An Information-Flow Type-System for Mixed Protocol Secure Computation

Florian Kerschbaum
SAP Applied Research
Karlsruhe, Germany
florian.kerschbaum@sap.com

ABSTRACT

There are a number of domain-specific programming languages for secure computation. Out of those, the ones that are based on generic programming languages support mixing different protocol primitives and enable implementing a wider, possibly more efficient range of protocols. On the one hand, this may result in better protocol performance. On the other hand, this may lead to insecure protocols. In this paper we present a security type system that enables mixing protocol primitives in a generic programming language, but also ensures that well-typed programs are secure in the semi-honest model. Consequently, a compiled protocol must be secure. We show an extension of the L1 language with our security type system and evaluate the implementation of two protocols from the literature. This shows that our type system supports the provably secure implementation even of complex protocols.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Security, Programming Languages

Keywords

Secure Two-Party Computation, Type System, Programming, Domain-Specific Language, Information Flow

1. INTRODUCTION

Secure (two-party) computation [37] allows two parties to compute a function f over their joint, private inputs x and y, respectively. No party can infer anything about the other party's input (e.g. y) except what can be inferred from one's own input (e.g. x) and output (e.g. $f(x, y)$). Secure computation has many applications, e.g. in the financial sector, and has been successfully deployed in commercial and industrial settings [5, 6, 7, 23].

Secure computation protocols are notoriously difficult to implement. First, they can encompass arbitrary functionality in the joint computation. Second, they need to follow a rigorous approach to security. Often special protocols – mixing several different primitives – are developed for important problems. This is expected to provide more efficient protocols due to insight into the problem domain [16]. These protocols then require a manual verification and security proof.

Current domain-specific programming languages for secure computation [2, 4, 9, 18, 19, 28, 35] do not adequately address these problems. Either they are tied to a specific protocol [2, 4, 18, 28] or they enable implementing insecure protocols [9, 19, 35]. On the one hand, if they are tied to a specific protocol, then this protocol may be proven secure manually independent of the functionality. Such a proof extends to all protocols implemented in this language, but the language prevents implementing many special, possibly more efficient protocols. On the other hand, if the programming language is built upon a generic programming language, such as Python [9] or Java [19, 35], all special protocols can be implemented. Yet, this allows the programmer to also implement insecure protocols that do not withstand security verification.

In this paper we address this problem in a novel way. We build upon the L1 language [35] which is a domain-specific language extension for secure computation based on Java that allows mixing several different protocol primitives. Although we use this specific language, our approach is generic and can be adapted for any domain-specific language that allows mixing protocols and is not tied to a protocol, e.g. [9, 19]. We augment the L1 language with a novel security type system. This type system provably ensures that well-typed programs are secure in the semi-honest model of secure computation. Loosely speaking, in the semi-honest model – as defined by Goldreich [15] – all parties follow the protocol description, but may keep a record of the interaction and try to infer additional information about the other party's input. Protocols secure in the semi-honest model provably prevent any such inference which includes many real-world attacks, such as insider attacks.

Through the use of the type system the compiler statically verifies the security of the protocol during compilation. Only secure protocols are compiled. The programmer is immediately notified about any (possible) security violation.

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

We evaluate our security type system for the L1 language by implementing two examples. The first one is a simple multiplication protocol based on [14]. The second one is a complex protocol for privacy-preserving string processing from [21]. This example shows that even such protocols – using a wide variety of protocol primitives in intricate ways – can be effectively implemented in our type system. Furthermore, the security proof of our type system also extends to this new implementation. This underpins the protocol's manual security proof from [21] by formal verification.

In summary, this paper contributes

- a *security type system* for mixed protocol secure computation.
- an *integration* of this type system into the L1 language.
- a *proof* that any well-typed program is secure in the semi-honest model.
- an *evaluation* using two exemplary protocols for multiplication and substring creation. This shows that implementing any functionality and even complex protocols using our type system is feasible.

The remainder of this paper is structured as follows. In Section 2 we review the related work in secure computation and security type systems. Section 3 reviews the protocol primitives implemented for secure computation and Section 4 reviews the L1 language. Sections 5 and 6 present our main results. Section 5 describes how re-randomization is implemented and Section 6 describes the security type system. We show our examples in Section 7 and conclude the paper in Section 8.

2. RELATED WORK

Our work is related to domain-specific languages (and extensions) for secure computation [2, 4, 9, 18, 19, 28, 32, 35] and type systems for information flow security [12, 29, 34, 36].

Domain-specific languages (DSL) for secure computations can be coarsely classified into those based on a specific secure computation protocol, such as [2, 4, 18, 28], and those that extend a generic programming language [9, 19, 35]. DSLs based on specific secure computation protocols implement functionality solely based on those protocols. This may not limit implementable functionality, since these protocols are generic and support any functionality, but it may limit performance, since problem-specific protocol primitives are usually expected to yield better efficiency. Nevertheless, once the specific protocols is proven secure, any implemented functionality is also secure.

The first such domain-specific language for secure computation was FairPlay [28]. It implemented Yao's two-party, garbled circuit protocol [37]. It was later extended to multi-party computations in [2] based on the multi-party version of Yao's protocol [1]. It introduced the SFDL programming language which provided an abstraction of the ideal functionality, i.e. the function computed by the secure protocol. Programs in SFDL are translated into a binary circuit which is interpreted using Yao's protocol. Secure multi-party computations were also implemented by ShareMind [4] based on the information-theoretically secure protocol of [3]. Its programming language is rather simplistic and resembles machine language. This makes programming harder, but also makes it easier to ensure security. All these single primitive DSLs generally suffer from poor performance of the compiled protocols. Mixing primitives in a DSL, such as Yao's protocol and homomorphic encryption, was introduced in TASTY [18]. Nevertheless, they restricted themselves to one provably secure protocol of [25] and enabled annotations in the language naming the primitive to use.

