PICCO: A General-Purpose Compiler for Private Distributed Computation

Yihua Zhang, Aaron Steele, and Marina Blanton
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA
{yzhang16,asteele2,mblanton}@nd.edu

ABSTRACT

Secure computation on private data has been an active area of research for many years and has received a renewed interest with the emergence of cloud computing. In recent years, substantial progress has been made with respect to the efficiency of the available techniques and several implementations have appeared. The available tools, however, lacked a convenient mechanism for implementing a *general-purpose* program in a secure computation framework suitable for execution in not fully trusted environments. This work fulfills this gap and describes a system, called PICCO, for converting a program written in an extension of C into its distributed secure implementation and running it in a distributed environment. The C extension preserves all current features of the programming language and allows variables to be marked as private and be used in general-purpose computation. Secure distributed implementation of compiled programs is based on linear secret sharing, achieving efficiency and information-theoretical security. Our experiments also indicate that many programs can be evaluated very efficiently on private data using PICCO.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: Security and Protection; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

Keywords

Secure multi-party computation; secure computation outsourcing; general-purpose computation; compiler; source-to-source translator; linear secret sharing; parallel execution

1. INTRODUCTION

This work is motivated by the broad goal of developing techniques suitable for secure and general data processing and outsourcing. The desire to compute on sensitive data without having to reveal more information about the data than necessary has led to several decades of research in the area of secure multi-party computation (SMC). Today, cloud computing serves as a major motivation for the development of secure data processing techniques suitable for use in outsourced environments. This is because security and privacy considerations are often cited as one of the top impediments to harnessing the benefits of cloud computing to the fullest extent.

Despite the sheer volume of research literature on privacy-preserving computation and newly appearing secure outsourcing techniques, most of the available techniques focus on a rather narrow domain, namely, integer-based arithmetic. Little or no attention has been paid to other types of computation, as well as to data structures and algorithms suitable for secure data processing in not fully trusted environments. With the recent progress in the performance of basic secure computation techniques and the shift toward cloud computing, we believe that it is the prime time to enable privacy-preserving execution of any functionality or program, or *general-purpose* secure data processing.

Toward this goal, this work introduces PICCO (Private dIstributed Computation COmpiler) — a system for translating a general-purpose program for computing with private data into its secure implementation and executing the program in a distributed environment. The main component of PICCO is a source-to-source compiler that translates a program written in an extension of the C programming language with provisions for annotating private data to its secure distributed implementation in C. The resulting program can consequently be compiled by the native compiler and securely run by a number of computational nodes in the cloud or similar environment. Besides the compiler, PICCO includes programs that aid secure execution of user programs in a distributed environment by preprocessing private inputs and recovering outputs at the end of the computation.

Our desire to build as general of a tool as possible is supported by two important aspects of this work:

- We make a distinction between the participant(s) who hold private inputs, participant(s) who receive the output, and computational parties who conduct the computation. This allows the framework to be used in many contexts including privacy-preserving collaborative computation with multiple participants and secure computation outsourcing by one or multiple clients.
- The goal of this work is to support as wide of a range of functionalities as possible, i.e., as long as the functionality is known at the run-time, it can be securely evaluated in our framework. Toward this goal, we supplement the functionality of C with a number of private data types and operations on them, and incorporate them in standard types of constructions used in C. For example, unlike other compilers, PICCO has support for floating point arithmetic on private data.

We hope that our system will aid accessibility and wider adoption of general-purpose secure computation and outsourcing.

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http:///dx.doi.org/10.1145/2508859.2516752.

Performance is a key factor for secure computation. For that reason, PICCO utilizes lightweight information-theoretically secure techniques based on secret sharing to build secure distributed implementations of user programs. The resulting implementations promise to be particularly efficient and suitable for large-scale applications, as evidenced by our experiments.

The remainder of this work describes the design and implementation of our system as well as reports on the performance of a number of programs compiled and executed using PICCO.

2. RELATED WORK

There are a number of existing tools and compilers for secure two- and multi-party computation, which are relevant in the context of this work. They include Fairplay [35], FairplayMP [8], Sharemind [10], VIFF [19], SEPIA [14], TASTY [28], and a secure two-party compiler for (a subset of) ANSI C [29]. While many of them were designed to support a rather general types of computation (normally on integers) and some have attractive properties, due to efficiency and flexibility goals that we target to achieve we choose to build our own compiler instead of directly extending one of the existing tools. In particular, the features that we find crucial are: (i) support for multi-party computation, (ii) support for a variable number of participants (including separation between computational, input, and output parties), (iii) the use of standard linear secret sharing techniques, (iv) applicability of techniques for malicious adversaries, (v) support for arithmetic of desired precision, and (vi) efficiency. None of the above mentioned tools simultaneously achieve the desired properties (or can be modified to achieve them at a relatively small cost).

In more detail, Fairplay [35] was the first compiler for secure two-party computation based on evaluation of garbled Boolean circuits. It allows a user to specify a function to be securely evaluated in a high-level language SFDL, compile it, and run the corresponding protocol. The circuit optimization part of compilation is known to have large memory requirements [9]. FairplayMP [8] extends Fairplay with multi-party functionality. It uses several interactive operations on secret-shared values for the evaluation of each Boolean gate, resulting in a performance disadvantage compared to multi-party solutions that directly build on a linear secret sharing scheme. TASTY [28] is a recent tool that combines two-party garbled circuit evaluation with computation based on homomorphic encryption. The desired functionality is specified in a custom language TASTYL, which is a subset of python. Another recent two-party compiler [37] was developed with the goal of extensibility, which would permit different techniques to be used in a single protocol. The most recent two-party compiler [29] allows a program written in ANSI C to be compiled into a garbled circuit. The current limitations of it include no coverage of real numbers, pointers, or variable-length numeric data types.

Sharemind [10] is the most developed and efficient tool to date with a built-in support of integer arithmetic, arrays, and matrices. It has a number of operations written in the assembly language, which results in an efficient implementation. A C-like language SecreC was developed for specifying the desired computation [31]. Unfortunately, Sharemind uses non-standard secret-sharing techniques with three parties, which do not automatically extend to other numbers of computational parties. In addition, the use of non-standard arithmetic makes many existing techniques inapplicable, including techniques for security against malicious adversaries. Sharemind also supports only integers of fixed 32-bit length.

SEPIA [14] is a library for secure distributed processing of network data based on linear secret sharing techniques. It has an explicit separation between computational parties and the parties who contribute inputs and obtain outputs. It was developed for the purposes of privacy-preserving intrusion detection and has features inherent to this application which limit its applicability to a wider range of collaborative computation.

Tool/compiler	No. parties	Basic technique	Varying precision	Non-int arithmetic	Type of parallelism
Fairplay	2	GC	$\surd$		N/A
C compiler [29]	2	GC		$\surd$	N/A
TASTY	2	GC & HE	$\surd$		N/A
FairplayMP	≥ 3	GC & SS	$\surd$		N/A
Sharemind	3	additive SS		$\surd$	arrays
VIFF	≥ 3	linear SS	$\surd$		each interactive op
This work	≥ 3	linear SS	$\surd$	$\surd$	loops, arrays, and user-specified

Table 1: Summary of related work.

VIFF [19] is a compiler based on standard multi-party linear secret sharing techniques. It was designed to be suitable, and has support, for security in presence of malicious parties and uses specially designed for this purpose asynchronous techniques. VIFF tries to parallelize as many operations as possible, with each interactive operation implemented as a callback. This means that all operations are scheduled when a protocol starts, but each operation is executed when its inputs become available. Unfortunately, the choice of the programming language in which it is implemented and the way parallelism is handled (with a heavy-weight thread per elementary operation) makes it unnecessarily slow in practice.

Lastly, the compiler described in [32] is complementary to our work, and if computational parties coincide with output parties, it can be used to optimize performance of user-specified programs.

The features of the available compilers and tools are summarized in Table 1. Parallel execution optimizations are not applicable to some tools where the computation can always be realized in a constant number of rounds. In the table, GC stands for garbled circuits, HE for homomorphic encryption, and SS for secret sharing.

