PCF: A Portable Circuit Format For Scalable Two-Party Secure Computation

Ben Kreuter
Computer Science Dept.
U. Virginia

Benjamin Mood
Computer and Info. Science Dept.
U. Oregon

abhi shelat
Computer Science Dept.
U. Virginia

Kevin Butler
Computer and Info. Science Dept.
U. Oregon

Abstract

A secure computation protocol for a function $f(x,y)$ must leak no information about inputs x,y during its execution; thus it is imperative to compute the function f in a data-oblivious manner. Traditionally, this has been accomplished by compiling f into a boolean circuit. Previous approaches, however, have scaled poorly as the circuit size increases. We present a new approach to compiling such circuits that is substantially more efficient than prior work. Our approach is based on online circuit compression and lazy gate generation. We implemented an optimizing compiler for this new representation of circuits, and evaluated the use of this representation in two secure computation environments. Our evaluation demonstrates the utility of this approach, allowing us to scale secure computation beyond any previous system while requiring substantially less CPU time and disk space. In our largest test, we evaluate an RSA-1024 signature function with more than 42 billion gates, that was generated and optimized using our compiler. With our techniques, the bottleneck in secure computation lies with the cryptographic primitives, not the compilation or storage of circuits.

1 Introduction

Secure function evaluation (SFE) refers to several related cryptographic constructions for evaluating functions on unknown inputs. Typically, these constructions require an *oblivious* representation of the function being evaluated, which ensures that the control flow of the algorithm will not depend on its input; in the two party case, boolean circuits are most frequently seen. These oblivious representations are often large, with millions and in some cases billions of gates even for relatively simple functions, which has motivated the creation of software tools for producing such circuits. While there has been substantial work on the practicality of secure function evaluation, it was only recently that researchers began investigating the practicality of compiling such oblivious representations from high-level descriptions.

The work on generating boolean circuits for SFE has largely focused on two approaches. In one approach, a library for a general purpose programming language such as Java is created, with functions for emitting circuits [13, 20]. For convenience, these libraries typically include pre-built gadgets such as adders or multiplexers, which can be used to create more complete functions. The other approach is to write a compiler for a high level language, which computes and optimizes circuits based on a high level description of the functionality that may not explicitly state how the circuit should be organized [18, 21]. It has been shown in previous work that both of these approaches can scale up to circuits with at least hundreds of millions of gates on modern computer hardware, and in some cases even billions of gates [13, 18].

The approaches described above were limited in terms of their practical utility. Library-based approaches like HEKM [13] or VMCrypt [20] require users to understand the organization of the circuit description of their function, and were unable to apply any optimizations across modules. The Fairplay compiler [21] was unable to scale to circuits with only millions of gates, which excludes many interesting functions that have been investigated. The PALC system [23] was more resource efficient than Fairplay, but did not attempt to optimize functions, relying instead on precomputed optimizations of specific subcircuits. The KSS12 [18] system was able to apply some global optimizations, but had to unroll all loops and store the complete circuit description, which caused some functions to require days to compile. Additionally, the language used to describe circuits in the KSS12 system was brittle and difficult to use; for example, array index values could not be arbitrary functions of loop indices.

1.1 Our Approach

In this work, we demonstrate a new approach to compiling, optimizing, and storing circuits for SFE systems. At a high level, our approach is based on representing the function to be evaluated as a program that computes the circuit representation of the function, similar to the circuit library approaches described in previous work. Our compiler then optimizes this program with the goal of producing a smaller circuit. We refer to our circuit representation as the *Portable Circuit Format* (PCF).

When the SFE system is run, it uses our interpreter to load the PCF program and execute it. As the PCF program runs, it interacts with the SFE system, managing information about gates internally based on the responses from the SFE system itself. In our system, the circuit is ephemeral; it is not necessary to store the entire circuit, and wires will be deleted from memory once they are no longer required.

