c in supply) {
    sint bid = c.getBid(price);
    supply = supply + bid;
   }

   if (open(supply > totalMW))
    price = price - 1;
   else
    done = true;
  }

  for (client c in supply) {
   c.result(price);
  }

  for (client c in demand) {
   c.result(shares*price);
  }
 }
```

Figure 5. Clock auction written in SMCL

e.g. simple eavesdropping and more complex attacks which are a function of the program trace, like interference, and timing [18, 21].

Adversary Traces

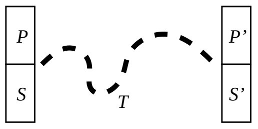
To formally define these notions, we have provided a small-step operational semantics of SMCL programs [29]. Here, the state of an entire system contains the state of the server, the state of each client, and the state of each tunnel. The semantic relation reflects the computational progress of the clients, the server, and their communications. Stores may contain both public, secret, and private values.

We consider only *well-typed* programs [29], which have the simple property that variables with public types can never contain secret values [38, 43, 45]. In the present SMCL language we only have two security levels [15]: *public* and *secret* with distinct run-time representations. However, it would be natural to extend this to the lattice of subsets of client groups, such that a secret value is owned by a subset of the clients in the line of the decentralized label model [28].

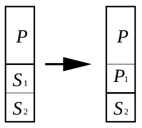
To formalize our security guarantees, we introduce a notion of *adversary traces*, which contain the information that is made available to an adversary. Such a trace consists of the entire sequence of system states (configurations in the small-step semantics) that is encountered during the evaluation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

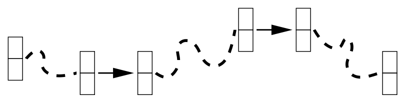
The capabilities of an adversary are then limited to observing these traces. We will use an illustration to show an adversary trace T from a state with public values P and secret values S to one with public values P' and secret values S':



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:



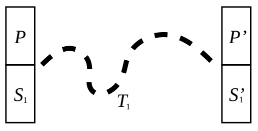
A complete computation that occasionally makes use of the **open** operation for downgrading is then described by an alternating sequence of adversary traces and these transitions:

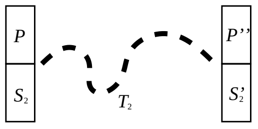


The security guarantees of well-typed SMCL programs can now be expressed through two properties that will be ensured by the compiler.

The Identity Property

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



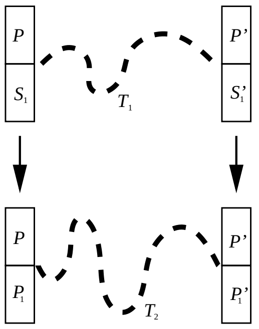


then $T_1 = T_2$ (and thus also $P' = P''$). This is a strong property stating that computations from initial states with the same public values will have *identical* observable traces from the point of view of the adversary. This implies the property of *noninterference*, which normally only requires that the resulting public values must be equal [32].

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 [7, 21] and also more exotic attacks, e.g. based on measuring radiation from the server [10]. 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** statement with a secret conditional are always evaluated in sequence; and
- static analyses of well-typed SMCL programs to verify that such branches always terminate and have no public side-effects.

The generated code for an **if** statement with a secret conditional will always execute both branches on copies of the local state. After these executions, the results of both branches are merged based on the secret boolean value of the conditional. For example, if a secret variable `x` is represented by the variable $\texttt{x}_{\texttt{then}}$ in the first branch and the variable $\texttt{x}_{\texttt{else}}$ in the second branch, then the merging of the two states is performed by the secret computation:

$$\texttt{x} = \texttt{condition}*\texttt{x}_{\texttt{then}} + \texttt{(1-condition)}*\texttt{x}_{\texttt{else}}$$

This is 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).

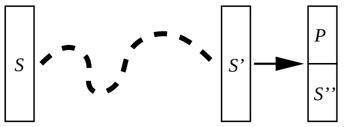
Furthermore, since the merging of the two branches cannot undo side-effects, we need to make sure that they agree on all public side-effects. This includes assignments to public variables with scope outside the branches, function calls, IO, and communication with clients. To this end, the SMCL compiler 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 [38, 43, 45]. 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, the static analysis that approximates hoistability is based on alias analysis, def-use analysis, and side-effect analysis of functions [30].

A similar solution is not possible for **while** loops on secret conditionals and calls to recursive functions which recur based on secret conditions, and are consequently not allowed. However, 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.

Semantic Information Leaks

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 Section 7, 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. In Figure 6 (ideal version) we open the sign of a polynomial evaluated at a given point, and should mention all of the variables `a,b,c` and `p` but when arguing for the security of releasing `p` one must consider it's constituets `a,b` and `c` thus for ease of annotation we also subtract the variables which may have influence any allready mentioned variable. The corresponding analysis is a mixture of a def-use analysis, a liveness analysis, and an available expressions analysis [30]. 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

```
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

Figure 6. Three versions of the polynomial program

(parties, threshold)	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 7. Timing results in SMCR

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.