3. FRAMEWORK

In this work, we utilize a threshold linear secret sharing scheme for representation of and secure computation over private values. We choose to concentrate on this setting due to its flexibility and speed. Throughout this work, we thus use the multi-party setting in which $n > 2$ parties securely evaluate a function on private inputs and output the result to the recipients of the computation. We divide all participants into three groups: The input parties distribute their inputs in the form of shares to the computational parties prior to the computation. The computational parties carry out the computation on secret-shared data. Upon completion of the computation, they communicate their shares of the result to the output parties, who reconstruct the values and learn the result. Note that there are no constraints on how these three groups are formed, and a single entity can be involved in a protocol taking on one or more of the above roles. This formulation of secure computation is flexible enough to naturally fit several broad categories of collaborative and individual computing needs. In particular, a number of parties with a private input each can engage in secure function evaluation themselves and learn the result (or their respective results). Alternatively, they can choose a subset of them, a number of outside parties, or a combination of the above to carry out the computation, while each input owner distributes her private data to the parties who carry out the computation. Another very important scenario consists of a single entity outsourcing its computation to a number of computational nodes. In this case, the data owner will be the only input and output

party. Finally, the setup also allows two parties with private inputs to seek help of one or more additional servers and proceed with secure evaluation of their function using multi-party techniques.

We refer to computational parties as $P_1, \ldots, P_n$ and assume that they are connected by secure authenticated channels with each other. Each input and output party also establishes secure channels with P_1 through P_n. With a (n, t)-secret sharing scheme, any private value is secret-shared among n parties such that any $t + 1$ shares can be used to reconstruct the secret, while t or fewer parties cannot learn any information about the shared value, i.e., it is perfectly protected in information-theoretic sense. Therefore, the value of n and t should be chosen such that an adversary is unable to corrupt more than t computational parties (for instance, data owners can acquire nodes located at different cloud service providers).

In a linear secret sharing scheme, a linear combination of secret-shared values can be performed by each computational party locally, without any interaction, but multiplication of secret-shared values requires communication between all of them. In particular, we utilize Shamir secret sharing scheme [38], in which a secret value s is represented by a random polynomial of degree t with the free coefficient set to s. Each share of s corresponds to the evaluation of the polynomial on a unique non-zero point. All operations are performed in a field $\mathbb{F}$ (normally $\mathbb{Z}_p$ for a small prime p larger than any value that needs to be represented). Then given $t + 1$ or more shares, the parties can reconstruct the polynomial using Lagrange interpolation and learn s. Possession of t or fewer shares, however, information-theoretically reveals no information about s. With this representation, any linear combination of secret-shared values is computed locally by each party using its shares, while multiplication involves multiplying shares locally then applying interactive re-sharing and interpolation operations to reduce the degree of the resulting polynomial from $2t$ to t. This places a restriction on the value of t and we require that $t < n/2$. Multiplication is used as the basic building block that requires interaction. More generally, a multivariate polynomial of degree k can be evaluated by the computational parties using a single interaction when $kt < n$. In our case, by assuming $t < n/2$ the parties can compute multiplication or evaluate any multivariate polynomial of degree 2 using a single interaction. We implement multiplication as given in [26], where each party transmits $n-1$ messages resulting in $O(n^2)$ overall communication. More advanced techniques (such as [22]) allow for linear communication complexity per multiplication and can be employed to optimize communication.

A number of operations that we utilize for arithmetic use pseudo-random secret sharing (PRSS) [18], which allows the computational parties to agree on shares of a random value without communication. We refer the reader to [18] for additional details.

Throughout this work we assume that the participants are semi-honest (also known as honest-but-curious or passive), in that they perform the computation as prescribed, but might attempt to learn additional information from the messages that they receive. Security in presence of semi-honest adversaries is commonly defined by assuming ideal functionality, in which the computation is performed by a trusted third party, and showing that the real execution of a protocol does not reveal any additional information about private data than in the ideal model. In our setting security means that the combined view of any coalition of t or less computational parties can be simulated without access to the private data. We anticipate that PICCO will often be used in the setting when the computational parties do not contribute any input and do not receive any output (i.e., the case of secure computation outsourcing).

We use existing efficient protocols for computation on integer and floating point values that have previously been shown secure in our security model. Furthermore, by the composition theorem [15], these building blocks can be combined together to achieve security of the overall computation. Because we do not introduce any new cryptographic protocols but rather build an automated tool for their composition, we omit formal security definitions.

Performance of secure computation protocols is of a paramount importance for their practical use, and in our framework performance of a protocol is normally measured in terms of two parameters: (i) the number of interactive operations (multiplications, distributing shares of a secret, or opening a secret-shared value) necessary to perform the computation and (ii) the number of sequential interactions, i.e., rounds. Our goal is to minimize both of those parameters and in particular bring the round complexity as close as possible to the optimal complexity that can be achieved in a custom solution with the fastest available building blocks.

One interesting component of this work to be explored is the use of parallelism in the compiled functionalities. Our goal is to avoid too restrictive and too general use of parallelism found in prior work and instead incorporate enough flexibility that will provide fast runtime of compiled programs. That is, in frameworks similar to ours, Sharemind [10] provides parallelism only at the level of arrays, where multiple instances of an operation applied to each cell of the array are performed in a batch. This is insufficient for reducing the round complexity of many types of computation to a practical level. In VIFF [19], on the other hand, each interactive operation is allocated its own thread and is scheduled at the program initiation, but is executed when its inputs become available. Managing a large number of threads (many of which are not necessary) slows down the execution. We propose to provide parallelism which is still restrictive, but general enough for efficient execution.

We build on the knowledge of distributed systems compilers such as OMPi [23] to achieve efficiency in compilation and execution.

4. COMPILER DESIGN

4.1 Overview

We choose to implement our compiler as a source-to-source translator. It is designed to take a program written in an extension of C, where data to be protected are marked as private, and transform it into a C program that implements SMC. The transformed program can then be compiled by each computational party using a native compiler into an executable code. The C language was chosen due to its popularity and, more importantly, performance reasons. By translating a user program into the source code of its secure implementation and using a native compiler to produce binary code, we can ensure that our compiler can be used on any platform and we are able to take advantage of optimizations available in native compilers. Figure 1(a) shows the compilation process of a user-specified program. Besides the user program itself, our compiler takes as its input a configuration file which contains parameters associated with the setup of secure computation that do not change with each program. These parameters include the specification of the secret sharing scheme (the values n and t and optional modulus or its size in bits), and the number of input and output parties. The compiler determines the optimal modulus size for secret sharing based on the program content. After the user translates her program into C code corresponding to its secure implementation, the user supplies the transformed program to each computational party. Each computational party locally compiles the code and is ready to start the execution upon the receipt of the input data.

When the user would like to initiate secure computation, the user produces shares of the input data and sends each share to the respective computational party. The execution process is shown in

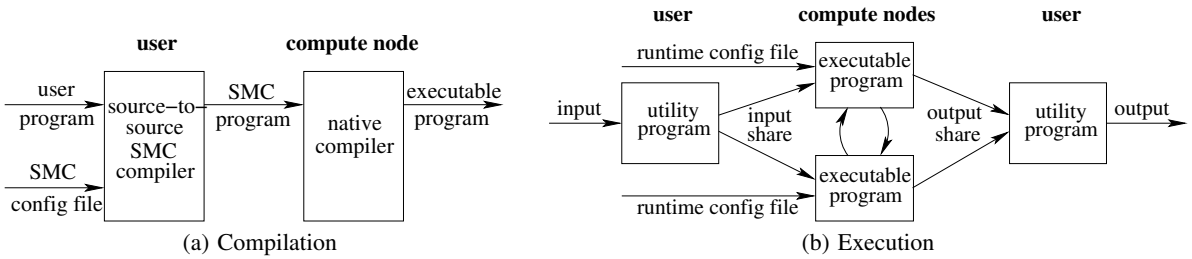

Figure 1: Compilation and execution of secure computation using PICCO.

Figure 1(b), where small utility programs are used for producing shares of the input and assembling the output from the produced shares. The runtime configuration file specifies information about the runtime setup and in particular contains IP addresses of computational parties and their public keys for the purpose of establishing secure channels. Once the user transmits shares of her input variables to the computational parties, the parties can start the computation. During the execution, the computational parties read shares of input data from user transmission and produce shares of output data. Upon computation completion, each computation party transmits shares of output variables to the user, who reconstructs the values using the utility program. This means that the client inputs data during an offline stage and is not an active participant of the computation. The compiled program is instructed to read input variables from user transmission according to the variable types.

