L1 - An Intermediate Language for Mixed-Protocol Secure Computation

Axel Schröpfer
SAP Research
Karlsruhe, Germany
Email: axel.schroepfer@sap.com

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

Günter Müller
Universität Freiburg, IIG
Freiburg, Germany
Email: guenter.mueller@iig.uni-freiburg.de

***Abstract*—Secure Computation (SC) enables secure distributed computation of arbitrary functions of private inputs. It has many useful applications, e.g. benchmarking or auctions. Several general protocols for SC have been proposed and recently been implemented in a number of compilers and frameworks. These compilers or frameworks implement one general SC protocol and then require the programmer to implement the function he wants the protocol to compute.**

Performance remains a challenge for this approach and it has been realized early on that special protocols for important problems can deliver superior performance.

In this paper we propose a new intermediate language (L1) for optimizing SC compilers which enables efficient implementation of special protocols potentially mixing several general SC protocols.

We show by three case studies – one for computation of the median, one for weighted average, one for division – that special protocols and mixed-protocol implementations in our language L1 can lead to superior performance. Moreover, we show that only a combined view on algorithm *and* cryptographic protocol can discover SCs with best run-time performance.

***Index Terms*—Multi-party Computation, Compiler**

I. Introduction

Secure Computation (SC) allows a set of n players P_i to jointly compute an arbitrary function $f()$ of their private inputs x_i (i.e. P_i has x_i): $f(x_1, \ldots, x_n)$. The computation is privacy-preserving, i.e. it reveals nothing to a player except what can be inferred by his private input and the output of the function. It has many useful applications, e.g. benchmarking [20], [22] and auctions [6].

Even if there are adversarial players, the guarantees for correctness and privacy can be proven to hold in well stated models. These models consider the type of adversary, active (malicious) or passive (semi-honest), and his computing power, bounded or unbounded. A passive adversary follows the protocol as prescribed but tries to learn additional information, while an active adversary may arbitrarily deviate from the protocol.

We distinguish two-party and multi-party (more than two) SC. The first general protocol for two-party SC was presented in [34]. It also defined the approach underlying all general SC protocols. First, the function to be computed is translated into a circuit. Second, each gate of the circuit is implemented using a secure protocol. Two types of gates – multiplication (logical and) and addition (exclusive-or) – suffice to compute any function. This approach reduces constructing a general SC protocol to two protocols implementing these two types of gates. For multi-party SC in the cryptographic model (computationally bounded adversary) with binary circuits this has been solved in [17]. For multi-party SC in the information-theoretic model (unbounded adversary) with arithmetic circuits this has been solved in [5]. Both protocols need to be run interactively, i.e. the number of rounds corresponds to the multiplicative depth of the circuit. Multi-party SC in the cryptographic model has been improved to a constant number of rounds in [3].

Another useful tool for SC is homomorphic encryption (HE), e.g. [12], [15], [28]. In HE one operation on the ciphertext maps to another operation on the plaintexts. Recently [15] it was shown that this can be performed for logical not-and gates which also suffice to compute any function. Threshold HE (for only addition) can be used to implement multi-party SC in the cryptographic model [11]. Also two-party SC can be implemented using HE by adapting the protocol from [16].

In a seminal paper – FairPlay [26] – Malkhi et al. build on the general approach for SC by building a compiler that translates the function specified in a programming language into a circuit. This circuit is then executed using a general SC protocol. In [26] the general protocol was [34]. Later others followed: [33] for [10], [31] for [5], and [4] for [3].

A critical aspect of SC remains performance. Even optimized protocols report a slowdown of several tens of thousand compared to non-privacy-preserving implementation [22]. It has been realized early on that special protocols for important problems can provide better performance [18]. Yet these protocols are not supported in existing SC compilers and frameworks.

This paper contributes

- an *intermediate language* (L1) which can implement special SC protocols. It currently supports general SC protocols [34], [5], [11] and HE [12], [28], but it is also easily extensible to other protocols and encryption schemes.
- a *compiler backend* that translates L1 into interacting Java programs.
- a simple *compiler frontend* for specifying L1 in its own syntax.
- a *benchmarking framework* built into the language and

compiler that supports measuring SC protocols.
- three *case studies*: one for computation of the median which supports that special SC protocols can be faster than general SC protocols, one for weighted average which supports that mixing general SC protocols can be faster than a single general SC protocol and one discovering the fastest combination of algorithm and cryptographic protocol for secure division cross-combining two division algorithms and two secure computation protocols.

The remainder of the paper is structured as follows: Section II reviews related work, in particular other compilers and frameworks for SC. Section III describes the L1 language, compiler and benchmarking framework. We present the case studies in Section IV and our conclusions in Section V.

II. Related Work

A. FairPlay

FairPlay [26] was the first compiler implementing SC. It implements Yao's two-party SC [34]. It provides a simple programming language for specifying the function to be computed. There are some restrictions on this function, such as no variable number of iterations in loops and no recursion. The language provides arrays and variable indexing into arrays which is reported as one of the most complex functions [34]. The compiler translates this language into a circuit which is then executed in Yao's protocol.

The language (SFDL) introduces some of the basic concepts for languages for SMC, such as input and output definition for variables.

B. FairPlayMP

FairPlayMP extends FairPlay to the multi-party setting. It is based on the protocol of [3] which is also an extension of [34] to the multi-party setting. Some code, e.g. in the compiler of the language to the circuit can therefore be shared. The interpretation of the circuit has to completely change, of course. FairPlayMP includes a novel preprocessing phase in order to reduce the communication overhead.