The key insight of our approach is that it is not necessary to unroll loops until the SFE protocol runs. While previous compilers discard the loop structure of the function, ours emits it as part of the control structure of the PCF program. Rather than dealing directly with wires, our system treats wire IDs as *memory addresses*; a wire is "deleted" by overwriting its location in memory. Loop termination conditions have only one constraint: they must not depend on any secret wire values. There is no upper bound on the number of loop iterations, and the programmer is responsible for ensuring that there are no infinite loops.

To summarize, we present the following contributions:

- A new compiler that has the same advantages as the circuit library approach
- A new representation of circuits that is more compact than previous representations which scales to arbitrary circuit sizes.
- A portable interpreter that can be used with different SFE execution systems regardless of the security model.

Our compiler is a *back end* that can read the bytecode emitted by a *front end*; thus our compiler allows *any* language to be used for SFE. Instead of focusing on global optimizations of boolean functions, our optimization strategy is based on using higher-level information from the bytecode itself, which we show to be more effective and less resource-intensive. We present comparisons of our compiler with previous work and show experimental results using our compiler in two complete SFE systems, one based on an updated version of the KSS12 system and one based on HEKM. In some of our test cases, our compiler produced circuits only 30% as large as previous compilers starting from the same source code. With the techniques presented in this work, we demonstrate that the RSA algorithm with a real-world key size and real-world security level can be compiled and run in a garbled circuit protocol using a typical desktop computer. To the best of our knowledge, the RSA-1024 circuit we tested is larger than any previous garbled circuit experiment, with more than 42 billion gates. We also present preliminary results of our system running on smartphones, using a modified version of the HEKM system.

For testing purposes, we used the LCC compiler [8] as a front-end to our system. A high-level view of our system, with the LCC front-end, is given in Figure 1.

The rest of this paper is organized as follows: Section 2 is a review of SFE and garbled circuits; Section 3 presents an overview of bytecode languages; Section 4 explains our compiler design and describes our representation; Section 5 discusses the possibility of using different bytecode and SFE systems; Section 6 details the experiments we performed to evaluate our system and results of those experiments; Section 7 details other work which is related to our own; and Section 8 presents future lines of research.

2 Secure Function Evaluation

The problem of secure two-party computation is to allow two mutually distrustful parties to compute a function of their two inputs without revealing their inputs to the opposing party (privacy) and with a guarantee that the output could not have been manipulated (correctness). Yao was the first to show that such a protocol can be constructed for any computable function, by using the *garbled circuits* technique [30]. In his original formulation, Yao proposed a system that would allow users to describe the function in a high level language, which would then be compiled into a circuit to be used in the garbled circuits protocol. The first complete implementation of this design was the Fairplay system given by Malkihi et al. [21].

Oblivious Transfer One of the key building blocks in Yao's protocol is *oblivious transfer*, a cryptographic primitive first proposed by Rabin [25]. In this primitive, the "sender" party holds a database of *n* strings, and the "receiver" party learns exactly *k* strings with the guarantee that the sender will not learn which *k* strings were sent and the receiver will not learn more than *k* strings; this is known as a *k*-out-of-*n* oblivious transfer. Given a public key encryption system it is possible to construct a 1-out-of-2 oblivious transfer protocol [7], which is the building block used in Yao's protocol.

Garbled Circuits The core of Yao's protocol is the construction of garbled circuits, which involves encrypting

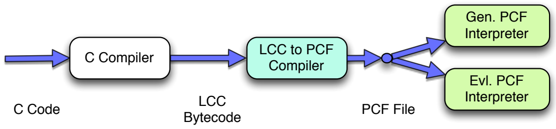

Figure 1: High-level design of our system. We take a C program and compile it down to the LCC bytecode. Our compiler then transforms the LCC bytecode to our new language PCF. Both parties then execute the protocol in their respective role in the SFE protocol. The interpreter could be any execution system.

the truth table of each gate in a circuit description of the function. When the protocol is run, the truth values in the circuit will be represented as decryption keys for some cipher, with each gate receiving a unique pair of keys for its output wire. The keys for a gate's input wires are then used to encrypt the keys for its output wires. Given a single key for each input wire of the circuit, the party that evaluates the circuit can decrypt a single key that represents a hidden truth value for each gate's output wire, until the output gates are reached. Since this encryption process can be applied to any circuit, and since any computable function has a corresponding circuit family, this allows the construction of a secure protocol for any computable function.