DSLs as extensions of existing, generic programming languages can implement any protocol based on mixing (almost) arbitrary primitives. This allows implementing the most efficient protocols, but also does not prevent the programmer from making mistakes and implementing insecure protocols. The first such language is VIFF [9] based on the Python language. Although it is catered for the multi-party protocol using linear secret sharing of [8] which is also considered as the basis for the related SCML programming language of [32], in its implementation it supports the full power of Python. Similar extensions have been presented for the Java language [19, 35]. The framework of [19] is based on Yao's protocol [37] and currently produces the most efficient protocols, but provides very little language support. A significant portion of its efficiency gain is based on optimizations by the programmer, which also invites mistakes likely at the expense of security. Implementing some such optimizations in the compiler was proposed in [22].

Our security type system is based on the L1 language [35] for mixed protocol secure computation, although it can be adapted for other mixed-protocol languages. This language also currently provides little support to the programmer for protocol security.

Our idea of supporting security in a programming language by a type system has been applied to information flow security. Information flow is concerned with confidentiality breaches between principals in computer systems [26]. Its lattice policy model of security levels has been introduced in [10]. Later, it was shown that compliance with such a policy can be statically verified given the program source code [11]. Volpano et al. were the first to recognize that these policies can also be implemented using a type system [36]. Their type system also influenced our construction of our secure computation security type system.

Since type systems can be effectively handled by programmers as well as compilers, this sparked recent interest in language-based information flow security. A very good survey of current research can be found in [34]. There is also an implementation of a type system for information flow security in Java called JIF [29].

Recently information flow types have been applied to homomorphic encryption [12]. These do not yet cover secure computations using homomorphic encryption, since all secure computations involve admissible information flows. Therefore the basic typing assumption of non-interference does not hold in secure computation. Our security type system must therefore cater for a more complicated use case.

3. SECURE COMPUTATION

We implement secure computations using the primitives of homomorphic encryption, secret sharing, garbled circuits and oblivious transfer. These primitives can be combined in many ways and even potentially result in an insecure protocol. For simplicity we only consider secure two-party computation.

3.1 Homomorphic Encryption

Secure computation can be implemented based on additively homomorphic encryption. On the one hand, opposed to fully homomorphic encryption [13] additively homomorphic encryption only implements addition (modulo a key-dependent constant) as the homomorphic operation. On the other hand, additively homomorphic encryption is as fast as standard public-key cryptography.

Let $E_X(x)$ denote the encryption of plaintext x encrypted under X's (Alice's or Bob's) public key and $D_X(c)$ the corresponding decryption of ciphertext c. Then the homomorphism can be expressed as

$$D_X(E_X(x) \cdot E_X(y)) = x + y$$

The following property can be easily derived

$$D_X(E_X(x)^y) = xy$$

In our implementation we mainly use Paillier's encryption system [33], although our language also supports different encryption systems such as Naccache-Stern [31] and allows the programmer to extend the available encryption systems. Paillier's and Naccache-Stern's encryption systems are not only public-key, but also secure against chosen plaintext attacks (IND-CPA). This semantic security implies that two different ciphertexts – even of the same plaintext – cannot be distinguished (without the private key).

In secure computation based on homomorphic encryption it is usually sufficient to use one key per party. Our type system - differently than [12] – therefore does not protect against mixing ciphertexts from different keys.

3.2 Secret Sharing

In order to implement the full functionality of secure computation we augment homomorphic encryption with secret sharing. Each variable is secretly shared between Alice and Bob. Let x be the variable secretly shared and p be the modulus of the homomorphic encryption. Then Alice has x_A and Bob has x_B, such that

$$x = x_A + x_B \bmod p$$

In the following we show how secret sharing and homomorphic encryption can be used to implement any functionality in secure computation. Many other combinations of homomorphic encryption and secret sharing may already be insecure, e.g. by revealing dependent shares. In order to implement any ideal functionality it suffices to implement addition and multiplication. Addition of $x = x_A + x_B$ and $y = y_A + y_B$ (of the same bit-length l) can be implemented locally by addition of each party's shares. Multiplication $z = x \cdot y$ needs to be implemented as a protocol. Let r be a uniformly random number in $\mathbb{Z}_p$. We use the commonly used protocol in Figure 1 from [14].

$A \longrightarrow B$	$E_A(x_A), E_A(y_A)$
$B \longrightarrow A$	$E_A(c) = E_A(x_A)^{y_B} E_A(y_A)^{x_B} E_A(r)$
A	$z_A = x_A y_A + c \bmod p$
B	$z_B = x_B y_B - r \bmod p$

Figure 1: Multiplication Protocol

It is easy to verify that $z_A + z_B = (x_A + x_B)(y_A + y_B)$.

3.3 Oblivious Transfer

Oblivious transfer (OT) is a protocol between a sender and a receiver. As input the sender has n messages $x_0, \ldots, x_{n-1}$ and the receiver has an index i ($0 \leq i < n$). After the protocol the receiver obtains x_i as output. A secure OT protocol ensures that the sender has not learnt i and the receiver has learnt nothing about the other messages x_j ($j \neq i$).

OT can be used as a primitive in mixed secure computations. It facilitates a distributed "if" statement, where one party has the results of the branches and the other party the condition. Furthermore, although we do not implement this protocol in our language, OT can also be used as the sole primitive to implement generic secure computation protocols [15].

There are a number of efficient implementations of OT including [30]. Nevertheless, all OT involve some public key operations. In [20] the authors describe a technique to extend a constant number of OTs to polynomially many OTs using symmetric cryptography.

3.4 Garbled Circuits

Yao introduced secure two-party computation in [37]. He uses a technique called garbled circuits. Garbled circuits allow the computation of any function, but in our language we combine garbled circuits with secret shares and somewhat restrict its functionality. Our restriction is that each input and output (function result) must be a secret share. Let Alice's input be a and Bob's input b which both may be a secret share. Let f be the function to be computed using garbled circuits. We then automatically augment the circuit using an additional input r by the circuit encryptor. The circuit then computes the function f':

$$f'(a, b, r) = f(a, b) - r$$

The circuit encryptor obtains no output from the secure computation, but uses its locally chosen r as return value. The circuit decryptor obtains the new function result f'. These two return values are secret shares of the function result $f(a, b)$ which can be subsequently used in computations using homomorphic encryptions or garbled circuits.