It also implements a popular concept in multi-party SC. Not all parties need to participate equally in the computation. Instead one can separate input, computation and output nodes. As long as there are sufficiently many computation nodes (usually at least three) the computation remains secure, but communication cost has been significantly reduced.

Its programming language includes some necessary extension for multi-party problems and some additional operators.

C. Virtual Ideal Functionality Framework

VIFF [33] is the basis for the first commercial application of SC [6]. It implements multi-party SC in the information-theoretic model [10]. It does not yet provide a high-level programming language for the functionality. Instead it extends the Python language to use operators on the basic data types. It implements the basic protocols for [10] as objects.

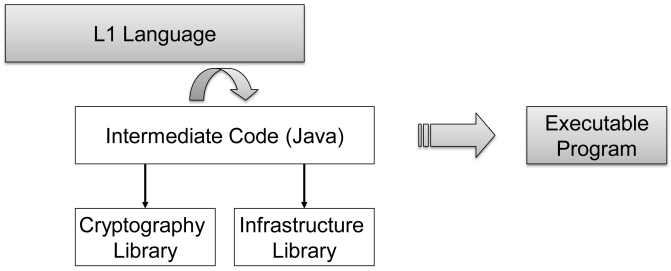

Fig. 1. Translation process from L1 source to executable program

A high-level language (SMCL) for VIFF has been specified in [27]. It extends FairPlay's language by some features such as loops with public number of iterations and allows security analysis similar to information flow analysis. For this purpose it tags all variables in a lattice as either secret, private or public.

D. Sharemind

Sharemind [31] is a framework built for experimenting with privacy-preserving data mining. It implements the SC protocol from [5]. It has been optimized for speed of simple operations, such as vector products. It therefore offers vectorized operations.

It is currently being extended with a programming language (SecreC) for specifying the function to be computed.

E. TASTY

Next to these implementations of one SC protocol, implementations of mixed protocols have emerged. These combine several protocols, such as garbled circuits and homomorphic encryption. The authors of [19] present a tool which allows to express a computation whose segments are translated in either GC or HE based sub-protocols. The work evaluates an exemplary SC and shows the performance improvement gained by mixing techniques.

III. L1 Language

The L1 language was designed to represent specialized SC protocols potentially mixing techniques from multiple general SC protocols. We expect this mix to enable significantly more efficient SC protocols as shown in our case studies in Section IV.

We have built a compiler backend that translates a SC protocol encoded in L1 into a set of interacting Java programs – one for each player in the protocol. The players can then compile and execute the Java program in order to run the protocol programmed in L1, as depicted in Figure 1.

A. Design Principles

The purpose of the L1 language is twofold. First, it is supposed to be used as intermediate language by an optimizing compiler for SC. It therefore has to have the expressive power for a wide range of SCs. It also must be translatable into efficient code at each party's site. We have implemented a compiler back-end for this purpose and show in a number of case studies (Section IV) that the performance of our

implementation significantly surpasses that of general SC compilers.

Second, we have used L1 to implement a variety of – even large-scale – secure computations. For example, we have implemented secure linear programming [32] using L1. In order to make this task acceptable to the programmer a L1 syntax has to be sufficiently easy to use. It has to abstract tasks not directly relevant to the SC, such as network programming. We have implemented a simple compiler front-end for this purpose and describe its syntax next (Section III-B).

An optimizing compiler for SC proceeds in several steps. As input it receives a specification of the functionality f to be computed. As output it produces code to be run at a player's site. The intermediate language sits in between and must help bridge the semantic gap between these two approaches.

We made two fundamental design choices. First, we represent the entire functionality in the intermediate language, i.e. we do not yet divide it into player-specific, interacting programs. Yet, we do represent network communication by primitives. This choice was made to later enable optimizations and security verifications on the entire program using L1 as an intermediate language. Such analyses are significantly complicated when considering interacting programs.

Second, we allow modularization of the functionality. Large SC are often iteratively composed of smaller sub-protocols. We intend to allow re-use of already programmed sub-protocols, e.g. in a library.

In the next section we describe the constructs of L1 by describing the syntax features of L1.

B. Syntax

Since many, if not most, programmers are familiar with Java or C, we chose our syntax similar to these languages in order to reduce the initial burden of adoption. We reduced the set of language constructs to the essential needs for implementing SC protocols. It is important for us to be able to implement SC protocols quickly and efficiently by providing the necessary language constructs, but avoid too much "syntactic sugar" in order to keep language and compiler maintainable. The L1 language contains the following constructs

- variables of different data types (*bool, int32, int, string, prvk, pubk*) in different composites (scalar, 1-dimensional and 2-dimensional array),
- expressions (assignments and operators),
- control flow (*if, while, for*, sequential and parallel execution, player-specific code),
- functions (user-defined and built-in),
- modules (*include*).