The typical garbled circuit protocol has two parties though it can be expanded to more. Those two parties are Bob, the generator of the garbled circuit, and Alice, the evaluator of the garbled circuit. Bob creates the garbled circuit and therefore knows the decryption keys, but does not know which specific keys Alice uses. Alice will receive the input keys from Bob using an oblivious transfer protocol, and thus learns only one key for each input wire; if the keys are generated independent of Bob's input, Alice will learn only enough to compute the output of the circuit.

Several variations on the Yao protocol have been published; a simple description of the garbling and evaluation process follows. Let $f : \{0,1\}^A \times \{0,1\}^B \rightarrow \{0,1\}^j \times \{0,1\}^k$ be a computable function, which will receive input bits from two parties and produce output bits for each party (not necessarily the same outputs). To garble the circuit, a block cipher $\langle E, D, G \rangle$ will be used.

For each wire in the circuit, Bob computes a pair of random keys $(k_0, k_1) \leftarrow (G(1^n), G(1^n))$, which represent logical 0 and 1 values. For each of Alice's outputs, Bob uses these keys to encrypt a 0 and a 1 and sends the pair of ciphertexts to Alice. Bob records the keys corresponding to his own outputs. The rest of the wires in the circuit are inputs to gates. For each gate, if the truth table is $[v_{0,0}, v_{0,1}, v_{1,0}, v_{1,1}]$, Bob computes the following ciphertext:

$$\begin{bmatrix} E_{k_{l,0}}(E_{k_{r,0}}(k_{v_{0,0}})), E_{k_{l,0}}(E_{k_{r,1}}(k_{v_{0,1}})) \\ E_{k_{l,1}}(E_{k_{r,0}}(k_{v_{1,0}})), E_{k_{l,1}}(E_{k_{r,1}}(k_{v_{1,1}})) \end{bmatrix}$$

where $k_{l,*}$ and $k_{r,*}$ are the keys for the left and right input wires (this can be generalized for gates with more than two inputs). The order of the four ciphertexts is then randomly permuted and sent to Alice.

Now that Alice has the garbled gates, she can begin evaluating the circuit. Bob will send Alice his input wire keys. Alice and Bob then use an oblivious transfer to give Alice the keys for her input wires. For each gate, Alice will only be able to decrypt one entry, and will receive one key for the gate's output, and will continue to decrypt truth table entries until the output wires have been computed. Alice will then send Bob his output keys, and decrypt her own outputs.

Optimizations Numerous optimizations to the basic Yao protocol have been published [10, 13, 17, 24, 27]. Of these, the most relevant to compiling circuits is the "free XOR trick" given by Kolesnikov and Schneider [17]. This technique allows XOR gates to be evaluated without the need to garble them, which greatly reduces the amount of data that must be transferred and the CPU time required for both the generator and the evaluator. One basic way to take advantage of this technique is to choose subcircuits with fewer non-XOR gates; Schneider published a list of XOR-optimal circuits for even three-input functions [27].

Huang et al. noted that there is no need for the evaluator to wait for the generator to garble all gates in the circuit [13]. Once a gate is garbled, it can be sent to the evaluator, allowing generation and evaluation to occur in parallel. This technique is very important for large circuits, which can quickly become too large to store in RAM [18]. Our approach unifies this technique with the use of an optimizing compiler.

3 Bytecode

A common approach to compiler design is to translate a high level language into a sequence of instructions for a simple, abstract machine architecture; this is known as the *intermediate representation* or *bytecode*. Bytecode representations have the advantage of being machine-independent, thus allowing a compiler front-end to be used for multiple target architectures. Optimizations performed on bytecode are machine independent as well; for example, dead code elimination is typically performed on bytecode, as removing dead code causes programs to run faster on all realistic machines.