Secrets as inputs are automatically reconstructed at the beginning of the function. For example, let x_A, y_A and x_B, y_B be shares of x and y, respectively. Then the function $f(x, y) = x < y$ to compare x and y is implemented as follows:

$$f''(x_A, y_A, x_B, y_B, r) = ((x_A + x_B) < (y_A + y_B)) - r$$

4. THE L1 LANGUAGE

We extend the L1 language [35] for mixed-protocol secure computation with our security type system. L1 is a simplification and extension of Java that incorporates features to better support secure computation. The two L1 features important for this paper are messaging support and player-specific code.

Messaging support enables the parties to send each other messages. It is implemented using a framework for asynchronous communication available to all parties. In its basic form the **send** command is invoked as in Listing 1. Here, we send the contents of the variable *_var* to the player with identifier 2 under the message name *var_id*. The variable *_var* is cleared after the **send** command, such that it cannot be reused. This ensures that each variable is sent at most once (which is also always sufficient).

```
1 send(2, _var, "var_id");
```

Listing 1: Send Command

Player identifiers are assigned globally in a configuration file at the start of the protocol execution. An identifier is associated with an IP address and port number where the player's code receives messages. Message names must be unique for a receiving party. Otherwise, messages may be lost due to overlapping communication.

The other party can receive a previously sent value using the **read** command. We show in Listing 2 an example where the player receives content under the message name *var_id*.

```
1 var = read("var_id");
```

Listing 2: Receive Command

A particular design principle of the L1 language is that the code for all players is integrated into one program. This one (and the same) program is then executed by all players. This corresponds to secure computation where the functionality (and the protocol) is known to all players in advance. Now in order to implement the sending and receiving of messages between the parties – using the **send** and **read** commands – we use player-specific code.

Player-specific code is only executed at a player with a specific identifier. This player's identifier is placed at the start of the statement. If no such identifier is present, the statement is executed by all parties. We can therefore implement the send and receive commands within the same program. The following Listing 3 combines the code of the previous two listings.

```
1 1: send(2, _var, "var_id");
2 2: var = read("var_id");
```

Listing 3: Message Sending and Receiving

4.1 Data Types

The L1 language has its own type system for basic, non-security relevant types. This type system is orthogonal to our security type system. L1 provides data types for multi-precision integers (**int**), public (**pubk**) and private (**privk**) keys. We extend this for the purpose of our type system with data types for secret shares and ciphertexts.

L1 implements copy semantics for variable assignments. Even if a complex variable (such as a share, ciphertext or a multi-precision integer) is assigned, then the assigned value is copied onto the memory of the assigned variable. This differs from the pointer semantics of Java where only a reference is assigned.

A secret share type is parameterized by the modulus of the share. We create only additive shares, i.e., let p be the modulus and s ($s < p$) be the secret. Then there are n ($n \geq 2$) shares s_i, such that

$$\sum_{i=0}^{n-1} s_i (\text{mod } p) = s$$

A secret share can be internally represented as an object containing a multi-precision integer – the value of the secret share – and a reference to a multi-precision integer – the modulus. An integer can be implicitly converted into a secret share, but converting a secret share to an integer requires a type cast. When an expression is assigned to a secret share, a modulo operation is performed automatically. We declare a secret share at each party as in Listing 4.

```
1 int p = getModulus(pubKey);
2 share<p> s;
```

Listing 4: Secret Share Declaration

There are several ciphertext data types. It not only depends on the encryption system used, but also for each public key encryption system there are two types of ciphertexts: one where the player has the private key and one where the player has only the public key. Our assumption is that for ciphertexts where the player has the private key no other player can access the plaintext, i.e. the private key is indeed confidential. Similar to secret shares we parameterize the ciphertext type with the respective (public or private) key.

A ciphertext contains by definition a secret share as payload. This is not explicitly declared in our language extension. Furthermore, this secret share has the same modulus as the homomorphic operation in the encryption system using the specified key. A ciphertext can be internally represented as an object containing multi-precision integers – the ciphertext value – and a reference to a key. Again, an integer can be implicitly converted into a ciphertext, but converting a ciphertext to an integer requires a type cast. We declare ciphertexts as in Listing 5.

```
1 pubk pubKey;
2 cipher<pubKey> c;
3
4 prvk privKey;
5 cipher<privKey> d;
```

Listing 5: Ciphertext Declaration

4.2 Cryptographic Protocols

As cryptographic protocol primitives we need to support garbled circuits and oblivious transfer. We implement garbled circuits using two commands. Yao's protocol has a sender who encrypts (and garbles) the circuit and a receiver who – after obtaining the decryption keys via oblivious transfer – decrypts the circuit. The ideal functionality to be computed in the circuit is specified as a function (in C like syntax). Each input variable is shared between both parties and reconstructed at the beginning of the circuit. Each party then inputs its shares as parameters to the command. The output is also automatically shared. The random variable r (using the modulus of the result) is automatically appended as input by the encryptor to the circuit. It is filled with a random input chosen by the command and returned as a result from the command. The output (the result of the circuit) is returned as a function result to the decryptor (subtracted by the random value of the encryptor).

In the ideal functionality we currently only support arithmetic operations (addition, subtraction, multiplication, division) and comparison (equal, lower, lower or equal). In Listing 6 we implement the circuit for the above function $f''(x_A, y_A, x_B, y_B, r) = ((x_A + x_B) < (y_A + y_B)) - r$.

Oblivious transfer is also implemented via two commands: one for sending and one for receiving. The sending command takes an array as parameter. Its length is the number

```
1  share<(1 << 32)> x, y;
2  share<2> r;
3
4  1: r = gcencrypt(2,
5           "bool compare(int x, int y) {
6                   compare = x < y;
7           }",
8           x, y);
9  2: r = gcdecrypt(1, "compare", x, y);
```

Listing 6: Garbled Circuit Protocol

of possible choices in the oblivious transfer. The input array is cleared after sending. The receiving command takes the index as a parameter and outputs the selected message. We need to specify the communicating party in the receive command as well, since it is an interactive protocol. In Listing 7 we show an example for an 1-out-of-2 oblivious transfer.[1]

```
1  1: share<p> _x*[2];
2  2: share<p> y;
3  2: int b;
4
5  1: otsend(2, "id", _x);
6  2: y = otread(1, "id", b);
```

Listing 7: Oblivious Transfer

5. REVERSE TAINT CHECKING

Our type system ensure that only freshly and independently chosen random variables are revealed to the other party. We distinguish these variables by typing them as *untainted* (versus tainted for variables during processing). Each basic data type of secret share, ciphertext or array can be tainted or untainted. We perform the operation of re-randomization for reverse taint checking in order to create the untainted type of variables.