Listing 1 shows a sample of L1 source code demonstrating all constructs of the language.

```
1  //modules
2  include "key.l1";
3
4  //self defined functions
5  int newHash(int value) {
6    int hash;
7    ...
8    return hash;
9  }
10
11 //variables
12 int hash;
13
14 //assignment with expression
15 int salt = rand(1000)+1;
16
17 //player dependent statement
18 1: {
19   //function call
20   startBenchmark("hash");
21   hash = newHash(salt);
22   stopBenchmark("hash");
23
24   //message sending (non-blocking)
25   send(2,hash,"hash_value_from"+id());
26 }
27
28 //message receiving (blocking)
29 2: readInt("hash_value_from_1");
30
31 if (hash%2 == 0)
32   output("odd hash: "+hash);
33
34 //for-loop
35 for (int32 i=0; i<200; i=i+1) <
36 //parallel execution
37   ...
38 >
39
40 //while-loop
41 while (condition) {
42   ...
43 }
```

Listing 1. L1 sample

C. Variables and Data Types

L1 supports variables of various basic data types. Furthermore L1 supports composites of a basic data type; in particular are supported: scalar (single value), 1-dimensional array (vector) and 2-dimensional array (matrix). The following basic data types are supported: *bool*, *int32*, *string*, *int*, *prvk*, *pubk*. *bool* and *int32* are the 1 and signed 32 bit integers as known from Java and many other programming languages. We have implemented the *true* and *false* keywords for Boolean variables. *string* is a built-in basic data type that behaves like Java String objects, i.e. a variable length array of characters.

1) int: SC has to uses of integers: within the (securely) computed function and as building blocks of that computation. For integers used in the function, such as for array access, counters or arithmetic computations, 32 bits are usually sufficient. The compiler translates *int32* into the Java native data type *int* (or its wrapper class if necessary). Operations on native data types are comparatively fast, since they can be translated into operations on CPU registers.

For integers used as building blocks in the computation, such as secret shares or ciphertexts, 32 bits are rarely sufficient. Using the example for secure linear programming [32], secret shares [30] can extend to several hundred bits in length. The bit length of cryptographic keys and ciphertexts is lower bound by computationally difficult problems [25]. We therefore im-

plement the data type *int* of variable bit length integers for multi-precision arithmetic. We use Java's BigInteger library in order to implement *int*. The compiler translates operators on *int* into the corresponding method calls.

With the combination of those two basic integer data types, L1 can address the needs for both: fast operations on small values and secure operations on large values.

2) prvk: The basic data type *prvk* is assigned to variables containing private keys of public key encryption schemes. Introducing its own basic data type enables the use of operators on that data type, e.g. for derivation of public keys. Furthermore it enables the built-in functions (see Section III-F1) for encryption and decryption to determine the corresponding encryption scheme. The compilers translate variables *prvk* into our own class `PrivateKey` which is accessible to and extensible by built-in and library functions. It is therefore easy to extend the compiler with new encryption schemes, not yet implemented in L1, that inherit from this class. The additional classes simply need to be integrated into the L1 library and are then readily accessible from the L1 language.

3) pubk: The basic data type *pubk* is the public key companion of *prvk*. Clearly it contains only the public part of the cryptographic key necessary for encryption. The compiler translates it into our class `PublicKey` which in case of extensions also needs to be implemented.

D. Expressions

As in almost all other imperative programming languages, expressions in L1 consist of operands which may be connected through operators. Operands can be function calls, variables or constants. In case of a type mismatch between operands we implicitly upcast *int32* to *int* and any data type to *string*. For other cases L1 provides built-in functions for explicit type conversion. Most operators and their handling correspond to Java and are directly translated into their Java counterpart by the compiler.

E. Control Flow

The basic control flow constructs in L1 are quite standard for an imperative programming language and closely adhere to Java. L1 provides the basic constructs *if-else*, *for* and *while* in combination with basic blocks. The compiler translates them directly into their Java counterpart. One notable difference occurs in case of the index variables in *for* loops. In L1 these variables are pass-by-value (instead of pass-by-reference in Java). This difference is necessary in case the loop spawns multiple threads which all access the index variable. We describe how to spawn multiple threads next.

1) Parallel Execution: Besides the Java semantics of sequential statements and basic blocks L1 offers parallel execution of basic blocks. As mentioned before SC protocols can be very computational intensive [22] challenging the performance of a single CPU. In some cases [6], [14] it is already known that SC protocols can be quite efficiently parallelized capitalizing on the trend to multi-core CPUs. Since most large-scale SC problems have not yet been tackled due to performance concerns and CPU speed does not seem to continue to increase exponentially, we expect that future implementations will need to heavily exploit parallelism.

```
1 output("S1");
2 for (int32 i=1; i<=2; i=i+1)
3 <
4   output("P"+i);
5 >
6 output("S2");
```

Listing 2. L1 Parallel Execution

L1 offers a unique feature for the definition of parallel code sections. Basic blocks to be execute as a new thread are specified by angle brackets as delimiters (instead of curly brackets). The compiler inserts the necessary instructions into the Java code in order to spawn and execute a new thread containing this basic block. Afterwards, the execution continues and any adjacent parallel basic blocks are spawned and run in parallel threads. L1 will also synchronize those threads before returning to sequential processing.

All parallel threads register with a barrier before executing the L1 basic block. A barrier is a synchronization mechanism that blocks execution until all registered threads have finished. The compiler uses the Java standard library class for barriers. After spawning parallel threads the L1 compiler inserts a call to the barrier to wait until all threads have finished before executing the next sequential statement.

Listing 2 shows an example for thread synchronization in L1. Line 1 contains an initial sequential statement that outputs "S1". The body of the *for* loop is a parallel basic block which will spawn and run two parallel threads. One outputs "P1" and one "P2". The last statement is a sequential one again and outputs "S2".

The barrier mechanism of L1 synchronizes the threads, such that the last statement will always be executed last, i.e. the first and last output of the program will always be "S1" and "S2", respectively. The only two possible traces of the program are: S1, P1, P2, S2 and S1, P2, P1, S2.