For the purposes of this work, we focus on a commonly used bytecode abstraction, the *stack machine*. In this model, operands must be pushed onto an abstract stack, and operations involve popping operands off of the stack and pushing the result. In addition to the stack, a stack machine has RAM, which is accessed by instructions that pop an address off the stack. Instructions in a stack machine are partially ordered, and are divided into subroutines in which there is a total ordering. In addition to simple operations and operations that interact with RAM, a stack machine has operations that can modify the *program counter*, a pointer to the next instruction to be executed, either conditionally or unconditionally.

At a high level, our system translates bytecode programs for a stack machine into boolean circuits for SFE. At first glance, this would appear to be at least highly inefficient, if not impossible, because of the many ways such an input program could loop. We show, however, that imposing only a small set of restrictions on permissible sequences of instructions enables an efficient and practical translator, without significantly reducing the usability or expressive power of the high level language.

4 System Design

Our system divides the compiler into several stages, following a common compiler design. For testing, we used the LCC compiler front end to parse C source code and produce a bytecode intermediate representation (IR). Our back end performs optimizations and translates the bytecode into a description of a secure computation protocol using our new format. This representation greatly reduces the disk space requirements for large circuits compared to previous work, while still allowing optimizations to be done at the bit level. We wrote our compiler in Common Lisp, using the Steel Bank Common Lisp system.

4.1 Compact Representations of Boolean Circuits

In Fairplay and the systems that followed its design, the common pattern has been to represent Boolean circuits as adjacency lists, with each node in the graph being a gate. The introduces a scalability problem, as it requires storage proportional to the size of the circuit. Generating, optimizing, and storing circuits has been a bottleneck for previous compilers, even for relatively simple functions like RSA. Loading such large circuits into RAM is a challenge, as even very high-end machines may not have enough RAM for relatively simple functions.

There have been some approaches to addressing this scalability problem presented in previous work. The KSS12 system reduced the RAM required for protocol executions by assigning each gate's output wire a reference count, allowing the memory used for a wire value to be deallocated once the gate is no longer needed. However, the compiler bottleneck was not solved in KSS12, as even computing the reference count required memory proportional to the size of the circuit. Even with the engineering improvements presented by Kreuter, shelat, and Shen, the KSS12 compiler was unable to compile circuits with more than a few billion gates, and required several days to compile their largest test cases [18].

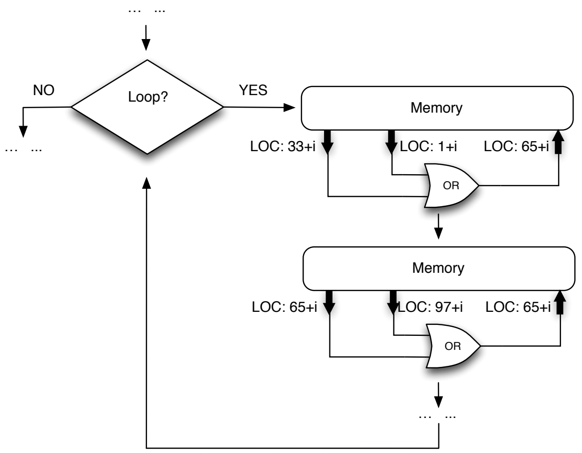

Figure 2: The high-level concept of the PCF design. It is not necessary to unroll loops at compile time, even to perform optimizations on the circuit. Instead, loops can be evaluated at runtime, with gates being computed on-the-fly, and loop indices being updated locally by each party. Wire values are stored in a table, with each gate specifying which two table entries should be used as inputs and where the output should be written; previous wire values in the table can be overwritten during this process, if they are no longer needed.

The PAL system [23] also addresses memory requirements, by adding control structures to the circuit description, allowing parts of the description to be re-used. In the original presentation of PAL, however, a large circuit file would still be emitted in the Fairplay format when the secure protocol was run. An extension of this work presented by Mood [22] allowed the PAL description to be used directly at runtime, but this work sacrificed the ability to optimize circuits automatically.

Our system builds upon the PAL and KSS12 systems to solve the memory scalability problem without sacrificing the ability to optimize circuits automatically. Two observations are key to our approach.