Taint checking[2] is a programming language operation that allows to track whether inputs are processed safely. It helps to prevent certain common programming errors, such as SQL injections or buffer overflows. In taint checking mode each variable set by an input is tainted. Then after certain checks have been performed, e.g. in PERL using regular expressions, the variable can be cleared as untainted. Every time a tainted variable is used in a dangerous operation, such as accessing a database, the program is aborted with a fatal error.

We call our operation reverse taint checking, because on the one hand we use a similar variable tracking mechanism (although statically verifiable), but on the other hand the taint flag is set by local operations and checked on sending data. In detail our algorithm works as follows. Each variable is either tainted or untainted. All variables assigned by expressions are tainted, since they are result of a local computation. It does not matter whether the assignment expression is a **read** command, some arithmetic operation or even a constant. Every regularly assigned variable is tainted.

The fundamental check is that the sending commands do not accept tainted variables as parameters. If the parameter is tainted, then the program does not type check. Only untainted variables may be sent to the other parties.

[1]The star in line 1 is explained after introducing our security type system.

[2]http://perldoc.perl.org/perlsec.html#Taint-mode

We define the semantics of an untainted variable as we use it in our security type system: An untainted variable is independently *random*, *immutable*, and *use-once*. Independently random implies that it contains indepedently randomly distributed data. It is immutable, as it can only be set by special commands and the randomness is chosen by the system. An untainted variable can only be used once and is cleared after use in a send or re-randomization command. We summarize these properties in Definition 1. Next, we need to describe how to create an untainted variable.

DEFINITION 1. *An untainted variable is an immutable object that contains random data independent of all other* untainted *variables. It can be used in send and re-randomization command and is cleared after use.*

5.1 Re-Randomization

We perform a type-dependent re-randomization operation in order to create untainted variables. Re-randomization renews the randomness in a tainted variable using fresh random input. Depending on the type we perform different re-randomization operations. A re-randomize operation is written as in Listing 8.

```
1  share<p> s;
2  cipher<pubKey> c;
3  cipher<privKey> d;
4  share<p> x[2];
5  share<p> r;
6
7  _s = rerandomize(s, r);
8  _c = rerandomize(c, r);
9  _d = rerandomize(d, null);
10 _x = rerandomize(x, r);
```

Listing 8: Re-Randomization

Our different re-randomization operations are as follows:

- *Secret share*: A secret share is associated with a modulus p. We choose a fresh random number r uniformly in $\mathbb{Z}_p$. Let s be the value of the secret share. We return the updated

$$\_s \leftarrow s - r \pmod{p}$$

  and copy unto the randomization parameter (given to the function) the new secret share with the value r.

- *Ciphertext with public key only*: The ciphertext and the contained secret share are associated with the same modulus p. We again choose a fresh random number r uniformly in $\mathbb{Z}_p$. We encrypt the negation of this random number r using the associated public key. Let $E(-r)$ denote this encryption and c denote the ciphertext of the re-randomized variable. We return the following, updated ciphertext

$$\_c \leftarrow c \cdot E(-r)$$

  and copy unto the randomization parameter the new secret share with the value r.

- *Ciphertext with private key*: We assume that the possessor of the private key is its sole possessor. Therefore the contained secret share cannot be accessed by any other party and there is no need to re-randomize it. Nevertheless, we need to re-randomize the ciphertext. We use the usual re-randomization of IND-CPA

secure, homomorphic encryption. Let d denote this ciphertext. We return it as

$$\_d \leftarrow d \cdot E(0)$$

The randomization parameter is not used.

- *Array*: For arrays we choose the randomization parameter (i.e. r, $E(r)$ or $E(0)$) only once. The same randomization is applied to each element of the array and the randomization parameter is copied as usual. The return array contains the re-randomized elements. Note that only the array is untainted by this operation. Each element in the array remains tainted. This enables using the array in an oblivious transfer command, but prevents the elements from being sent individually using the send command.

5.2 Re-Randomization with Untainted Variables

Sometimes it is necessary to choose random shares before performing an operation and sending the result. An instance of this requirement occurs in our second example of substring creation in Section 7.2. Usually an untainted variable is created during a re-randomization command and then sent over the network, but instead of directly sending it, the untainted variable can be used to re-randomize another variable. The semantics of this re-randomization operation is that the untainted input parameter is cleared after the re-randomization command (just as it is cleared after a send command), such that it cannot be reused. Yet, the return variable is then subsequently untainted and can be used as such. Listing 9 shows an example where an untainted variable is used to re-randomize another variable. The variable $\_s$ is null after the execution of this code.

The complete re-randomization function is depicted in Algorithm 1. It is easy to verify from Algorithm 1 that after re-randomization one variable – the untainted one – is independently randomly distributed as required by Definition 1. This holds even in case of re-randomization with untainted variables.

```
1 share<p> r;
2
3 // _s = s - r
4 // r is set and chosen fresh
5 _s = rerandomize(s, r);
6 // _t = t - _s
7 // _s is used and then cleared
8 _t = rerandomize(t, _s);
```

Listing 9: Re-Randomization with Untainted Variable

5.3 Output

If all variables were only to be sent as re-randomized variables, no useful result could ever be obtained, i.e. every (useful) secure computation has an admissible information flow. Therefore we need to allow reconstruction of the output values from the secret shares. Let s be a secret share designated as output. We use an output statement in L1 in order to declare output, but before we can output the result the parties need to exchange result secret shares. The output command is restricted to secret shares as parameter. They can only send the secret share if the variable is untainted. The programmer needs to declare that this variable is designated for output before the output is actually performed. We perform a mutual exchange of (result) shares between two parties as in Listing 10.