2) Player-Specific Code: In many SC protocols, particularly in almost all general multi-party SC protocols, all players execute the same code (just on different data). The L1 compiler therefore produces several instances of the Java code – one for each player.

Some SC protocols, e.g. Yao's two-party SC [34], deviate from this pattern and execute different code for different players. Since the player identifier is accessible within L1, the differentiation could be performed at run-time using an *if* statement. Instead we chose for performance reasons to differentiate at compile time and potentially produce different Java code for each player. A programmer can specify *player-specific code* sections in the L1 source. A player-specific code section is a statement or basic block prepended by the identifier of the player to execute this code followed by colon. Line 18 of Listing 1 shows an example of a player-specific code section.

The interpretation at compile time ensures leaner code at each player that only needs to execute the statements for this

```
1 public static BigInteger inv(
2   BigInteger value, BigInteger modulus)
3 {
4   return value.modInverse(modulus);
5 }
```

Listing 3. L1 built-in function

player. Furthermore we feel that it makes the L1 code easier to read and maintain.

F. Functions

Like procedural languages L1 structures its code into functions, but L1 offers two types of functions: user-defined and built-in functions.

User-defined functions are specified and compiled as expected from the similarity to the C language. As a constraint we currently require the definition of a user-defined function before its first invocation. Line 5 of Listing 1 shows an example of a function definition.

1) Built-In Functions: Built-in functions are programmed in Java and not L1, but can be called from L1 just like any other function. Built-in functions are defined in a Java class `BuiltInFunctions` of the compiler. The compiler uses reflection in order to import the built-in functions. Using built-in functions the compiler can be extended with new features. Listing 3 shows a built-in function for computing the inverse in a field.

Built-in functions can be polymorphic with respect to the parameter type. If the parameter type is `Object`, the compiler creates a function instance for each L1 data type including composites for arrays and matrices.

Many features of the L1 language have been implemented as built-in functions. Two which are particularly worth mentioning are

- messaging
- benchmarking

a) Messaging: Messaging allows the transmission of messages between players enabling the distributed (secure) computation. L1 provides two sub-systems both based on TCP/IP: synchronous and asynchronous.

The built-in functions *send* and *sendSync* send messages to other players (line 25). Their parameters are the identifier of the receiving player, a name for the message and its value. If the identifier of the player is 0, the player will broadcast to all other players. The name of the message is a replacement of its address and used by the recipient to retrieve the message in case of asynchronous communication.

The asynchronous *send* function implements non-blocking behavior (i.e., the next statement in line will be executed immediately). The synchronous *sendSync* will block the execution until the message has been acknowledged by all recipients. Synchronous send also supports an optional timeout parameter. Furthermore L1 also supports buffered sends which bundle several send invocations.

The recipient has a built-in receive (*read*) function for every data type. These functions require the message name as a parameter (line 29). Message receiving is always blocking, i.e., the read function will block and wait until the message with the specified name has been received. An optional timeout can be specified as a second parameter or else a default timeout is used.

b) Benchmarking: The design goal of L1 is programming faster SC protocols. Measuring the performance improvement therefore enables verifying whether this goal has been reached.

L1 provides a benchmarking sub-system using built-in functions. Several benchmarks can be measured in parallel. Each benchmark is started by calling the built-in function *startBenchmark* (line 20) and stopped by calling *stopBenchmark* (line 22). Its parameter is the name for this benchmark called a benchmarking section. L1 implicitly takes care of multiple threads by internally appending the thread identifier to the name.

In a benchmarking section the following quantities are captured

- run time (wall clock time)
- number of messages sent or received
- number of bytes sent or received

These correspond to computation and communication complexity in theoretic papers on SC.

G. Modules

A programmer can structure larger L1 programs into modules. Each module is stored as a separate file and can be loaded using the *include* statement (line 2).

We successfully used modules to analyze complex SC protocols which are composed of several sub-protocols (e.g. see [32]). Modules allow easily swapping sub-protocols for different implementations and then benchmarking the composed functionality may reveal novel dependencies and side-effects.

H. Discussion on Security

The foremost requirement for a SC is to be secure. It is common practice to prove protocols secure in a well stated security model (most often semi-honest or malicious). The L1 language encompasses no functionality that ensures security. This is in opposition to SC compilers using one general SC protocol, such as FairPlay [26].

Nevertheless, this option was chosen by design. For efficient SC, such as our first case study (Section IV-A), there are simply no automated verification techniques. Our choice is therefore to either not represent these protocols in our language or remove the constraint for automated verification.

Without security enforcement on the intermediate language layer, immediately the question of correctness for the compiler is raised. Clearly, one can verify the correctness – security preservation – of each optimization on the L1 language. Instead, we anticipate to retrofit L1 with automated verification techniques as soon as they become available. In our view, L1 offers an ideal language for automated verification and can serve as the basis for many verifications of SCs.

Furthermore, a verified intermediate language, such as L1 offers significant advantages in tackling side channel attacks. SC protocols secure in the semi-honest model are side-channel free by design. If then all primitives of the language libraries are implemented side-channel free, the composed SMC protocol will be side-channel free.

IV. Case Studies

Using three case studies we exemplify the performance benefit of using L1. In the first case study we compare two SC protocols for median computation, one entirely implemented using Yao's protocol [34] in FairPlay [26] and one implemented in L1. Although both implement the same function, the performance results are quite different.

In the second case study we compare two SC protocols for weighted average computation, again one entirely implemented using Yao's protocol and one implemented in L1 also using an adaptation of Goethals et al.'s protocol [16].