7. 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 6. 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 7, 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.

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 [13] based on fundamental analyses described in [30], but they are interesting because they solve important domain-specific problems and thus illustrate the benefits of using a domain-specific language.

Overflow Checking

SMCR is initialized with a large prime number p and all secret integer operations are performed modulo p. Thus, the runtime must insert overflow checks to ensure a sound semantics. Such checks are expensive in SMCR comparable to a comparison operation, and a straightforward interval analysis will generally be too conservative to eliminate many of those. Thus, we allow size annotations such as **`sint`**`[16]` and **`int`**`[2]` to state the maximal size in bits of the corresponding values. We also allow types such as **`sint`**`[%]` meaning that the values are actually represented modulo p (since this is sometimes used in SMC applications). With such annotated types it becomes more feasible to perform a static analysis with good precision that tries to infer the sizes of integers. The compiler then only needs to insert an overflow check if a computed integer value may exceed p and if, furthermore, it is used before being stored in a variable with type annotation `[%]`.

Batch Processing

Multiplication and comparison of secret integers are particularly expensive operations, mainly because of numerous communication rounds taking place between the server parties. This expense may be reduced by performing several such operations in batches so that they share the communication overhead. For example, we use a static analysis to detect pairs (or generally tuples) of multiplications that may be bundled in this manner. It is a variation of an available expression analysis that bundles `a*b` with `c*d` if it can be guaranteed that the values of `c` and `d` do not change between the evaluation of `a*b` and the occurrence of `c*d`. As an example, this optimization applied to 100 multiplications reduces the running times as shown in Figure 8.

Multiplications are frequent in SMC applications. However, they often occur on array entries, and it poses more of a challenge to bundle multiplications such as `a[i]*b[j]` and `c[k]*d[l]` since an advanced integer analysis must first be performed.

(parties, threshold)	sequential	batch
(3,1)	1,002 ms	117 ms
(5,2)	2,298 ms	333 ms
(7,3)	4,880 ms	1,210 ms

Figure 8. Timing results of multiplications

Representation Heuristics

Secret integers may alternatively be represented at runtime as a sequence of secret bit values. Not surprisingly, this representation is much more efficient when bitwise operations are performed. However, the overhead between changing representation is quite large. We are experimenting with heuristics that identify points in the control-flow graph where it may pay off to toggle the representation of a given secret integer variable.

8. Related Work

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 briefly survey the areas of language-based information-flow security and cryptography and explain their relationship to our work.

Languages for SMC

Closely related is the Fairplay project [25], 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 [37]. 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.

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 [32]. 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.

Noninterference 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 [18, 32, 41, 47]. SMCL basically employs a two-level lattice of security levels, a type system based on [43, 45] (in the current implementation), which together with a semantics where the trace is independent of secret values to enforce noninterference. The work by Volpano and Smith has ignited a wide variety of work on the use of security types for enforcing noninterference. The work has been extended in various directions to languages with first-order procedures [43], multiple threads [35, 38], and concurrent programs [44]. The type system based approach has been applied to wide range of settings like the calculi SLam [19] and DCC [1], the functional language FlowCaml [31], and recently VHDL [41].

Another approach to noninterference is the use of an "information-flow logic". Amtoft et al. propose a Hoare-like logic [4, 5] 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 [43] and Zdancewic [47].

In the decentralized label model (DLM) of [28], 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.

Downgrading 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 [36] provides a good survey of declassification. According to the survey downgrading may be classified according to *what* information is revealed (The PER model [34], delimited release [33], relaxed noninterference [24], and quantitative abstractions [12]), by *whom* (The DLM and robust declassification [48]), *where* (non-disclosure [27]), and *when*.

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.