The figure illustrates the case when a single user is the only input provider and output receiver. When, however, the computation is performed on behalf of a number of parties, they must jointly perform source-to-source translation of their target computation and separately supply shares of their respective inputs to the computational parties. In that case, each input variable in the program is annotated with the id of the input party supplying it and each output variable with the id of the party receiving it. This means that during the computation the compiled program will read an input variable from the transmission of the corresponding input party and write shares of an output variable to the file destined to the output party as specified in the program.

4.2 Specification of user programs

A program that specifies a functionality for secure collaborative computation or outsourcing is written in an extension of C. We extend C with support for private data types and operations on them. Public variables are handled without protection and our source-to-source compiler does not modify computation on them. For that reason, in the description that follows we treat private values and the interaction between private and public values.

Private and public variable qualifiers. A programmer can declare a variable to be either public or private using standard data types. To ensure that no information is accidentally leaked about a private variable, a variable will default to a private data type. For example, declarations `private int x` and `int x` will both create a private variable `x` of integer type.

Private data types. Because the length of the representation of numeric data types has a large influence on performance of secure computation, we allow lengths of private variables to be configurable, with set default values to aid usability. That is, the programmer can specify the length of numeric data types in bits and use the default value when the bitlength is not set. For example, if private integers are declared to be of type `int<20>`, we can set the field size to be of the minimum value necessary for computing with 20-bit integers. Support for custom-length data types is one of the features that distinguishes this work from some other compilers and SMC implementations such as [10] and [29].

For integer arithmetic, users can define private variables to be of type `char`, `short`, `int`, `long`, `long long`, and custom x-bit `int<`x`>`. The bitlength of standard data types is set according to the platform on which the program is being translated into secure implementation, but both standard-size and custom-size data types are implemented as integers with their bitlength recorded in the translated code.

To realize private floating point arithmetic, we use representation (and corresponding operations) from [7]. In particular, each floating point number is represented as a 4-tuple $\langle v, p, s, z\rangle$, where v is an ℓ-bit significand, p is a k-bit exponent, and s and z are sign and zero bits, respectively. The user can specify standard types `float`, `double`, and `long double` as well as custom-length `float<`x`,`y`>` with x-bit significand and y-bit exponent.

Operations on private data types. We provide support for many operations on private integer and floating point numbers. In particular, we implement integer addition +, subtraction −, multiplication $\star$, division /, left and right shifts « and », bitwise XOR ^, AND & and OR |, comparisons and equality tests. We also support floating point addition, subtraction, multiplication, division, and comparisons. The set of supported operations can be easily extended with additional operations and conversions between selected data types.

With the current implementation, users can perform operations on strings using arrays of one-byte integers `char`, but in the future we plan to provide built-in functions for string manipulation.

Built-in I/O functions. The programmer can specify variables that should be read from the input using special function `smcinput`. The function takes two arguments: the name of the variable to read and the id of the input party from whom the variable (or shares of the variable) come. For example, specifying `smcinput(x,1)` for private integer `x` will instruct the compiled program to read shares of an integer number from the transmission of input party 1. For private floating point numbers, shares of four elements of the floating point representation will be read. Because the input parties supply their data offline, `smcinput` is used for both private and public variables which during execution are read from the transmitted file of the appropriate input party.

Similarly, `smcoutput` is used to specify output variables and the output party who is to learn each of them. That is, including `smcoutput(x,1)` for private `x` will instruct the compiled program to output shares of `x` into a file which at the end of the computation is sent to output party 1. Both `smcinput` and `smcoutput` can be called anywhere in the program.

Conditional statements and loop constructs. All conditions in conditional statements can be private, which will allow if-statements to be evaluated obliviously. The only exception is that in loops

the termination condition must either be public or be evaluated privately, after which its result is revealed to the parties carrying out the computation. In other words, the function being evaluated must be known at the run time. To the best of our knowledge, private conditions are not supported in most secure computation tools and in particular are not available in any SMC software based on secret sharing (i.e., Sharemind or VIFF).

Support for concurrent execution. One of the most important design decisions that determines performance of compiled secure protocols is how parallelism is achieved. Because latency of interactive operations dominates performance of multi-party protocols in this setting, recent literature concentrated on techniques that lower round complexity of operations. For that reason, it is important to be able to execute several interactive operations in parallel.

A number of existing tools that adopt the same or similar frameworks have a variable degree of support for parallel execution of operations. For example, Sharemind executes all instances of an operation defined for each element of a (one- or two-dimensional) array in a batch. SEPIA runs a number of operations in a batch in each time slot (called round in [14]) using an application-specific implementation of parallelism. VIFF provides the most flexible way of parallel execution, in which an interactive operation is initiated as soon as its inputs are available. Experience with VIFF, however, shows that the cost of managing a large number of threads (many of which are often unnecessary) slows down protocol execution [25].

We believe that to achieve efficient execution of a broad range of functionalities, the available options for parallelizing the execution should not be either too narrow or unnecessarily too flexible. That is, the parallelism offered by Sharemind and SEPIA will allow for efficient execution of many functionalities, but can be very limiting for other applications and operations. The callback mechanism of VIFF, on the other hand, results in large overheads which can often be avoided. For these reasons, we support parallel execution of parts of a program in the ways defined next. Our goal is to provide enough flexibility to achieve very efficient performance for a broad range of programs while avoiding unnecessary slowdown due to managing threads for every elementary operation.

1. We support parallel execution of loop iterations, which generalizes Sharemind's and SEPIA's batch processing of identical operations. Because many programs require sequential execution of loop iterations, we leave it to the programmer to decide whether parallel or sequential execution should take place. The distinction is made using different syntax in the code. The conventional syntax, e.g., `for (statement; condition; statement) {statement; ...}` is used for sequential execution, and modified syntax, e.g., `for (statement; condition; statement) [statement; ...]` is used to execute all iterations of the loop in parallel.
2. For the programs in which execution of different code in parallel is possible and desirable (and parallel execution cannot be specified using a loop construct), we will offer a mechanism to specify the code that should be executed in parallel. We will use similar syntax with code enclosed in `[]` brackets for these purposes. For example, construction `[statement1;] [statement2;]` indicates that the two statements can be executed concurrently.

The use of loops with concurrent execution of loop iterations dictates the rule that the conditions affecting the number of loop iterations (i.e., the loop terminating conditions) are not allowed to be modified within the loop iterations. This is due to the obvious fact that the loop iterations must be scheduled without executing their contents and thus the number of loop iterations should be known or computable without executing instructions contained in the loop iterations. This implies that only for-loops can be specified to use simultaneous execution of loop iterations.

Array operations. As an additional mechanism for improving performance of distributed computation, we provide the ability for the programmer to specify operations on private vectors (arrays). For example, the user can specify expression `A*B` on private arrays `A` and `B` of the same size, which will lead to element-wise multiplication of `A` and `B`, i.e., multiplication of each element of `A` to the corresponding element of `B`. Besides the operations on private data types listed above, we support the inner product computation on two vectors using syntax `A@B`, which can be particularly efficiently implemented. Also, the I/O functions `smcinput` and `smcoutput` can be used with array variables to input/output a block of values at a time, using the third argument to specify the number of elements. For example, `smcinput(A,1,100)` reads 100 values into array `A` from the data of party 1.

We also provide a way of using portions of multi-dimensional arrays in (one-dimensional) array operations. This is performed by specifying the index in each dimension except the last one. For example, if matrix `A` is declared to be a two-dimensional array, notation `A[`i`]` will refer to the ith row of `A` (or elements `A[`i`][`j`]` for all j). This can, for instance, lead to an efficient implementation of matrix multiplications and other operations.

Enforcement of secure data flow. To ensure that no information about private values is leaked during program execution, we place certain restrictions on the data flow that involves both private and public data. In particular, statements that assign an expression that contains private values to a public variable are not allowed. This includes the case when the expression consists of function evaluation with a private return data type. The opposite, however, is true: a private variable can be assigned a public value, which is subsequently converted to shares.

In circumstances when the user intentionally wants to reveal certain information about a private value, we provide a built-in function `smcopen` that reveals the content of a private variable. For example, if the computation requires that the sign of private `x` is to be publicly revealed, the user can write

```
if (x >= 0) a = 1;
b = smcopen(a);
```

with private `a` and public `b`. This provides a mechanism for intentional declassification of information about private values and ensures that information is not accidentally leaked in an incorrectly written assignment.