The reported runtime measurements is the average of 20 samples. Tests were performed on two 2GHz Intel CoreDuo T7200 dual CPU machines with 2GB RAM each.

The third case study investigates combinations of algorithm and cryptographic protocols for the problem of secure division. We show how fast algorithms can be implemented in L1 using different underlying protocol: either garbled circuit or homomorphic encryption. The gained insights on performances of algorithm/protocol combinations are remarkable.

A. Median

Imagine two players, Alice and Bob, each with n elements drawn from a finite domain. They want to compute the median of their joint set of elements, i.e. the n-th ranked element in their combined, (ascendingly) sorted set, but do not want to disclose any of their other elements to the other party.

Aggarwal et al. proposed the following protocol for this problem [1]. Alice and Bob compare the median of their individual sets using a SC protocol for comparison, i.e. they only learn the result of the comparison, but not each other's input values. The party with the lower value selects the upper half of its elements and the party with the higher value selects the lower half of its elements. They then repeat the comparison with set half the initial size and continue doing so until the sets are of size 1.

We implement this algorithm in L1 (see Listing 5). We also perform comparisons in L1 using Yao's protocol, but do not show the code in the paper, since it is quite standard and import the function via a module. Note that using Yao's protocol one could implement the same algorithm, but unfortunately FairPlay does not support the necessary operations, such as division (or shift). We therefore use FairPlay's example for median from its distribution (see Listing 4).

The key insight of [1] is that the result of the comparison and subsequent selection of elements can be public (known to both parties), since it can be inferred from the (public) result of the computation. FairPlay's problem is that this insight cannot be implemented in its code while L1 can implement it.

```
1  include "compareYao.l1";
2  include "minimumYao.l1";
3  int32 aggarwal() {
4    int32 bits=32;
5    int32 j=4;
6    int32 k=pow(2, j);
7    int32 s[]=loadInt32Matrix("med.txt");
8    int32 offset = 0;
9    for (int32 i=0; i<j; i=i+1) {
10     k = k / 2;
11     int32 m=s[offset + k];
12     int32 r=comp(m, bits);
13     1: if (r==1)
14          offset=k;
15     2: if (r==0)
16          offset=k;
17   }
18   return min(s[offset], bits));
19 }
20 1: output(aggarwal());
21 2: output(aggarwal());
```

Listing 5. Median by Aggarwal et al. in L1

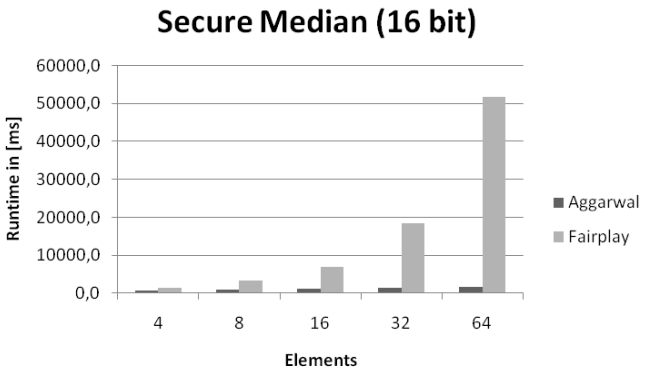

Fig. 2. Secure Median with 16 bit

The variable indexing into the array of elements is an $O(n)$ operation in FairPlay while it is almost free in L1, since it does not require any communication.

We ran the protocols for varying numbers of elements (4, 8, 16, 32 and 64) and also for varying input bit lengths (16 and 32).

Table I shows the runtime. Figures 2 and 3 depict the results as graphs. L1 always outperforms FairPlay and L1's advantage is increasing with an increasing number of elements. Unfortunately the results are distorted by FairPlay's lack of operations, but we believe that even if the more efficient algorithm would have been implemented in Yao's protocol L1's advantage of public selection of the remaining elements would have prevailed.

B. Weighted Average

Suppose Alice has n elements and Bob has m elements drawn from a finite domain. Let c be the sum of Alice's elements and d be the sum of Bob's. Furthermore, Alice and Bob share a weight w, such that Alice has w_A, Bob has w_B and $w = w_A + w_B$. This also covers less general problems

```
1  program Median {
2    const inp_size = 64;
3    type Elem  =  Int<32>;
4    type AliceInput = Elem[inp_size];
5    type AliceOutput = Int<32>;
6    type BobInput = Elem[inp_size];
7    type BobOutput = Int<32>;
8    type Input = struct {AliceInput alice, BobInput bob};
9    type Output = struct {AliceOutput alice, BobOutput bob};
10
11   function Output output(Input input) {
12     var Int<8> i;
13     var Int<8> ai;
14     var Int<8> bi;
15     ai=0;
16     bi=0;
17     for (i = 1 to inp_size-1) {
18       if (input.alice[ai] >= input.bob[bi])
19         bi = bi + 1;
20       else
21         ai = ai + 1;
22     }
23     if (input.alice[ai] < input.bob[bi]) {
24       output.alice = input.alice[ai];
25       output.bob = input.alice[ai];
26     } else {
27       output.alice = input.bob[bi];
28       output.bob = input.bob[bi];
29     }
30   }
31 }
```