Our first observation is that it is possible to free the memory required for storing wire values without computing a reference count for the wire. In previous work, each wire in a circuit is assigned a unique global identifier, and gate input wires are specified in terms of these identifiers (output wires can be identified by the position of the gate in the gate list). Rather than using global identifiers, we observe that wire values are ephemeral, and only require a unique identity until their last use as the input to a gate.

We therefore maintain a table of "active" wire values, similar to KSS12, but change the gate description. In this format, wire values are identified by their index in the table, and gates specify the index of each input wire and an index for the output wire; in other words, a gate is a tuple $\langle t, i_1, i_2, o \rangle$, where t is a truth table, i_1, i_2 are the input wire indexes, and o is the output wire index. When a wire value is no longer needed, its index in the table can be safely used as an output wire for a gate.

Now, consider the following example of a circuit described in the above format, which accumulates the Boolean AND of seven wire values:

$\langle AND_1, 1, 2, 0 \rangle$
$\langle AND_2, 0, 3, 0 \rangle$
$\langle AND_3, 0, 4, 0 \rangle$
$\langle AND_4, 0, 5, 0 \rangle$
$\langle AND_5, 0, 6, 0 \rangle$
$\langle AND_6, 0, 7, 0 \rangle$

Our second observation is that circuits such as this can be described more compactly using a loop. This builds on our first observation, which allows wire values to be overwritten once they are no longer needed. A simple approach to allowing this would add a conditional branch operation to the description format. This is more general than the approach of the PAL, which includes loops but allows only simple iteration. Additionally, it is necessary to allow the loop index to be used to specify the input or output wire index of the gates; as a general solution, we add support for indirection, allowing wire values to be copied.

This representation of Boolean circuits is a bytecode for a one-bit CPU, where the operations are the 16 possible two-arity Boolean gates, a conditional branch, and indirect copy. In our system, we also add instructions for function calls (which need not be inlined at compile time) and handling the parties' inputs/outputs. When the secure protocol is run, a three-level logic is used for wire values: 0, 1, or $\perp$, where $\perp$ represents an "unknown" value that depends on one of the party's inputs. In the case of a Yao protocol, the $\perp$ value is represented by a garbled wire value. Conditional branches are not allowed to depend on $\perp$ values, and indirection operations use a separate table of pointers that cannot computed from $\perp$ values (if such an indirection operation is required, it must be translated into a large multiplexer, as in previous work).

We refer to our circuit representation as the *Portable Circuit Format* or PCF. In addition to gates and branches, PCF includes support for copying wires indirectly, a function call stack, data stacks, and setting function parameters. These additional operations do not emit any gates and can therefore be viewed as "free" operations. PCF is modeled after the concept of PAL, but instead of using predefined sub-circuits for complex operations, a PCF file defines the sub-circuits for a given function to allow for circuit structure optimization. PCF includes lower level control structures compared to PAL, which allows for more general loop structures.

In Appendix A, we describe in detail the semantics of the PCF instructions. Example PCF files are available at the authors' website.

4.2 Describing Functions for SFE

Most commonly used programming languages can describe processes that cannot be translated to SFE; for example, a program that does not terminate, or one which terminates after reading a specific input pattern. It is therefore necessary to impose some limitation on the descriptions of functions for SFE. In systems with domain specific languages, these limitations can be imposed by the grammar of the language, or can be enforced by taking advantage of particular features of the grammar. However, one goal of our system is to allow any programming language to be used to describe functionality for SFE, and so we cannot rely on the grammar of the language being used.

We make a compromise when it comes to restricting the inputs to our system. Unlike model checking systems [2], we impose no upper bound on loop iterations or on recursive function calls (other than the memory available for the call stack), and leave the responsibility of ensuring that programs terminate to the user. On the other hand, our system does forbid certain easily-detectable conditions that could result in infinite loops, such as unconditional backwards jumps, conditional backwards jumps that depend on input, and indirect function calls. Although false positives, i.e. programs that terminate but which contain such constructs are possible, our hypothesis is that useful functions and typical compilers would not result in such instruction sequences, and we observed no such functions in our experiments with LCC.