Another restriction is related to enforcing secure data flow when conditional statements with a private condition are used. In particular, assignments to public variables within the scope of such statements are not allowed. This is due to the fact that by observing the value of a public variable after executing a conditional statement with a private condition, the result of the condition evaluation can be learned, which leaks information about private values. The same applies to functions called from the inside of a conditional statement with a private condition: the function must have no public side effects. This in particular means that the function is not allowed to modify global or shared public variables or public arguments passed by reference. This can be generalized to any instructions or computation with observable action, e.g., jump statements, which may leak information about the private condition.

Limitations and future work. Our current implementation does not provide support for pointers in user programs besides the use of arrays (note that the size of allocated memory for an array can

depend on prior computation). This limitation is also present in all other tools and compilers including ANSI C-based [29]. As our goal is to build as a general-purpose tool as possible, adding support for pointers is our most immediate goal for future work.

Another future work direction consists of adding resilience to malicious behavior. A number of existing techniques can be directly applied to our framework, e.g., asynchronous implementation in [19]. As part of future work we plan to evaluate performance of compiled programs in malicious security model.

4.3 Processing of user programs

When our compiler receives a user-specified program, it parses it and builds an abstract syntax tree (AST), which is consequently used to produce a modified program that implements secure distributed computation. To create a grammar for our extension of C, we build on existing source-to-source translators that parse C code. In particular, we build on an open-source OpenMP compiler OMPi [23]. We use flex [3] and bison [1] for defining tokens, specifying the context-free grammar, and building the AST.

Besides the AST, during parsing the compiler also builds and maintains a symbol table of defined variables and functions together with their types. Once the input program is parsed, the compiler performs transformations on the AST if necessary and outputs a modified program. All computation on public values is left unchanged, while all operations that use private values will be transformed into secure computation using GMP library [5] for large-precision arithmetic.[1]

Program transformations. The first transformation that the compiler performs on the input program is that of locating the main function within the user's code and renaming it into function called `old_main`. The main function of the transformed program contains necessary declarations for secure distributed computation and consequently calls the user's original main `old_main`.

Each private variable in the original user's program is transformed into one or more GMP large-precision variables of type `mpz_t` for storing its share, and all operations on private variables are rewritten to execute on GMP variables using the corresponding SMC algorithms. For example, declaration of a private integer variable `x` will result in declaring `x` of type `mpz_t` and initializing it using a call to `mpz_init()`. A private floating point variable is transformed into a structure that contains four variables of type `mpz_t`.

Now, because each operation on private variables needs to be modified to reflect operating on secret shares and call the corresponding secure distributed operations, evaluation of expressions on private variables introduces temporary variables in the modified program. For example, expression `x+y*z` on private integer `x`, `y`, and `z` will be transformed into statements `mpz_mul(tmp,y,z);` `mpz_add(tmp,x,tmp);` based on the precedence order of operators in the grammar. Our compiler is optimized to use the minimal number of temporary variables that ensure correct execution of the original program.

Another transformation that we perform is changing arguments of functions with private return values. Due to the implementation specifics of the `mpz_t` data type, functions cannot return variables of type `mpz_t`. For that reason, we change the arguments of all user-declared functions with private return values to include an extra argument passed by the reference which corresponds to the return value of the function and the return type is set to void. This does not restrict the way functions can be specified by the programmer and is transparent to the users.

[1]We note that the choice of a large number arithmetic library is not detrimental to the operation of the compiler.

Variable initialization. If a private variable is initialized to a constant (e.g., as in `int a=5`), the implementation needs to convert the constant to its shares. To achieve this, we create instructions for the computational parties in the transformed program to generate random shares of 0 (using a pseudo-random sharing of zero PRZS from [18]) and add the constant to each share. This will result in properly distributed shares of the constant, but it means that the value of the constant will be known to the computational parties. We choose to treat these private initialization constants as public because it complies with the intuition that the program's source code should not be treated as private. Furthermore, in many circumstances the constants to which private variables are initialized are indeed public, which simplifies their processing. For example, if a private variable is used in a computation of summation of private values, the fact that it is initialized to 0 does not need to be protected. In the event that a private variable needs to be initialized to a private constant, it should be input using a call to `smcinput`.

Handling of program input and output. Recall that input and output parties are not active participants in the computation. Thus, when a variable is specified to be read from the input using the `smcinput` function, its value will come from one of the input parties prior to the computation. Similarly, the content of every variable used in a call to `smcoutput` is stored by the computational parties until the end of the computation. At that point all values recorded for output party i will be transmitted to that party by all computational nodes. Because `smcinput` and `smcoutput` functions are the only available mechanism for I/O operations, we use the same interface for both public and private variables.

When a call to `smcinput` or `smcoutput` with arguments var and i is observed in the user program, the compiler looks up the type of variable var in the symbol table that stores all declared variables. The compiler than replaces a call to `smcinput` in the transformed program with instructions for reading data of the appropriate type from the input of party i and placing it in variable var. Similarly, a call to `smcoutput` is replaced with instructions for writing the content of var according to its type to the output destined to output party i. The type of variable var determines how many fields are used to represent the variable and their lengths. For example, a private floating point variable is represented by four random field $\mathbb{F}$ elements (secret shares), while a public integer is represented by a single value of its declared size.

Because a user program might be processed using multi-threaded implementation, the compiler instructs the transformed program to read the input of all parties into a data structure (that also maintains variable names) before the computation starts. The data structure allows each thread to access the variables it reads or writes from the memory without having to implement I/O synchronization.

Implementation of numeric operations. Numeric operations on private values constitute a significant portion of our implementation of secure distributed computation, and we briefly describe how these operations are implemented. Integer addition, subtraction, and multiplication are basic building blocks for computation on secret shares. Integer division is implemented according to [7], right shift $\ll x$ is implemented as multiplication by 2^x, and left shift is implemented using truncation as in [16]. Integer bitwise operations are implemented using bit decomposition according to [17] followed by the corresponding bitwise computation using arithmetic operations. That is, for each bit i of operands $a = a_1 \ldots a_\ell$ and $b = b_1 \ldots b_\ell$, XOR is implemented as $a_i \oplus b_i = a_i + b_i - 2a_ib_i$, AND is implemented as $a_i \wedge b_i = a_ib_i$, and OR is implemented as $a_i \vee b_i = a_i + b_i - a_ib_i$. Comparisons and equality tests are implemented according to [16]. All operations can be used on both signed and unsigned integers. The complexities of divi-

sion, bit decomposition, comparisons, and equality tests are linear in the bitlength of the operands, and the complexity of truncation is linear in the number of bits being truncated. All floating point operations (addition, subtractions, multiplication, and division) are implemented according to [7].

Implementation of array variables. To be able to support direct operations on arrays of private data types using the traditional syntax (e.g., infix notation for binary operators), the compiler must know the size of the arrays. While the size of an array generally cannot be determined in C due to the use of pointers, in our C extension arrays of private elements are created through a controlled interface. This allows us to internally represent private arrays after program parsing as a structure consisting of a field that stores the size of the array and a field with the array elements themselves. The size is then used to rewrite operations on private arrays into secure distributed computation, where all elements of the array are processed together. For example, multiplying two private arrays of size n will lead to executing n element-wise multiplications in one round of computation. This means that all operations are processed using the round complexity of only one operation, but in some cases even greater computational savings are possible. For example, inner product computation can be realized using only a single interactive operation regardless of the size of the arrays. Recall that our setting allows us to evaluate any multivariate polynomial of degree 2 using a single interaction between the computational parties which results in a particularly efficient inner product protocol.

Note that, while our current implementation does not support pointers besides arrays, using a structure to represent a private array can be made compatible with implementations with full support for pointers. In particular, we can make a pointer to a private array to point to the second field of the structure that stores array elements to achieve data compatibility.

Memory access at private locations. When the user-specified code contains access to an element of an array at a private location, we protect the location information by touching all elements of the array.[2] Private indexing is implemented as a multiplexer, where we first privately bit-decompose the index and then evaluate the multiplexer according to its Boolean formula. AND gates (multiplications) are computed for each element of the array in parallel, after which the result is locally added together by each computational node. NOT operation of bit b is implemented as $1 - b$. The result of this operation is always private regardless of the type of the data stored in the array.

Handling of private data types in assignments. To ensure that information about private values is not accidentally leaked, the compiler checks all assignment statements and produces a terminal error if a private expression is being assigned to a public variable. This is enforced using the AST, in which each node is marked as public or private and private status propagates from a child to its parent. Then a node that corresponds to assignment is not permitted to be composed of a public variable and private expression.