Listing 4. Median in Fairplay

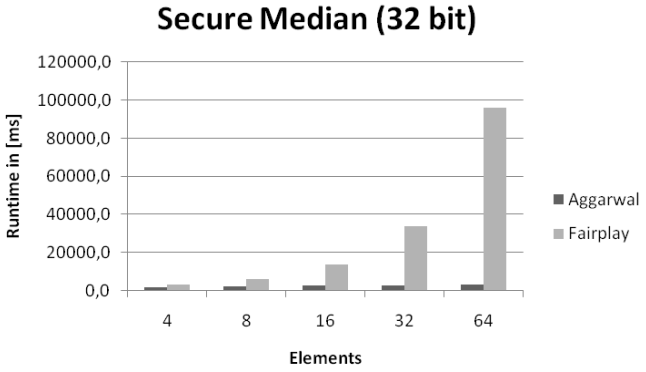

Fig. 3. Secure Median with 32 bit

TABLE I
RUNTIME IN [MS] FOR SECURE MEDIAN WITH 16 AND 32 BIT

Elements per Player	Aggarwal 16 bit	Fairplay 16 bit	Aggarwal 32 bit	Fairplay 32 bit
4	730,5	1525,9	1439,9	2838,4
8	931,3	3308,9	1885,3	6001,1
16	1164,9	7043,3	2353,2	13543
32	1395,4	18473,4	2793,9	33950,8
64	1620,4	51760,8	3250,2	96166,9

where either only Alice or only Bob has the weight w. They want to jointly (and securely) compute the weighted average f:

$$f = \frac{(c+d)(w_A + w_B)}{n+m} \tag{1}$$

We implement this formula in Yao's protocol [34]. Due to a minor implementation error in FairPlay we had to resort to our own implementation of [34].

We then replaced the multiplication in the divisor in Yao's protocol by a variant of the protocol from [16]. We briefly review this variant.

Let Alice have a and Bob b they can compute $x + y = ab$ using HE. Alice sends $E_A(a)$ to Bob who computes $E_A(a)^b E_A(R) = E_A(ab + R)$ and returns it to Alice. Alice decrypts and stores the result as x while Bob sets $y = -R$. When combining Yao's protocol with HE care must be taken when choosing the length of the secret number R. Secret shares, such as in $w_A + w_B$ have significantly less bits than the key length of most HE schemes. Therefore R should be chosen longer according to the technique from [13] and later reduces using a modulo operation on the plaintexts. See Listing 7 for details. Listing 6 contains the remaining code of the mixed SC protocol in L1 except an imported module for division using Yao's protocol.

We expect a significant performance improvement by the mixed SC protocol, simply because using HE one multiplication can be implemented using one operation (albeit an expensive one) as in an arithmetic circuit while Yao's protocol works on binary circuits where integer multiplication must be (cumbersomely) emulated using $O(n^2)$ operations (gates). This insight of cleverly combining arithmetic and binary circuits has also been noted in [24].

Table II shows the runtimes of both protocols for 16 and 32 bits of input length. Figure 4 shows the same results as a graph.

```
1  include "utilSplitMul.l1";
2  include "utilAddDivYao.l1";
3
4  int32 weightedAverageHEGCPlayer1() {
5    int32 c = input("c:");
6    int32 wA = input("wA:");
7    int32 n = input("n:");
8
9    prvk paillierPrivateKey = createPaillierPrivateKey(1024);
10
11   int modulus = getModulus(paillierPrivateKey);
12
13   pubk paillierPublicKey = getPublicKey(paillierPrivateKey);
14
15   send(2, "pubKey", paillierPublicKey);
16
17   int32 p1 = splitMulPlayer1(c, wA, paillierPrivateKey);
18
19   return addDivCircuitPlayer1(bits, p1, n);
20 }
21
22 int32 weightedAverageHEGCPlayer2() {
23   int32 d = input("d:");
24   int32 wB = input("wB:");;
25   int32 m = input("m:");
26
27   pubk paillierPublicKey = readPubk("pubKey");
28   int modulus = getModulus(paillierPublicKey);
29
30   int32 p2 = splitMulPlayer2(d, wB, paillierPublicKey);
31
32   return addDivCircuitPlayer1(bits, p2, m);
33 }
34
35 1: output(weightedAverageHEGCPlayer1());
36 2: output(weightedAverageHEGCPlayer2());
```

Listing 6. Weighted average using Yao's protocol and HE in L1

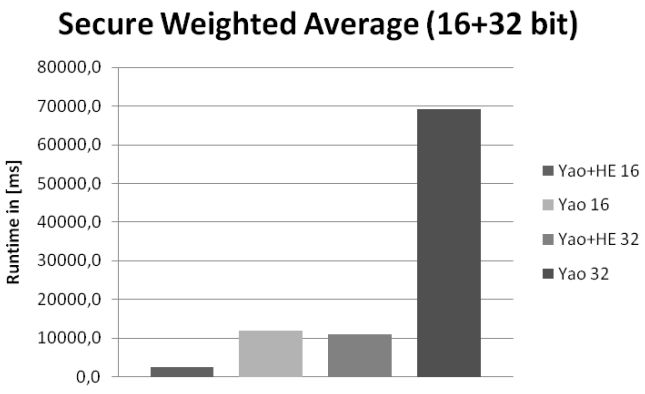

Fig. 4. Secure Weighted Average with 16 and 32 bit

TABLE II
RUNTIME IN [MS] FOR SECURE WEIGHTED AVERAGE WITH 16 AND 32 BIT

Input Size	GC+HE	GC
16	2446,2	12034,1
32	11009,8	69149,6

As anticipated the mixed SC protocol clearly outperforms Yao's protocol.