Algorithm 1 Rerandomization Algorithm

```
function RERANDOMIZEUNIT(v, r)
    if INSTANCEOF(v, SecretShare) then
        return v − r
    end if
    if INSTANCEOF(v, PublicKeyCipherText) then
        return v· ENCRYPT(−r)
    end if
    if INSTANCEOF(v, PrivateKeyCipherText) then
        return v· ENCRYPT(0)
    end if
end function
function RERANDOMIZE(v, r)
    if ISTAINTED(r) then
        if INSTANCEOF(v, Array) then
            m ← GETMODULUS(v_1)
            r ← RANDOM(m)
            for all v_i ∈ v do
                v_i ← RERANDOMIZEUNIT(v_i, r)
            end for
            return v
        else
            m ← GETMODULUS(v)
            r ← RANDOM(m)
            return RERANDOMIZEUNIT(v, r)
        end if
    else
        if INSTANCEOF(v, Array) then
            for all v_i ∈ v do
                v_i ← RERANDOMIZEUNIT(v_i, r)
            end for
            r ← null
            return v
        else
            v' ← RERANDOMIZEUNIT(v, r)
            r ← null
            return v'
        end if
    end if
end function
```

```
1 share<p> s_prime;
2
3 _s = output(s);
4 send(id() % 2 + 1, _s, "result_share");
5 s_prime = read("result_share");
6 s = s + s_prime;
7 output("result = ", s);
```

Listing 10: Share Reconstruction and Output

6. SECURITY TYPE SYSTEM

In our L1 language we augment the regular type system with a security type system where every expression carries both a type (such as secret share, ciphertext or array) and a security type (tainted or untainted). Following the definition of [34] our security type system is a collection of typing rules that describe what security type is assigned to a program (or expression), based on the types of sub-programs (sub-expressions). We write $\vdash exp : t$ to mean that the expression exp has security type $t \in \{tainted, untainted\}$ according to the typing rules. This assertion is known as a typing judgment. Similar to the construction of the security

type system for information flow [36] we use a security context $[sc]$ associated with a label of the program counter. We use the security context to prevent the programmer from sending messages depending on the truth value of expressions with tainted variables. Untainted variables are random and therefore not useful for branching conditions. This limitation is important, since not only the content of a message may reveal information, but just sending the message itself. The typing judgment $[sc] \vdash C$ means that the program C is typable in the security context sc.

Figure 2 presents the typing rules for a simplification of the L1 language. We omit loops, since they can only have a constant number of iterations in secure computation and can therefore be unrolled. We also omit the typing rules for the regular type system. Expression security types and security contexts can be either tainted or untainted. A typing rule is an inference rule that describes how types are assigned. The statements above the line must be fulfilled for the rule to be applied, yielding the statement below the line.

According to the rules [E1-3], some variables (u) have security type untainted. All other expressions and variables have type tainted. This includes expression which include (but are not limited to) occurrences of untainted variables. Rules [R1-2] ensure that untainted variables can only be assigned by re-randomization and output statements, but are otherwise static.

Consider the rules [M1-6]. The message sending commands can only be executed in an untainted security context. Of those commands the ones that transmit plaintext messages as payload (**send**, **otsend**) can only transmit untainted variables. Received variables are always immediately tainted.

The typing rules [C1-5] control the security context in a composite program. The commands **skip** and variable assignment are typable in any context. Branches must be typable in a tainted context. This is justified by the above requirement that message sending commands may not be executed depending on tainted variables. The rules [C4-5] enable composing programs similar to [36] including the subsumption rule. It enables sending messages before or after a tainted context (branching).

In L1 we declare untainted variables using an asterisk after the type declaration. We show examples for a share and an untainted array in Listing 11 (and already in Listing 7).

6.1 Proofs

Goldreich [15] defines security in the semi-honest model. The view $VIEW^{\Pi}(x, y)$ of a party during protocol Π on this party's input x and the other party's input y is its input x, the outcome of its coin tosses and the messages received during the execution of the protocol.

DEFINITION 2. *We say a protocol Π computing $f(x, y)$ is secure in the* semi-honest *model, if for each party there exist a polynomial-time simulator S given the party's input and output that is computationally indistinguishable from the party's view $VIEW^{\Pi}(x, y)$:*

$$S(x, f(x, y)) \stackrel{c}{=} VIEW^{\Pi}(x, y)$$

Our main theorem states that well-typed program is compiled into a protocol secure in the semi-honest model.

$$\vdash u : untainted \quad (E-1)$$

$$\frac{var \neq u}{\vdash var : tainted} \quad (E-2)$$

$$\frac{exp \neq u}{\vdash exp : tainted} \quad (E-3)$$

$$\frac{\vdash var_1 : untainted}{[sc] \vdash var_1 = \mathbf{rerandomize}(exp, var_2)} \quad (R-1)$$

$$\frac{\vdash var : untainted}{[sc] \vdash var = \mathbf{output}(exp)} \quad (R-2)$$

$$\frac{\vdash var : untainted}{[untainted] \vdash \mathbf{send}(var)} \quad (M-1)$$

$$\frac{\vdash var : tainted}{[untainted] \vdash var = \mathbf{read}()} \quad (M-2)$$

$$\frac{\vdash var : untainted}{[untainted] \vdash \mathbf{otsend}(var)} \quad (M-3)$$

$$\frac{\vdash var : tainted}{[untainted] \vdash var = \mathbf{otread}(exp)} \quad (M-4)$$

$$\frac{\vdash var : tainted}{[untainted] \vdash var = \mathbf{gcencrypt}(exp, \dots)} \quad (M-5)$$

$$\frac{\vdash var : tainted}{[untainted] \vdash var = \mathbf{gcdecrypt}(exp, \dots)} \quad (M-6)$$

$$[sc] \vdash \mathbf{skip} \quad (C-1)$$

$$\frac{\vdash var : tainted}{[sc] \vdash var = exp} \quad (C-2)$$

$$\frac{[tainted] \vdash C_1 \quad [tainted] \vdash C_2}{[sc] \vdash \mathbf{if}\ exp\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2} \quad (C-3)$$

$$\frac{[sc] \vdash C_1 \quad [sc] \vdash C_2}{[sc] \vdash C_1; C_2} \quad (C-4)$$

$$\frac{[tainted] \vdash C}{[untainted] \vdash C} \quad (C-5)$$

Figure 2: Security Typing Rules

```
1 share<p>* _s;
2 share<p> _a*[2];
```

Listing 11: Untainted Variable Declaration

THEOREM 3. *Let program C implement function $\mathcal{F}$. Then if C is well-typed ($\vdash C$), it implements $\mathcal{F}$ securely in the semi-honest model.*

Proof Outline: We need to show that we can construct a simulator for each party's view. The problem is that the view is determined by the messages received and not sent (where type safety applies). We therefore assume that the entire protocol is written in L1 using player-specific code. Then we simulate the messages received from the other party. We give only the construction for one party's simulator – due to the symmetry of the language it also applies to the other party.