4.3 Algorithms for Translating Bytecode

Our compiler reads a bytecode representation of the function, which lacks the structure of higher-level descriptions and poses a unique challenge in circuit generation. As mentioned above, we do not impose any upper limit on loop iterations or the depth of the function call stack. Our approach to translation does not use any symbolic analysis of the function. Instead, we translate the

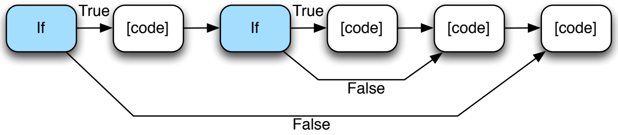

Figure 3: Nested if statements, which can be handled using the stack-based algorithm.

bytecode into PCF, using conditional branches and function calls as needed and translating other instructions into lists of gates. For testing, we use the IR from the LCC compiler, which is based on the common stack machine model; we will use examples of this IR to illustrate our design, but note that none of our techniques strictly require a stack machine model or any particular features of the LCC bytecode.

In our compiler, we divide bytecode instructions into three classes:

Normal Instructions which have exactly one successor and which can be represented by a simple circuit. Examples of such instructions are arithmetic and bitwise logic operations, operations that push data onto the stack or move data to memory, etc.

Jump Instructions that result in an unconditional control flow switch to a specific label. This does not include function calls, which we represent directly in PCF. Such instructions are usually used for if/else constructs or preceding the entry to a loop.

Conditional Instructions that result in control flow switching to either a label or the subsequent instruction, depending on the result of some conditional statement. Examples include arithmetic comparisons.

In the stack machine model, all operands and the results of operations are pushed onto a global stack. For "normal" instructions, the translation procedure is straightforward: the operands are popped off the stack and assigned temporary wires, the subcircuit for the operation is connected to these wires, and the output of the operation is pushed onto the stack. "Jump" instructions appear, at first, to be equally straightforward, but actually require special care as we describe below.

"Conditional" instructions present a challenge. Conditional jumps whose targets precede the jump are assumed to be loop constructs, and are translated directly into PCF branch instructions. All other conditional jumps require the creation of multiplexers in the circuit to deal with conditional assignments. Therefore, the branch targets must be tracked to ensure that the appropriate condition wires are used to control those multiplexers.

In the Fairplay and KSS12 compilers, the condition wire for an "if" statement is pushed onto a stack along with a "scope" that is used to track the values (wire assignments) of variables. When a conditional block is closed, the condition wire at the top of the stack is used to multiplex the value of all the variables in the scope at the top with the values from the scope second to the top, and then the stack is popped. This procedure relies on the grammar of "if/else" constructs, which ensures that conditional blocks can be arranged as a tree. An example of this type of "if/else" construct is in Figure 3. In a bytecode representation, however, it is possible for conditional blocks to "overlap" with each other without being nested.

In the sequence shown in Figure 4, the first branch's target *precedes* the second branch's target, and indirect loads and assignments exist in the overlapping region of these two branches. The control flow of such an overlap is given in Figure 5. A stack is no longer sufficient in this case, as the top of the stack will not correspond to the appropriate branch when the next branch target is encountered. Such instruction sequences are not uncommon in the code generated by production compilers, as they are a convenient way to generate code for "else" blocks and ternary operators.

To handle such sequences, we use an algorithm based on a priority queue rather than a stack, and we maintain a global condition wire that is modified as branches and branch targets are reached. When a branch instruction is reached, the global condition wire is updated by logically ANDing the branch condition with the global condition wire. The priority queue is updated with the branch condition and a scope, as in the stack-based algorithm; the priority is the target, with lower targets having higher priority. When an assignment is performed, the scope at the top of the priority queue is updated with the value being assigned, the location being assigned to, the old value, and a copy of the global condition wire. When a branch target is reached, multiplexers are emitted for each assignment recorded in the scope at the top of the priority queue, using the copy of the global condition wire that was recorded. After the multiplexers are emitted, the global condition wire is updated by ORing the *inverse* of the condition wire at the top of the priority queue, and then the top is removed.