C. Secure Division

Secure division is relevant for many real world secure computations, e.g., k-means clustering or supply chain optimization [29]. Various cryptographic protocols for secure two-party and multi-party division protocols have been proposed (e.g., [2], [23], [7], [9]). They use different approaches in terms of algorithm and cryptographic protocol. However, it is difficult to determine which one performs best.

In this case study we compare the run-times of protocols cross-combining two different algorithms for division and two different secure two-party computation protocols. The choices are long division (aka school method) (LD) and New Raphson approximation (NR) for the algorithms and garbled circuits (GC) and homomorphic encryption (HE) for the secure computation protocol. We describe how fast we can implement the programs LDGC, LDHE, NRGC and NRHE. We then perform run-time benchmarks of all four programs.

Our results show that this formerly exhausting approach of comparing secure computations now becomes tractable using L1. It also shows that this approach allows a more accurate assessment of the performance than using standard metrics like computational complexity, communication complexity and round complexity.

We first generalize the set of required operations in both algorithms to be

```
1  int32 splitMulPlayer1(int32 x1, int32 y1, prvk prvKey) {
2    int N = getModulus(prvKey);
3    int N2 = N*N;
4    pubk pubKey = getPublicKey(prvKey);
5    int x1enc = encrypt(pubKey,x1);
6    int y1enc = encrypt(pubKey,y1);
7    send(2,"x1Enc",x1enc);
8    send(2,"y1Enc",y1enc);
9    int v = readInt("v");
10   int vdec = decrypt(prvKey,v);
11   int32 vdecInt32 = intToInt32(zToI(vdec,N)%pow(2,32);
12
13   return (vdecInt32+(x1*y1))%pow(2,32);
14 }
15
16 int32 splitMulPlayer2(int32 x2, int32 y2, pubk pubKey) {
17   int x1enc = readInt("x1Enc");
18   int y1enc = readInt("y1Enc");
19   int N = getModulus(pubKey);
20   int N2 = N*N;
21   int32 r = rand(pow(2,32));
22   int v = (modPow(x1enc,y2),N2)*modPow(y1enc,x2),N2)*encrypt(pubKey,r))%N2;
23   send(1,"v" v);
24
25   return (-r+x_2*y_2)%pow(2,32);
26 }
27
28 1: output(weightedAverageHEGCPlayer1());
29 2: output(weightedAverageHEGCPlayer2());
```

Listing 7. Split Multiplication Protocol

- addition $\oplus$
- subtraction $\ominus$
- scalar multiplication $\odot_\epsilon$ with operands being vectors of ϵ elements
- multiplication by a constant $\odot_c$
- division by a constant $\oslash_c$
- left shift $\ll$
- right shift $\gg$
- less-or-equal $\leq$

We generate for the GC variant the binary circuits matching LD and NR using the construction in [29]. The circuits are imported by the corresponding L1 built-in functions.

For the HE variant we implement a user-defined function for each operator. While some operators like $\oplus$ are local operators (i.e., addition of field values), others have to be implemented as sub-protocols. We use $\odot_\epsilon$ by [16], $\oslash$ by [2] and $\leq$ by [21].

The compiler generates for each combination of algorithm and two-party protocol the corresponding pair of Java programs. We then benchmark the secure division implementations for an exemplary environmental settings. For the benchmark we use the following setup. We deploy the programs on two servers each hosting four AMD Opteron 885 dual-core 64-bit CPUs and 16 GB RAM. We connect the servers via a PC in the middle which is running dummynet [8]. Dummynet is a tool for emulating network conditions.

We perform the benchmark using the following parameters. Let l denote the bit-length of input and output values. Let k_{HE} denote the key-length of the used HE scheme and k_{GC} the key-length for the one-way hash function used in GC. Let b denote the bandwidth of the network connection and t_{LAT} denote the latency of network packets.

We perform a benchmark to get an overview on the four programs. We set $l = 8$ bit, $k_{HE} = 1024$ bit, $k_{GC} = 128$ bit, $b = 5$ Mbit/s and $t_{LAT} = 50$ ms. The network settings exemplary represent WAN connections in small and medium enterprises.

The results are depicted in Figure 5. NRGC yields the slowest variant. The circuit we generated for NRGC is an order of magnitude larger than that of LDGC. Since the run-time for the GC protocol is linear in the size of circuits, run-time of NRGC is an order of magnitudes slower than LDGC. Also for LDHE we measure noticeably slower run-times than for LDGC and NRHE. Hence, neither NRGC nor LDHE are a candidate for the best performing variant. The best performing variant must be either LDGC or NRHE.

We conclude from the diagram that it is neither the algorithm itself nor the cryptographic protocol itself determining the best performing protocol. The modularity and ease of use of the L1 syntax enabled our rapid experiments underpinning this conclusion.

V. Conclusion

We have presented the L1 intermediate language intended to implement mixed-protocol SC. It supports different general SC protocols, such as using secret shares [5], [10], homomorphic encryption [11], [16] and garbled circuits [34]. It also supports special SC protocols designed for important problems.