This check covers the cases when the expression contains function calls and the return type of at least one function used in the expression is known to be private. If, however, a function call is used, but its return type is not known (i.e., function declaration cannot be found), the compiler displays a warning of a potential violation of secure data flow (i.e., unknown data type and a potential information leakage). The goal of these checks is to help the programmer avoid unintentional information leakage without restricting the functionality. Public variables can be assigned values based on private data through `smcopen` calls.

If, on the other hand, a private variable is assigned a public expression, the compiler rewrites the code to convert the result of the expression evaluation to secret shares and then assign the resulting private value to the variable.

Handling of conditional statements. As mentioned earlier, if-statements with private conditions are not allowed to contain observable public actions in their body to prevent information leakage about private conditions. To enforce this constraint, the compiler analyzes all statements in the body of the conditional statement (recursively parsing all constructions) to detect violations and produce a terminal error is a violation is found. Similarly, the compiler extracts the list of functions called from the body of a conditional statement with a private condition. For each such function, if its content is known, the compiler analyzes its body for public side effects (such as changes to public global or shared variables) and produces a terminal error if such side effects are present. If, however, the content of the function is not known, the compiler produces a warning indicating a possible information leakage. Note that private side effects in functions called from the body of an if-statement with a private condition are permitted.

If no terminal errors have been found, the compiler proceeds with transforming the if-statements with private conditions. As the first step, the compiler determines all variables, the values of which are modified within the body of the if-statement (by searching through the list of statements for assignment operations), and their values are preserved in temporary variables. Next, the body is evaluated as if the condition was true and the condition of the if-statement is also privately evaluated. Now all variables affected by the instructions in the body of the if-statement are updated as if the condition holds and we need to roll back to their original values if the condition does not hold. To accomplish this, we update each affected variable v by setting its value to $c \cdot v + (1 - c)v_{orig}$, where c is a private bit corresponding to the result of evaluating the condition and v_{orig} is the original value of v prior to executing the body of the if-statement.

If the if-statement contains the else clause, we repeat the process for all variables modified within the body of the else clause with two important differences: the private condition no longer needs to be evaluated, and each variable v affected by the body of the else clause is updated to $(1 - c)v + c \cdot v_{orig}$ after executing all instructions in the else clause. This approach allows us to support arbitrary constructions while using the minimal amount of resources.

The above approach allows us to evaluate private conditional statements that modify ordinary variables efficiently, but can lead to suboptimal memory use when the variables are arrays. In particular, if the body of an if-statement modifies a single element of the array (or a small number of them), storing the original content of the entire array may result in substantial amount of extra memory. For example, in constructions of the type

```
for (i=0; i<n; i++)
  if (a[i]<0) a[i]=-a[i];
```

with private array `a`, there is no need to make a copy of the entire array `a` for each conditional statement. Similarly, if a function receives an index into a global array as its parameter and conditionally modified the element at that index, storing a copy of the array is wasteful. To mitigate the problem, we make special provisions for array handling. In particular, if an element of an array is modified within the body of a conditional statement with private condition and that assignment is not surrounded by a loop, the element of the array is treated as an ordinary variable, i.e., only a copy

[2]For large-sized arrays or databases, alternative techniques such as oblivious RAM [27] or the approach from [33] are likely to result in faster performance and their use will be investigated as part of this project.

of the element is made, not a copy of the array. If, on the other hand, the assignment is within the scope of a loop that affects the element's index within the array, that assignment will correspond to modifications to a number of array elements. In that case, we create a single temporary variable and in each loop iteration store the current value of the array element to be modified prior to executing the assignment operation. This will ensure that the optimal amount of memory is used by the program. For example, if the original program contains code

```
if (t>0)
  for (i=0; i<n; i+=5)
    a[i]=a[i]+1;
```

with private `t` and `a`, it will be transformed into:

```
mpz_t cond1;
mpz_t tmp1;
smc_gt(t,0,cond1);
for (i=0; i<n; i+=5) {
  tmp1=a[i];
  a[i]=a[i]+1;
  a[i]=cond1*a[i]+(1-cond1)*tmp1;
}
```

where the above code is a simplification of the actual code produced and `smc_gt` corresponds to secure distributed implementation of the greater than operation that stores the result of comparing its first two arguments into the third argument.

Handling of parallel constructs. Recall that we support two ways for the programmer to specify that portions of the code can be processed concurrently: (1) parallel loop iterations and (2) arbitrary parts of code that can be processed simultaneously. While both constructions use similar syntax, their implementations differ. In particular, a number of identical operations from different loop iterations can be efficiently processed in a batch (using the same number of rounds as that of a single loop iteration), with the overall load partitioned among the available cores via threads. In the second case, however, the only available mechanism for parallelizing the computation is by executing the portions of the code that can be concurrently processed in different threads. Threads can also be used within a single operation (not visible to the programmer) to provide the fastest execution of that operation.

In what follows, we first describe how we handle batch execution used for parallel loop constructs only, and then address the use of threads, which are used for concurrent execution of both loop iterations and arbitrary code.

Batch execution. When the compiler observes a loop construction with parallel iterations indicated using square brackets, it transforms the instructions contained in loop iterations using batch execution. As the first step, the transformed program will need to determine the number of loop iterations that should be scheduled for batch execution. For that reason, the compiler places loop specification code (without the loop's body) in the transformed program, so that the number of loop iterations can be computed at run time. In the case when loops with parallel iterations are nested in the user program, all iterations can still be processed using a single batch.

A loop construction, all iterations of which can be executed simultaneously, must be written in such a way that its iterations update independent memory locations. This can be achieved using arrays and similar data structures. Each iteration, therefore, can consist of statements that operate on array indices computed using arbitrary expressions. For example, a program can contain:

```
for (i=1; i<n; i*=3) [
  a[i]=b[i*i]*c[i+2];
  c[i-1]=b[i+1]*k;
]
```

where `a`, `b`, and `c` are arrays consisting of private elements and `k` is a private value. Our compiler handles this by creating code in the transformed program that allocates an integer array for storing array indices used in each expression or assignment. This array contains indices for each array operand and the array being updated (when applicable). By doing this, we can pass only necessary array elements used in the operation to its batch execution. This process is repeated for all operations within the statement. When a batch operation is executed, the transformed program uses the computed index array together with the start address of each array operand and the array being updated to perform computation. For example, batch execution of the first statement in the loop above will pass the address of arrays `a`, `b`, and `c` together with index array consisting of indices i for all i specified in the loop, indices $i \cdot i$, and indices $i+2$ to the multiplication operation. Execution of the second statement does not use array indices associated with the second operand.

When the body of a loop with parallel execution contains a private condition, the transformed program needs to store a copy of the variables being modified, evaluate the private condition in a batch, and compute the value of each variable v as $c \cdot v + (1-c)v_{orig}$, where c is the result of condition evaluation. Unlike regular loops, where we could store the original value of array elements one at a time right before the modification, the use of batch operations requires that we store all necessary data before the operation.

Thread management. To ensure that the number of threads is limited and the overhead associated with their management does not become overwhelming, a transformed program that uses concurrent execution is written to maintain a thread pool with provisions to maximize CPU utilization and avoid deadlocks (when a task assigned to a thread is blocked waiting on a child task and the child task is unable to acquire a thread). For loops with parallelizable iterations, we divide all iterations in batches of a predefined size, and place the next batch in a queue to be picked up by an available thread. This will ensure that the context switching overhead is very low relative to the overall computation time even for loops with a very large number of iterations. Given a specific setup, the optimal number of threads in the thread pool and the optimal batch size can be determined experimentally.

Our thread management mechanism is similar to that used in OMPi [23] and is realized as follows. Each user program starts by running in a single thread, and we refer to the code that can be scheduled to run in a separate thread as a task. Each task maintains a data structure (called task node) that stores information about its environment including local (i.e., visible only to the running thread) and shared (i.e., shared by all running threads) variables necessary for its execution. Furthermore, each thread from the thread pool manages a queue data structure (called task queue) that stores task nodes corresponding to the tasks whose execution is pending. The queue structure maintained by each thread has a fixed upper bound, which means that it can store only a certain number of tasks to execute. When a new task is being created, the thread that creates it will normally store it in its task queue, although there is a mechanism to indicate that the task execution must start immediately. When the task queue is full, but a new task is received, the thread will suspend the task it was running (which spawned the new task), and execute the newly received task to its completion. This ensures that no tasks are ever lost due to queue overflow. Lastly, to minimize the overall idle time for threads, the compiler applies the work-stealing strategy [24], which allows an idle thread to execute a task from another thread's queue.