Furthermore, differently from information flow type system there is always an admissible information flow in secure computation. This is due to the nature of the joint computation, i.e. some result is revealed. It is therefore not possible to strictly distinguish between the program and the

information conveyed. We capture this using the output variables.

PROOF. We now construct the simulator for one party. The simulator consists of input, coin tosses and messages received. Input can be simulated by the real input (which is given to the simulator) and coin tosses are chosen by the Java random number generator. This leaves the received messages to be simulated.

Let $m_1, \ldots, m_n$ be the messages received. We then construct the simulator step-by-step, i.e. for each message received we add a simulated message. This means, we start with the empty simulator S_0 which does nothing. Then we construct S_i from S_{i-1} by appending a simulation of message m_i to S_{i-1}. Therefore it is first necessary to note that the number of messages is constant.

LEMMA 4. *In a well-typed program ($\vdash C$) the number of sending statements (**send**, **otsend**, **gcencrypt**) is constant.*

Note that no sending statement may appear inside a branching statement. We can complete the proof by induction over the structure of C.

We now construct a simulated message for each message m_i received. We refer to Goldreich's composition theorem [15] for our cryptographic protocols. Oblivious transfer and Yao's protocol can be simulated as an oracle transmitting only the resulting messages, as long as the implementations have been proven secure in the semi-honest model. The security proof for our implementation of oblivious transfer can be found in [30]. The security proof for generic Yao's protocol can be found in [27]. We can therefore simulate only the resulting messages.

There are two types of messages received: output and intermediate.

Output messages: These are messages marked as output by the other party. Let m_i be such a message received. In a correctly implemented program these messages are indeed output at the local party (after combination with a local share). Therefore the local party can simulate the message as follows: Let o be the output which is given to the simulator and l the local share for reconstructing this value. The local share l can be computed from the local state of the party up to that point in the program. This must be the case, since otherwise the party could not produce the output. Then the message is simulated as

$$m_i = o - l \pmod{p}$$

Intermediate messages: These are messages re-randomized by the other party. Let m_i be such a message received. In a well-typed program these message are all randomly independently distributed.

LEMMA 5. *In a well-typed program all re-randomized variables are independently randomly distributed.*

This follows directly from our construction of re-randomization. After the algorithm of re-randomization (see Algorithm 1) only one untainted variable is assigned. This variable is always independently randomly distributed. Therefore the set of re-randomized variables is always a set of independently randomly distributed variables. Note that local and output variables may be dependent on the re-randomized variables.

The received message m is then simulated as an independent random variable depending on the type of the message (secret share or public-key ciphertext)

$$m_i \leftarrow_R \mathbb{Z}_p / E(\mathbb{Z}_p)$$

This completes the simulator – all messages can be simulated – and consequently the proof of semi-honest security. □

7. EXAMPLES

7.1 Multiplication

We first show a simple example. We implement the multiplication protocol using homomorphic encryption and secret shares from Figure 1 based on [14]. This protocol shows that using our typed language any functionality can be implemented securely (Listing 12).

The encrypt command (lines 33-34) returns a tainted variable. Subsequently (lines 35-36) we re-randomize the ciphertext (with known private key). This may seem superfluous, since the randomness is fresh during encryption, but the contained share may be tainted in ciphertexts with public key only. We chose this re-randomization approach over a polymorphic command based on return type. We see in line 47 that using the **rerandomize** command we can omit explicitly adding a random variable. Instead we use the randomness introduced by the re-randomization command.

7.2 Substring Creation

As a second example we show the substring creation protocol of Jensen and Kerschbaum [21]. This protocol is one of a set of three for privacy-preserving string processing. It is particularly suited to show the power of our language and type system. First, it uses all of our primitives: Yao's garbled circuits, oblivious transfer, secret shares and homomorphic encryption. Second, it's security proof is not obvious, such that a well-typed implementation underpins its provable security.

The protocol operates on strings. A string consists of a length field and a fixed-length array of characters. Each string is secretly shared between Alice and Bob, such that neither Alice nor Bob know its contents or even length.

The purpose of the protocol is to create a substring of length t of a string x starting at position t. The resulting substring will also be secretly shared – just as the inputs, i.e. there is no output reconstruction. The protocol starts with a rotation of the shared string to the left by s characters. Then the trailing characters beyond the length field are masked, such that they are all set to 0 again. Each share of the string is masked separately, such that the masking protocol is executed twice with the roles reversed. The composed main protocol is shown in Listing 13.

The types for strings (containing length field and array) are defined in a separate header (not shown for brevity) This header of the protocol also loads the cryptographic keys for the homomorphic encryption. Both a private, public keypair and a public key (of the other party) are read from disk. In this protocol we use Naccache-Stern encryption [31], because the modulus of the homomorphic operation must be the same at both parties. This cannot be achieved using Paillier's encryption system [33]. We use a fixed modulus of 256 in order to simplify integration with Yao's protocol. In [21] the authors use Goldwasser-Micali encryption [17],

```
1  int multiply (int s, int t) {
2     // key variables
3     1: prvk privKey;
4     pubk pubKey;
5
6     // read keys from hard disk
7     1: {
8        privKey = readKey("priv.key");
9        pubKey = getPublicKey(privKey);
10    }
11    2: pubKey = readKey("pub.key");
12
13    int n = getModulus(pubKey);
14    share<n> x, y, z;
15
16    // initialize shares
17    x = s;
18    y = t;
19
20    // declare ciphertexts
21    1: {
22       cipher<privKey> e, f, g;
23       cipher<privKey>* _e, _f;
24    }
25    2: {
26       cipher<pubKey> e, f, g;
27       cipher<pubKey>* _g;
28       share<n> r;
29    }
30
31    // send 1st message
32    1: {
33       e = encrypt(privKey, x);
34       f = encrypt(privKey, y);
35       _e = rerandomize(e, null);
36       _f = rerandomize(f, null);
37       send(2, _e, "E_A(x_A)");
38       send(2, _f, "E_A(y_A)");
39    }
40    // receive 1st message, send 2nd
41    2: {
42       e = read("E_A(x_A)");
43       f = read("E_A(y_A)");
44
45       g = modPow(e, y, n);
46       g *= modPow(f, x, n);
47       _g = rerandomize(g, r);
48       send(1, _g, "E_A(c)");
49    }
50    // receive 2nd message, compute result
51    1: {
52       g = read("E_A(c)");
53
54       z = decrypt(g) + x * y;
55    }
56    // also compute result at Bob
57    2: z = r + x * y;
58
59    return (int) z;
60 }
```