The downgrading in SMCL is somewhat related to the DLM and robust declassification in that a downgrade can only occur if all server parties (or at least a number of parties equal to the threshold) agree, but a downgrade may occur based on input received from a possible corrupted client and thus a client may control what information is revealed. SMCL guaranties that the given server program is executed according to the semantics of SMCL and that no information is leaked as a product of the execution with the except of any information explicitly declassified using the **open** operator.

The approach by Mantel and Sands [26] based on intransitive noninterference and the non-disclosure approach by Matos and

Boudol [27] are similar to downgrading in SMCL. The usual non-interference 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.

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 [44] 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 [6] 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 [40] 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 [22] 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.

Secure Program Partitioning

In [49] 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 is 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 based on SMC. 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 net worth to the other millionaires.

Validation of Cryptographic Protocols

SMCL is a domain-specific language for SMC applications and uses a cryptographic runtime with several cryptographic protocols which could conceivably be verified using techniques for validating cryptographic protocols [2, 17].

9. Conclusion and Future Work

We have presented SMCL, the first imperative domain-specific language for programming SMC applications. The language design expresses a conceptual analysis of the application domain and uses a host of static analyses to ensure security and to boost efficiency. A prototype compiler has been written which translates SMCL to Java exploiting a distributed cryptographic runtime (SMCR). The SMCL design is based on initial experiences with SMC programming, but with this platform we plan to perform extensive experiments with larger applications, mainly in the area of e-commerce and business processes. The information gained from such usage of SMCL forms a suitable basis for the next generation of the language. Many ideas for improvements already exist, but only practice will give a sound basis for prioritizing these.

Acknowledgments

The authors would like to thank the anonymous referees for valuable comments and the participants of the SIMAP project for providing valuable insight into the area of SMC along with many ideas and fruitful discussions. We especially thank Thomas Jakobsen for his great work on realizing the SMCR. We also thank Claus Brabrand, Christian Kirkegaard, Anders Møller, and Johnni Winther for comments and discussions related to this work.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon 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] Martín Abadi and Andrew 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] Johan Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53. ACM Press, 2000.

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

[5] Torben Amtoft, Sruthi Bandhakavi, and Anindya 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.

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

[7] Daniel J. Bernstein. Cache-timing attacks on AES, 2004.

[8] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas 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.

[9] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Proc. of Financial Cryptography*, volume 4107 of *LNCS*. Springer-Verlag, 2006.

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

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

[12] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *J. Theoretical Computer Science*, 59(3):1–14, January 2004.

[13] Patrick Cousot and Radhia 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.

[14] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. Theory of Cryptography Conference*, volume 3876 of *LNCS*, pages 285–304. Springer-Verlag, May 2006.

[15] Dorothy Elizabeth Rob Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[16] Dorothy Elizabeth Rob Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

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

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

[19] Nevin Heintze and Jon 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.

[20] Thomas Jakobsen and Strange From. Secure multi-party computation on integers. Master's thesis, Department of Computer Science, DAIMI, University of Aarhus, Denmark, July 2005.

[21] Paul 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.

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

[23] Vijay Krishna. *Auction Theory*. Academic Press, 2002.

[24] Peng Li and Steve 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.

[25] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - A Secure Two-Party Computation System. In *Proc. of USENIX Security Symposium*, pages 287–302, 2004.

[26] Heiko Mantel and David 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.

[27] Ana Ameida Matos and Gerard 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.

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

[29] Janus Dam Nielsen and Michael I. Schwartzbach. The SMCL Language Specification. Technical Report RS-07-9, BRICS, April 2007.

[30] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

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

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

[33] Andrei Sabelfeld and Andrew 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, October 2004.

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

[35] Andrei Sabelfeld and David 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.

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

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

[38] Geoffrey Smith and Dennis 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.

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

[40] Terkel K. Tolstrup and Flemming Nielson. Analyzing for Absence of Timing Leaks in VHDL. In Dieter Gollmann and Jan Jürjens, editors, *Proc. International Workshop on Issues in the Theory of Security*, March 2006.

[41] Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson. Information Flow Analysis for VHDL. In Victor E. Malyshkin, editor, *Proc. International Conference on Parallel Computing Technologies*, volume 3606 of *LNCS*, pages 79–98. Springer-Verlag, September 2005.

[42] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

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

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

[45] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

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

[47] Steve Zdancewic. A type system for robust declassification. In *Proc. of the Mathematical Foundations of Programming Semantics*, March 2003.

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

[49] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. Technical Report 2001-1846, Cornell University, 2001.