The thread management mechanism on which we build did not require a thread to communicate with other nodes and thus had no provisions for a thread to retrieve only its own communication

off the network. For that reason, if execution of a compiled program uses multiple threads, at the point of execution where multiple threads are created, we also create a manager thread that handles incoming communication in behalf of all other threads. In particular, the manager thread listens to the sockets corresponding to all peer nodes with which the current node communicates and reads the data once it becomes available. After retrieving the data from a specific socket for a specific thread (each transmission is now marked with thread id), the manager thread signals the receiving thread that it now has data available. This ensures that each thread correctly receives data destined to it. Sending the data, on the other hand, is handled by each thread as before, after securing a lock for mutual exclusion on the socket that the thread wants to use.

To the best of our knowledge, no implementation other than VIFF supports multi-threaded execution and this topic has not been sufficiently treated in the SMC literature.

Modulus computation. The programmer can specify the size of the modulus or the modulus itself[3] to be used for secure computation. If that information is not provided, the compiler computes the minimum modulus size necessary for correct and secure operation. The size is computed based on the bitlength of the declared variables and operations on them. The algorithm proceeds by computing the maximum bitlength of all declared variables and maximum bitlength necessary for carrying out the specified operations. For example, for integer data types, the algorithm computes the maximum of all declared bitlengths ℓ_i, $\max(\ell_1, \ell_2) + \kappa$ for each comparison operation on arguments of bitlengths ℓ_1 and ℓ_2, where κ is the statistical security parameter, $\ell_1 + \kappa$ for each right shift operation on an argument of bitlength ℓ_1, and $2\max(\ell_1, \ell_2) + \kappa + 1$ for each division on arguments of bitlengths ℓ_1 and ℓ_2.

Once the modulus size is determined, our compiler chooses the modulus of the appropriate size and includes it in the transformed user program. The compiler also outputs the modulus to a runtime configuration file for use with other components of the system.

4.4 Supplemental programs

While our source-to-source translator is the main component of PICCO, the system also contains two utility programs for private input preprocessing and private output assembly as shown in Figure 1(b). Here we briefly describe their functionality.

As mentioned earlier, each input and output variable is marked with the id of the input party supplying or receiving it, and the computational parties receive data from each input party prior to the computation start. To aid the input parties with creating their input transmissions, we instruct the compiler to produce a supplemental configuration file (not shown in Figure 1) in addition to the transformed user program itself. The configuration file contains the modulus necessary for share manipulation and a list of input and output variables (in the order of appearance of `smcinput` and `smcoutput` function calls) annotated with their input/output type, data type, and the id of the responsible input or output party.

At runtime, every input party will provide this configuration file to the input utility program. The program will prompt the user for the id and will ask to input data marked with that user's id in the configuration file. Data entry can be done either manually or from a file. The utility program consequently converts private inputs into their shares while leaving public inputs unchanged and produces n output files. The input party communicates the ith produced file to computational party P_i.

[3]When the user chooses the modulus for secure computation, the modulus has to be some prime $p > n$ for the computation to be carried in the field $\mathbb{Z}_p$ or be equal to 256 to indicate that the computation is in the field GF(2^8).

The output utility program has a similar functionality: Upon receipt of output transmissions from at least $t + 1$ computational parties, an output party will input the supplemental configuration file together with the user id into the program. The output utility program consequently reconstructs user data according to the data types and outputs the result to the user.

5. PERFORMANCE EVALUATION

In this section we provide experimental results using PICCO. Following [29], we create a number of user programs of various functionalities, compile and run them in the distributed setting, and measure their performance. To fully evaluate the effect of parallel execution (using array operations, batching, and threads), for each tested functionality we provide its basic version as well as optimized version that uses concurrent execution features of PICCO. The types of computations tested as part of this work include:

1. A mix of arithmetic operations consisting of 90% additions and 10% multiplications with 10 input and 5 output variables. The basic functionality sequentially executes all operations, while the optimized version uses batch operations.
2. Multiplication of two matrices. The basic and optimized user programs are given in Figures 2 and 3, respectively. For compactness of presentation, we assume that the second matrix is given as the transpose of the matrix to be multiplied. Computing the transpose uses only local memory operations and does not increase the overall runtime.
3. Mergesort computation that outputs the median of the sorted array. The programs are given in Figures 4 and 5.
4. Computation of the Hamming distance between two binary vectors. The programs are given in Figures 6 and 7.
5. Evaluation of one block of AES on a private 128-bit message using private 128-bit key.
6. Computation of the edit distance between two strings of equal size via dynamic programming. The cost of insertion, deletion, and substitution is fixed and set to 1.
7. Matching of two fingerprints consisting of an equal number of minutiae. Each minutia point consists of x and y coordinates in a two-dimensional space, as well as its orientation.

The source code of the last three programs is omitted, but will be reported in the full version. Instead, we give a brief description. For compactness of presentation of other programs, we use array operations with input/output functions even in unoptimized programs, which introduces only a marginal improvement on the runtime of basic programs. All of the programs included in the experiments use integer arithmetic, which facilitates comparison of their performance with the same functionalities produced using other tools. Performance of our implementation of floating point operations, however, can be found in [7], including performance of executing multiple instances of an operation in a batch.

The results of the experiments for $n = 3$ computational parties are given in Table 2. We consider this value of n to be commonly used in practice. Because the number of computational parties is independent of the number of input owners and output recipients, in cases of both single client computation outsourcing and secure collaborative computation it is possible to find three independent computational parties which are unlikely to collude with each other. The programs our compiler produces are written in C/C++ using the GMP [5] library for large number arithmetic, the Boost libraries [2] for communication, and OpenSSL [6] for securing the computation. Our LAN experiments were conducted using 2.4 GHz 6-core machines running Red Hat Linux and connected through 1 Gb/s Ethernet. Our WAN experiments used machines from the GENI infrastructure [4], where two of the machines were

Experiment	Modulus p length (bits)	Basic functionality		Optimized functionality		Sharemind		Two-party compiler [29]	
		LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)
100 arithmetic operations	33	1.40	315	**0.18**	**31.6**	71	203	1,198	1,831
1000 arithmetic operations	33	13.4	3,149	**0.60**	**32.3**	82	249	3,429	5,823
3000 arithmetic operations	33	42.7	9,444	**1.60**	**34.5**	127	325	10,774	11,979
5×5 matrix multiplication	33	17.7	3,936	**0.27**	**31.6**	132	264	3,419	5,244
8×8 matrix multiplication	33	67.8	16,126	**0.45**	**32.1**	168	376	18,853	21,843
20×20 matrix multiplication	33	1,062	251,913	**2.41**	**35.7**	1,715	2,961	N/A	N/A
Median, mergesort, 32 elements	81	703.7	98,678	**256.7**	6,288	7,115	22,208	4,450	**5,906**
Median, mergesort, 64 elements	81	1,970	276,277	**649.6**	**12,080**	15,145	47,636	N/A	N/A
Median mergesort, 256 elements	81	13,458	1,894,420	**3,689**	**47,654**	66,023	203,044	N/A	N/A
Median mergesort, 1024 elements	81	86,765	–	**20,579**	**170,872**	317,692	869,582	N/A	N/A
Hamming distance, 160 bits	9	21.2	5,038	**0.17**	**31.1**	72	188	793	816
Hamming distance, 320 bits	10	42.3	10,092	**0.22**	**31.3**	102	203	850	1,238
Hamming distance, 800 bits	11	105.8	25,205	**0.35**	**31.5**	117	254	933	989
Hamming distance, 1600 bits	12	212.7	50,816	**0.57**	**31.8**	132	284	1,037	1,265
AES, 128-bit key and block	8	319.1	75,874	**35.1**	**3,179**	652 [34]	N/A	N/A	N/A
Edit distance, 100 elements	57	48,431	9,479,330	**4,258**	**116,632**	69,980	214,286	N/A	N/A
Edit distance, 200 elements	57	201,077	–	**16,038**	**432,456**	196,198	498,831	N/A	N/A
Fingerprint matching, 20 minutiae	66	3,256	541,656	**830**	**74,704**	24,273	75,820	N/A	N/A
Fingerprint matching, 40 minutiae	66	13,053	2,140,630	**2,761**	172,455	55,088	**172,266**	N/A	N/A