Listing 12: Secure Multiplication

because they need its properties in a subsequent protocol. In this paper we do not implement this subsequent protocol and can therefore use the more efficient Naccache-Stern encryption system. There are also some helper functions for adding (secret shares), rotating strings, re-randomization functions for the different string types, en-/decryption functions and message sending functions – all not shown for brevity.

Next we consider the sub-protocols of the substring protocol. The complete rotation protocol is presented in Listing 14. The rotation protocol rotates the string, such that the initial character is at position 0. This is achieved us-

```
1  string substring (string x,
2                    share<n> s,
3                    share<n> t) {
4     string a, b, sub;
5
6     sub = rotate(x, s);
7     sub.len = t;
8     1: a = mask_send(sub);
9     2: a = mask_recv(sub);
10    1: b = mask_recv(sub);
11    2: b = mask_send(sub);
12
13    return add(a, b);
14 }
```

Listing 13: Substring: Composed Protocol

```
1  string rotate (share<n> s, string in) {
2     string x, r;
3     _string _x;
4     string_priv e;
5     _string_priv _e;
6     string_pub y;
7     _string_pub _y;
8
9     x = in;
10    1:{
11       _x = rerandomize_shift(x, s);
12       _e = encrypt_string(x);
13       send_string("1", _e);
14    }
15    2:{
16       y = recv_string_pub("1");
17       rotate_left(y, s);
18       _y = rerandomize_string(y, r);
19       rotate_right(r, s);
20       x = add(x, r);
21       send_string("2", _y);
22       _e = encrypt_string(x);
23       send_string("3", _e);
24    }
25    1:{
26       e = recv_string_priv("2");
27       x = decrypt_string(e);
28       rotate_left(x, s);
29       y = recv_string_pub("3");
30       rotate_left(y, s);
31       _y = rerandomize_string(y, _x);
32       send_string("4", _y);
33    }
34    2:{
35       e = recv_string_priv("4");
36       x = decrypt_string(e);
37       rotate_left(x, s);
38    }
39
40    return x;
41 }
```

Listing 14: Substring: Rotation

```
1  _string rerandomize_shift (string x,
2                              share<n> s) {
3     _string _x;
4     string r;
5
6     x = rotate_right(x, s);
7     _x = rerandomize_string(x, r);
8     x = r;
9     x = rotate_left(x, s);
10    return _x;
11 }
```

Listing 15: Substring: Initial Re-randomization

ing the following technique: Each share is encrypted and sent to the other parties which then rotates it by its share of the initial position. It then re-randomizes it and returns it before it is rotated by the local share of the initial position. This requires a particular interlocking technique. Before Alice can send her share to Bob she needs to re-randomize it with the re-randomization values for Bob's share (these will cancel out). She cannot do this after she receives the returned ciphertexts from Bob, since they are then already rotated. Therefore, in order to implement this protocol Alice needs to choose the re-randomization parameters for Bob's share before she sends her share. This is achieved by re-randomization using untainted variables created in the **rerandomize_shift** function (Listing 15). This function creates an untainted variable containing a random share that is already rotated by the local share of the initial position.

We have implemented the entire protocol in one function where each party's code is implemented using player-specific code. We chose this type of implementation, since the two sides of the protocol are quite asymmetric. In line 11 (Listing 14) there is the call of the initial re-randomization creating the untainted variable. This untainted variable is then used for re-randomization in line 31 (Listing 14) completing the interlock technique. Bob can simply add his re-randomization to his share in line 20 (Listing 14).

```
 1 string mask_send (string x) {
 2    share<2> b;
 3    string m, y, r;
 4    _string _y;
 5    string_priv e;
 6    _string_priv _e;
 7
 8    m.len = n;
 9    for (int i = 0; i < n; i++)
10       if (i < x.len)
11          m.chars[i] = 1;
12       else
13          m.chars[i] = 0;
14    _e = encrypt_string(m);
15    send_string("5", _e);
16    b = gcdecrypt("compare",
17                  (int) x.len, 0);
18    e = otread("6", (int) b);
19    y = decrypt_string(e);
20    _y = rerandomize_string(y, r);
21    send_string("7", _y);
22    return r;
23 }
```

Listing 16: Substring: Masking Sender

The masking protocol creates a 0, 1 encrypted string which is used to mask the local share. Care has to be taken, if the shares of the length of the substring wrap around the modulus. Two cases need to be prepared: one in which they do and one in which they do not wrap around. The correct one is chosen by oblivious transfer. The condition is computed by a comparison implemented as a Yao's protocol. Therefore we see also these primitives in this protocol.