The intention of mixed SC protocols is to improve performance and we tested this hypothesis using three case studies. In one case study a special SC protocol was compared to a general SC protocol. In the second case study a single SC protocol was compared to a mixed SC protocol. In both cases

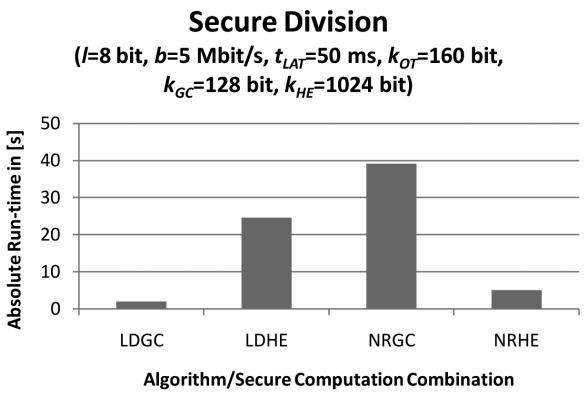

Fig. 5. Run-time of secure division with $l = 8$ bit, $k_{HE} = 1024$ bit, $k_{GC} = 128$ bit, $b = 5$ Mbit/s and $t_{LAT} = 50$ ms.

L1 provides superior performance. A third case study showed how L1 fosters optimal decisions in practice on selecting the fastest combination of algorithm and cryptographic protocol.

We used L1 internally for a number of experiments and implementations of SC. Future work is to continually enhance it to cover different techniques, encryption schemes and protocols. Its built-in benchmarking sub-system allows to perform experiments efficiently leading to novel insights on the practical performance of SC protocols.

A. Future Work

We intend to develop on top of L1 another, more high level programming language for cryptographic protocols, called L2. L2 will support more abstract primitives like secure data types, with their operations compiled into a mixed protocol in L1 optimized for performance. We also aim to introduce in L1 support for automatic security verification of the cryptographic protocol programs.

REFERENCES

[1] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the k th-ranked element. In *EUROCRYPT*, pages 40–55, 2004.

[2] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the ACM Workshop on Privacy in an Electronic Society*, pages 103–114, 2004.

[3] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.

[4] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, 2008.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault tolerant distributed computation. In *Proc. of 20th ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.

[6] P. Bogetoft, D. L. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Kroigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *13th International Conference on Financial Cryptography and Data Security*, 2009.

[7] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 486–497, 2007.

[8] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40:12–20, 2010.

[9] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin / Heidelberg, 2010.

[10] R. Cramer, I. Damgard, and U. Maurer. General secure multi-party computation from any linear secret sharing scheme. In *Eurocrypt*, 2000.

[11] R. Cramer, I. Damgard, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proceedings of EUROCRYPT, Lecture Notes in Computer Science 2045*, pages 280–299, 2001.

[12] I. Damgard and M. Jurik. A generalisation, a simplification and some applications of paillier probabilistic public-key system. In *Proceedings of International Conference on Theory and Practice of Public-Key Cryptography, Lecture Notes in Computer Science 1992*, pages 119–136, 2001.

[13] I. Damgard and R. Thorbek. Efficient conversion of secret-shared values between different fields, 2008.

[14] R. Deitos and F. Kerschbaum. Improving practical performance on secure and private collaborative linear programming. In *DEXA Workshops*, pages 122–126, 2009.

[15] C. Gentry. Fully homomorphic encryption using ideal lattices. In *41st Annual ACM Symposium on Theory of Computing*, 2009.

[16] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikäinen. On private scalar product computation for privacy-preserving data mining. In *7th International Conference on Information Security and Cryptology*, 2004.

[17] O. Goldreich, S. Micali, , and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Symposium on the Theory of Computing*, pages 218–229, 1987.

[18] S. Goldwasser. Multi-party computations: Past and present. In *16th ACM Symposium on Principles of Distributed Computing*, 1997.

[19] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 451–462. ACM, 2010.

[20] F. Kerschbaum. Practical privacy-preserving benchmarking. In *23rd IFIP International Information Security Conference*, 2008.

[21] F. Kerschbaum, D. Biswas, and S. de Hoogh. Performance comparison of secure comparison protocols. In *Database and Expert Systems Application, 2009. DEXA '09. 20th International Workshop on Business Processes Security*, pages 133 –136, 2009.

[22] F. Kerschbaum, D. Dahlmeier, A. Schröpfer, and D. Biswas. On the practical importance of communication complexity for secure multi-party computation protocols. In *24th ACM Symposium on Applied Computing*, 2009.

[23] E. Kiltz, G. Leander, and J. Malone-Lee. Secure computation of the mean and related statistics. In *Proceedings of Theory of Cryptography Conference, Lecture Notes in Computer Science 3378*, pages 283–302, 2005.

[24] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Modular design of efficient secure function evaluation protocols. Cryptology ePrint Archive, Report 2010/079, 2010.

[25] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *J. Cryptology*, 14(4):255–293, 2001.

[26] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of the USENIX security symposium*, pages 287–302, 2004.

[27] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 21–30, New York, NY, USA, 2007. ACM.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EUROCRYPT, Lecture Notes in Computer Science 1592*, pages 223–238, 1999.

[29] R. Pibernik, Y. Zhang, F. Kerschbaum, and A. Schröpfer. Secure collaborative supply chain planning and inverse optimization - the jels model. *European Journal of Operational Research*, 208(1):75–85, 2011.

[30] A. Shamir. How to share a secret. In *Communications of the ACM, 22(11)*, pages 612–613, 1979.

[31] Sharemind. http://sharemind.cs.ut.ee/wiki/, 2010.

[32] T. Toft. *Primitives and Applications for Multi-party Computation*. PhD thesis, Department of Computer Science, University of Aarhus, 2007.

[33] Virtual Ideal Functionality Framework. http://www.viff.sk, 2010.

[34] A. Yao. How to generate and exchange secrets. In *In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.