Table 2: Performance of representative programs using PICCO.

```
public int main() {
   public int i, j, k, S;
   smcinput(S, 1, 1);
   int A[S][S], B[S][S], C[S][S];
   smcinput(A, 1, S*S);
   smcinput(B, 1, S*S);

   for (i = 0; i < S; i++)
     for (j = 0; j < S; j++)
       C[i][j] = 0;

   for (i = 0; i < S; i++)
     for (j = 0; j < S; j++)
       for (k = 0; k < S; k++)
         C[i][j] += A[i][k]*B[k][j];

   smcoutput(C, 1, S*S);
   return 0;
}
```

Figure 2: Basic matrix multiplication program.

```
public int main() {
   public int i, j, S;
   smcinput(S, 1, 1);
   int A[S][S], B[S][S], C[S][S];

   smcinput(A, 1, S*S);
   //assume B is given as transpose of original B
   smcinput(B, 1, S*S);

   for (i = 0; i < S; i++) [
     for (j = 0; j < S; j++) [
        C[i][j] = A[i] @ B[j];
     ]
   ]
   smcoutput(C, 1, S*S);
   return 0;
}
```

Figure 3: Matrix multiplication program that uses concurrent computation.

located on the same LAN and one at a different location. All machines were running Fedora 15 with either 4 cores using 3.0 GHz or 8 cores using 2.4 GHz. The pairwise round-trip times between the machines were 0.2 msec, 63 msec, and 63 msec. Each experiment was run 5 times and the mean time over the runs and computational parties is reported in the table. We omit unoptimized experiments with excessive runtime. Also, batch execution in the optimized functionalities does not use threads as the number of operations in batch execution is not sufficiently large to exceed the threshold and be divided into threads.

As can be seen from Table 2, the running time of unoptimized programs for arithmetic operations, matrix multiplication, and Hamming distance grows linearly with the number of multiplications that they use. That is, performing multiplication of 5×5 matrices, computing 125-element Hamming distance, and performing 1250 arithmetic operations takes very similar amount of time for both LAN and WAN experiments. For these functionalities, combining multiple interactive operations into a single batch results in tremendous performance improvement. That is, using a loop with concurrent iterations in arithmetic operations program and using vector operations in the remaining functionalities, the round complexity of all of these programs reduces to 1. For programs and batches of small size (e.g., 10 multiplications in 100 arithmetic operations program), the overall time can hardly be distinguished from the time that a single operation takes. For computation of larger size, however, although round complexity does not change, both increased computation and communication contribute to the overall time. Consider, for example, optimized matrix multiplication results. In the LAN experiments, almost all of the overhead for 20×20 matrix multiplication (8,000 multiplications, but only 400 interactive operations) is due to computation. In the WAN experiments, however, most of the time is still contributed by communication (sending and receiving shares for one batch operation). The execution times for matrix multiplication can be compared to that of the Hamming distance, where the optimized functionality always invokes one multiplication operation regardless of the problem size. The Hamming distance results tell us how the computational effort increases as the problem size increases and communication remains the same.

To summarize, for these types of functionalities, the use of concurrent loop iterations and vector operations brings the round com-

```
public int K=32;
private int A[K];

void mergesort(public int l, public int r) {
   public int i, j, k, m, size;
   size = r - l + 1;
   int tmp;

   if (r > l) {
      m = (r + l)/2;
      mergesort(l, m);
      mergesort(m + 1, r);

      for (i = size >> 1; i > 0; i = i >> 1)
        for (j = 0; j < size; j += 2*i)
          for (k = j; k < j + i; k++)
            if (A[k+l] > A[k+i+l]) {
               tmp = A[k+l];
               A[k+l] = A[k+i+l];
               A[k+i+l] = tmp;
            }
   }
}

public int main() {
   public int i, median = K/2;
   smcinput(A, 1, K);
   mergesort(0, K-1);
   smcoutput(A[median], 1);
   return 0;
}
```

Figure 4: Basic mergesort median program.

plexity to the minimal single round and thus the performance is bounded by a one-way communication delay from below. In our experiments, the performance was improved by 2–3 orders of magnitude depending on the problem size and the savings will continue to increase for larger computational problems. The use of threads additionally improves the running time for problems of large size by a small factor that depends on the number of cores.

Mergesort computation is dominated by $O(m \log^2 m)$ comparisons for a set of size m and this growth is reflected in the runtimes of the unoptimized (sequential) functionality. The use of concurrent loop iterations greatly decreases the time that the merge step takes, but for this functionality the use of threads to concurrently sort the first half and the second half of the set also improves the performance. The round complexity of mergesort is $O(\log^2 m)$ comparisons, but we do not achieve it because the number of threads is limited to the number of cores. (Likewise, our Sharemind program written in a similar way does not achieve optimal round complexity.)

For the AES implementation, the best performance can be achieved when the computation is in GF(2^8) and thus the user needs to specify this field as a configuration parameter. Then the user secret-shares each byte of the key and message with the computational nodes. Recall that each round of AES consists of AddRoundKey, SubBytes, ShiftRows, and MixColumns algorithms and the most complex part of round key computation is the same as SubBytes. When using arithmetic in GF(2^8), all operations other than S-box computation in SubBytes and round key computation are linear and thus are performed non-interactively. Instead of following the traditional implementations of the S-box as a table lookup, we write the user program to compute the transformation directly, where the computation of the inverse and bit decomposition is implemented as suggested in [20]. One round of key expansion and one round of

```
void mergesort(public int l, public int r) {
   public int i, j, k, m, size;
   size = r - l + 1;
   int tmp[size];

   if (r > l) {
      m = (r + l)/2;
      [ mergesort(l, m); ]
      [ mergesort(m + 1, r); ]

      for (i = size >> 1; i > 0; i = i >> 1)
        for (j = 0; j < size; j += 2*i) [
          for (k = j; k < j + i; k++) [
            tmp[k] = A[k+l];
            if (A[k+l] > A[k+i+l]) {
               A[k+l] = A[k+i+l];
               A[k+i+l] = tmp[k];
            }
          ]
        ]
   }
}
```

Figure 5: Mergesort median program that uses concurrent computation (the difference from basic program).

```
public int main() {
   public int i, M;
   smcinput(M, 1, 1);
   private int<1> A[M], B[M];
   private int<10> dist = 0;

   smcinput(A, 1, M);
   smcinput(B, 1, M);
   for (i = 0; i < M; i++)
     dist += A[i] ^ B[i];

   smcoutput(dist, 1);
   return 0;
}
```

Figure 6: Basic Hamming distance program.

block cipher are computed simultaneously, where all bytes of the block are processed in parallel.

We obtain that the compiled program performs the key expansion and one block of AES encryption in only 35 msec, which is clearly very competitive even compared to custom implementations that do not use a compiler. In particular, in recent years several publications optimize evaluation of AES in the context of secure multi-party computation. The results range from circuit minimization techniques [12, 13] to concrete realizations in the two-party and multi-party settings. In the two-party setting, [36], [28], and [30] report 7 sec, 3.3 sec, and (without key expansion) 0.2 sec evaluation time, respectively, for AES using custom (as opposed to compiler-generated) circuits. In the multi-party setting, the results are available for 3 computational parties on a LAN. [20] reports 2 sec in a setting similar to ours, the fastest recent results in the Sharemind framework [34] are 652 msec, and, most notably, [33] reports 9 msec without key expansion using an implementation with 2 concurrent threads (and 14.3 msec using a single thread) based on a new lookup-table protocol of high efficiency. The last approach can be integrated into our compiler for achieving high-performance table lookups for AES and other purposes. Also, [21] reports runtime on the order of minutes (offline and online work without key expansion) in two stronger security models (while all other results cited above are in the semi-honest model).

The edit distance program uses dynamic programming to fill in the cells of a $(m + 1) \times (m + 1)$ grid, where m is the length of

```
public int main() {
   public int i, M;
   smcinput(M, 1, 1);
   private int<1> A[M], B[M];
   private int<10> dist;

   smcinput(A, 1, M);
   smcinput(B, 1, M);
   dist = A @ B;
   dist = -2*dist;
   for (i = 0; i < M; i++) {
      dist += A[i] + B[i];
   }
   smcoutput(dist, 1);
   return 0;
}
```

Figure 7: Hamming distance program that uses concurrent computation.