The masking protocol is executed twice with roles reversed. Therefore we implemented each side of the protocol as a separate function. Listing 16 shows one side and Listing 17 the other. Then each side is called with the local input in the composed protocol (Listing 13 lines 6–9).

```
 1 string mask_recv (string x) {
 2    share<2> b;
 3    string r, q;
 4    string_pub y, e, f[2];
 5    string_pub _f*[2];
 6
 7    y = recv_string("5");
 8    f[0].len = x.len;
 9    f[1].len = x.len;
10    for (int i = 0; i < x.len; i++)
11       f[0].chars[i] =
12          encrypt(pubKeyExt, 1);
13    for (int i = x.len; i < n; i++)
14       f[0].chars[i] =
15          y.chars[i - x.len];
16    for (int i = 0; i < n - x.len; i++)
17       f[1].chars[i] =
18          y.chars[i + n - x.len];
19    for (int i = n - x.len;
20          i < n - x.len; i++)
21       f[1].chars[i] =
22          encrypt(pubKeyExt, 0);
23    for (int i = 0; i < n; i++)
24       e.chars[i] =
25          f[0].chars[i] * f[0].chars[i];
26    for (int i = 0; i < n; i++) {
27       f[0].chars[i] =
28          modPow(e.chars[i],
29                 (int) x.chars[i]);
30       f[1].chars[i] =
31          e.chars[i] *
32          encrypt(pubKeyExt, n-1);
33       f[1].chars[i] =
34          modPow(f[1].chars[i],
35                 (int) x.chars[i]);
36    }
37    b = gcencrypt(
38          "bool compare(int x, int y) {
39             compare = x < y;
40          }",
41          (int) x.len, n);
42    if (b == 1) {
43       e = f[0];
44       f[0] = f[1];
45       f[1] = e;
46    }
47    _f = rerandomize(f, r);
48    otsend(_f);
49    q = recv_string("7");
50    q = add(q, r);
51    return q;
52 }
```

Listing 17: Substring: Masking Receiver

8. CONCLUSIONS

In this paper we consider the problem of ensuring semi-honest security of secure computations implemented in domain-specific languages. Currently, either the language is based on a specific protocol which has been proven secure or on a generic language that enables also implementing insecure protocols. We are the first to propose a type system to limit the implementable protocols to only secure ones, but also enable the programmer to freely choose the protocol primitives. We prove that any well-typed program is secure in the semi-honest model of secure computation. This presents a new trade-off between security and performance for the implementation of secure computations. Our type system enables implementing complex protocols, such as one of our examples for privacy-preserving string processing.

Future work is to extend the security guarantees to stronger security models, such as the malicious model. In the malicious model the parties may behave arbitrarily and still privacy of inputs and correctness of outputs are preserved. Such a security guarantee can be of practical relevance in highly sensitive secure computations such as e-voting.

9. REFERENCES

[1] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. *Proceedings of the 22nd ACM Symposium on Theory of Computing*, 1990.

[2] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-Party Computation. *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. *Proceedings of the 13th European Symposium on Research in Computer Security*, 2008.

[5] D. Bogdanov, R. Talviste, and J. Willemson. Deploying Secure Multi-Party Computation for Financial Data Analysis. *Proceedings of the 16th International Conference on Financial Cryptography and Data Security*, 2012.

[6] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure Multiparty Computation Goes Live. *Proceedings of the 13th International Conference on Financial Cryptography and Data Security*, 2009.

[7] O. Catrina, and F. Kerschbaum. Fostering the Uptake of Secure Multiparty Computation in E-Commerce. *Proceedings of the International Workshop on Frontiers in Availability, Reliability and Security*, 2008.

[8] R. Cramer, I. Damgård and U. Maurer. Efficient General Secure Multi-Party Computation from any Linear Secret-Sharing Scheme. *Proceedings of EUROCRYPT*, 2000.

[9] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography*, 2009.

[10] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM 19(5)*, 1976.

[11] D. Denning, and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM 20(7)*, 1977.

[12] C. Fournet, J. Planul, and T. Rezk. Information-Flow Types for Homomorphic Encryptions. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[13] C. Gentry. Fully Homomorphic Encryption using Ideal Lattices. *Proceedings of the 41st ACM Symposium on Theory of Computing*, 2009.

[14] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikäinen. On Private Scalar Product Computation for Privacy-Preserving Data Mining. *Proceedings of the 7th International Conference on Information Security and Cryptology*, 2004.

[15] O. Goldreich. Secure Multi-party Computation. Available at *www.wisdom.weizmann.ac.il/~oded/pp.html*, 2002.

[16] S. Goldwasser. Multi-Party Computations: Past and Present. *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, 1997.

[17] S. Goldwasser, and S. Micali. Probabilistic Encryption. *Journal of Computer and Systems Science 28(2)*, 1984.

[18] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-Party Computations. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[19] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. *Proceedings of the USENIX Security Symposium*, 2011.

[20] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. *Proceedings of CRYPTO*, 2003.

[21] M. Jensen, and F. Kerschbaum. Towards Privacy-Preserving XML Transformation. *Proceedings of the 9th IEEE International Conference on Web Services*, 2011.

[22] F. Kerschbaum. Automatically Optimizing Secure Computation. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[23] F. Kerschbaum, A. Schröpfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, and E. Damiani. Secure Collaborative Supply Chain Management. *IEEE Computer 44 (9)*, 2011.

[24] F. Kerschbaum, and A. Sorniotti. RFID-based Supply Chain Partner Authentication and Key Agreement. In *Proceedings of the 2nd ACM Conference on Wireless Network Security (WISEC)*, 2009.

[25] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Modular Design of Efficient Secure Function Evaluation Protocols. *Cryptology ePrint Archive Report 2010/079*, 2010.

[26] B. Lampson. A Note on the Confinement Problem. *Communications of the ACM 16(10)*, 1973.

[27] Y. Lindell, and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. *Journal of Cryptology 22(2)*, 2009.

[28] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-party Computation System. *Proceedings of the USENIX Security Symposium*, 2004.

[29] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, 1999.

[30] M. Naor, and B. Pinkas. Efficient Oblivious Transfer Protocols. *Proceedings of the Symposium on Data Structures and Algorithms*, 2001.

[31] D. Naccache, and J. Stern. A New Public-Key

Cryptosystem Based on Higher Residues. *Proceedings of the ACM Conference on Computer and Communications Security*, 1998.

[32] J. D. Nielsen and M. I. Schwartzbach. A Domain-Specific Programming Language for Secure Multiparty Computation. *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, 2007.

[33] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. *Proceedings of EUROCRYPT*, 1999.

[34] A. Sabelfeld, and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications 21(1)*, 2003.

[35] A. Schröpfer, F. Kerschbaum, and G. Müller. L1 - An Intermediate Language for Mixed-Protocol Secure Computation. *Proceedings of the IEEE Computer Software and Applications Conference*, 2011.

[36] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security 4(3)*, 1996.

[37] A. Yao. Protocols for Secure Computations. *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1982.