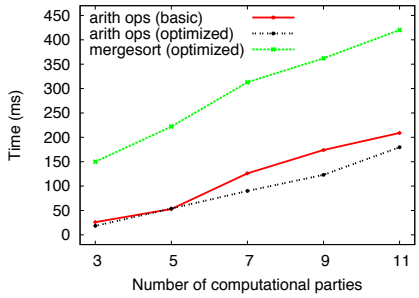

Figure 8: Performance of selected programs with a varying number of computational parties on a LAN.

the input strings. Each cell at position (i, j) is computed as the minimum of three distances computed from the distances at cells $(i-1, j)$, $(i, j-1)$, $(i-1, j-1)$ and the result of comparison of the characters at positions i and j. The optimized version of the program processes all cells on a diagonal in parallel, resulting in $2m-3$ rounds that compute the minimum function.

Lastly, the fingerprint program compares one minutia of fingerprint A to all minutiae of fingerprint B, and among those that matched (i.e., within close Euclidean distance and small angle) chooses the one with the closest distance. The chosen minutia in fingerprint B is marked as no longer available. The process is repeated with all other minutiae of fingerprint A, where all minutiae marked as no longer available are ignored in finding a matching. The output of the algorithm is the number of minutiae marked as having a mate in the other fingerprint. The optimized version of this program runs all distance and orientation comparisons for a minutia in fingerprint A in parallel. It then also computes the minimum and its location using a tree with $\log m$ rounds of comparisons for m-minutia fingerprints. Note that the computation cannot reveal if a minutia in B already has a mate or not close enough to a minutia in A, and thus for each minutia in A we have to treat all minutiae in B in the same way.

For comparison, we also provide performance of secure two-party computation compiler from [29] and Sharemind [10]. The former is a general-purpose compiler for ANSI C for the two-party setting and it is informative to compare performance of programs produced using the compiler. In Table 2 we list runtimes for similar programs when available as reported in [29]. The latter is a framework that allows a program written in language called SecreC to be compiled into secure distributed implementation for 3 computational parties. The tool is constantly evolving and well optimized, while the language currently has a few limitations such as no support for if-statements with a private condition[4]. We chose Sharemind for its speed, as other general secure multi-party computation tools and compilers such as FairplayMP and VIFF result in slower performance. We run Sharemind experiments in the same setups as our LAN and WAN experiments.

From Table 2, it is clear that despite using more computational parties and highly interactive techniques, the performance of programs compiled using PICCO compares very favorably to those compiled using the two-party approach of [29]. In particular, the largest difference we observe is by more than 3 orders of magnitude for some LAN experiments. Performance of the programs executed through Sharemind is also slower than with PICCO, especially for simple programs on a LAN. In our Sharemind experiments we were unable to match the performance reported in [11], despite extensive use of the available (batch) optimization mechanisms. Furthermore, we experience a large 4-fold reduction in execution time of arithmetic operations programs by replacing operations of the form $a = a + a$ with operations of the form $b = a + a$, where a and b are vectors. This could be in part due to excessive logging (which we were unable to disable) and is not the expected behavior. The execution of interactive multiplication operations is more costly and execution of local addition operations should take a small fraction of the overall time, regardless of how addition operations are handled in the user program. We conclude that Sharemind was optimized for massively-parallel computation with low amortized cost per operation, but not general-purpose computation where sequential execution is common. Furthermore, Sharemind is likely to outperform PICCO when a large number of homogeneous operations are to be run in parallel, but we expect our tool to result in lower execution time for general-purpose functionalities.

To investigate the impact of varying the number of computational parties on the performance, we conducted additional experiments. The programs that we chose are unoptimized 1000 arithmetic operations, optimized 1000 arithmetic operations, and 32-element mergesort, which provide insights into the performance of plain multiplications, batch multiplications, and combination of batch and multi-threaded implementation, respectively. To perform multiplication, each computational party transmits $n-1$ messages and performs Lagrange interpolation that in the most general form involves quadratic (in the number of parties) computation. On a LAN, this computation contributes a substantial portion of the overall time and [25] reports that the growth in multiplication time can be best described by function $f(n) = 0.009n^2 + 0.006n + 0.799$. We, however, observe that in the semi-honest setting all parties send their data as prescribed and the set of available shares is known. This means that Lagrange coefficients used during interpolation can be precomputed, resulting in linear computation time. Furthermore, we reconstruct only the free coefficient of the polynomial that encodes the secret as opposed to the entire polynomial. This gives us that the work a computational party performs for a single interactive operation is linear in n, but the total volume of communication placed on the network is quadratic in n. Our LAN results (using a new set of machines) are given in Figure 8, where optimized arithmetic operations' time is scaled up by a factor of 20. We can observe linear or slightly higher than linear growth in n.

[4]SecreC allows for comparisons and other basic predicates to be evaluated on private values outside of if-statements. This has similar expressiveness to using if-statements with arbitrary conditions over private variables, but results in degradation of usability, as the programmer has to expand and rewrite all if-statements by hand.

6. CONCLUSIONS

The goal of this is work is to enable secure execution of general-purpose programs in not fully trusted environments. Toward this goal, we introduce PICCO — a suite of programs for compiling a user program written in an extension of C with variables that need be protected marked as private, into its secure distributed implementation, and running it in a distributed setting. Our implementation uses techniques based on secret sharing with several optimizations to improve the runtime. This results in efficient secure execution suitable for sizeable computations in a variety of settings.

Acknowledgments

We are grateful to Ethan Blanton for valuable suggestions on the design of the compiler, Dan Bogdanov for the help with Sharemind experiments, and anonymous reviewers for their valuable feedback. This work was supported in part by grants CNS-1319090 and CNS-1223699 from the National Science Foundation and FA9550-13-1-0066 from the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies. We also acknowledge the NSF-sponsored Global Environment for Network Innovations (GENI) test bed, which allowed us to run WAN experiments.

7. REFERENCES

[1] Bison – GNU parser generator. http://www.gnu.org/software/bison.

[2] Boost C++ libraries. http://www.boost.org.

[3] flex: The Fast Lexical Analyzer. http://flex.sourceforge.net.

[4] GENI: Global environment for network innovations. http://www.geni.net.

[5] GMP – The GNU Multiple Precision Arithmetic Library. http://gmplib.org.

[6] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org.

[7] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *CCS*, 2008.

[9] M. Blanton. Empirical evaluation of secure two-party computation models. Technical Report TR 2005-58, CERIAS, Purdue University, 2005.

[10] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[11] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *IJIS*, 11(6):403–418, 2012.

[12] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Symposium on Experimental Algorithms*, 2010.

[13] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-box. In *Information Security and Privacy Research*, pages 287–298, 2012.

[14] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–240, 2010.

[15] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[16] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[17] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *FC*, pages 35–50, 2010.

[18] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, pages 342–362, 2005.

[19] I. Damgård, M. Geisler, and M. Krøigård. Asynchronous multiparty computation: Theory and implementation. In *PKC*, pages 160–179, 2009.

[20] I. Damgård and M. Keller. Secure multiparty AES. In *FC*, pages 367–374, 2010.

[21] I. Damgård, M. Keller, E. Larraia, C. Miles, and N. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. IACR Cryptology ePrint Archive Report 2012/262, 2012.

[22] I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

[23] V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A portable C compiler for OpenMP V.2.0. In *European Workshop on OpenMP (EWOMP)*, pages 5–11, 2003.

[24] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[25] M. Geisler. *Cryptographic protocols: Theory and implementation*. PhD thesis, Aarhus University, 2010.

[26] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[27] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[28] W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *CCS*, pages 451–462, 2010.

[29] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *CCS*, 2012.

[30] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[31] R. Jagomägis. SecreC: A privacy-aware programming language with applications in data mining. Master's thesis, University of Tartu, 2010.

[32] F. Kerschbaum. Automatically optimizing secure computation. In *CCS*, pages 703–714, 2011.

[33] J. Launchbury, I. Diatchki, T. DuBuisson, and A. Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *ICFP*, pages 189–200, 2012.

[34] S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. In *ACNS*, pages 84–101, 2013.

[35] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[36] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.

[37] A. Schroepfer, F. Kerschbaum, and G. Mueller. L1 – An intermediate language for mixed-protocol secure computation. In *COMPSAC*, pages 298–307, 2011.

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