Unconditional jumps are only allowed in the forward direction i.e. only if the jump precedes its target. When such instructions are encountered, they are translated into conditional branches whose condition wire is the inverse of the conjunction of the condition wires of all enclosing branches. In the case of a jump that is not in any conditional block, the condition wire is set to false; this does not necessarily mean that subsequent assignments will not occur, as the multiplexers for these assignments

```
EQU4 A
INDIRI4 16
EQU4 B
INDIRI4 24
LABELV A
ASGNI4
LABELV B
ASGNI4
```

Figure 4: A bytecode sequence where overlapping conditional blocks are not nested; note that the target of the first branch, “A,” precedes the target of the second branch, “B.”

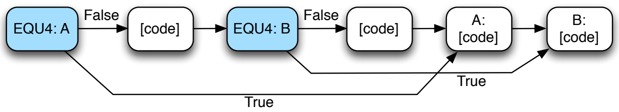

Figure 5: A control flow with overlapping conditional blocks.

will be emitted and will depend on a global control line that may be updated as part of a loop construct. The optimizer is responsible for determining whether such assignments can occur, and will rewrite the multiplexers as direct assignments when possible.

Finally, it is possible that the operand stack will have changed in the fall-through path of a conditional jump. In that case, the stack itself must be multiplexed. For simplicity, we require that the depth of the stack not change in a fall-through path. We did not observe any such changes to the stack in our experiments with LCC.

4.4 Optimization

One of the shortcomings of the KSS12 system was the amount of time and memory required to perform optimizations on the computed circuit. In our system, optimization is performed before loops are unrolled but after the functionality is translated into a PCF representation. This allows optimizations to be performed on a smaller representation, but increases the complexity of the optimization process somewhat.

The KSS12 compiler bases its optimization on a rudimentary dataflow analysis, but without any conditional branches or loops, and with single assignments to each wire. In our system, loops are not eliminated and wires may be overwritten, but conditional branches are eliminated. As in KSS12, we use an approach based on dataflow analysis, but we must make multiple passes to find a fixed point solution to the dataflow equations. Our dataflow equations take advantage of the logical rules of each gate, allowing more gates to be identified for elimination than the textbook equations identify.

We perform our dataflow analysis on individual PCF instructions, which allows us to remove single gates even where entire bytecode instructions could not be removed, but which carries the cost of somewhat longer compilation time, on the order of minutes for the experiments we ran. Currently, our framework only performs optimization within individual functions, without any interprocedural analysis. Compile times in our system can be reduced by splitting a large procedure into several smaller procedures.

Optimization	128 mult.	5x5 matrix	256 RSA
None	707,244	260,000	904,171,008
Const. Prop.	296,960	198,000	651,504,495
Dead Elim.	700,096	255,875	883,307,712
Both	260,073	131,875	573,156,735

Table 1: Effects of constant propagation and dead code elimination on circuit size, measured with simulator that performs no simplification rules. For each function, the number of non-XOR gates are given for all combinations of optimizations enabled.

4.4.1 Constant Propagation

The constant propagation framework we use is straightforward, similar to the methods used in typical compilers. However, for some gates, simplification rules can result in constants being computed even when the inputs to a gate are not constant; for example, XORing a variable with itself. The transfer function we use is augmented with a check against logic simplification rules to account for this situation, but remains monotonic and so convergence is still guaranteed.

4.4.2 Dead Gate Removal

The last step of our optimizer is to remove gates whose output wires are never used. This is a standard bit vector dataflow problem that requires little tailoring for our system. As is common in compilers, performing this step last yields the best results, as large numbers of gates become dead following earlier optimizations.

4.5 Externally-Defined Functions

Some functionality is difficult to describe well in bytecode formats. For example, the graph isomorphism experiment presented in Section 6 uses AES as a PRNG building block, but the best known description of the AES S-box is given at the bit-level [4], whereas the smallest width operation supported by LCC is a single byte. To compensate for this difficulty, we allow users to

Function	With	Without	Ratio
16384-bit Comp.	32,228	49,314	65%
128-bit Sum	345	508	67%
256-sit Sum	721	1,016	70%
1024-bit Sum	2,977	4,064	73%
128-bit Mult.	76,574	260,073	20%
256-bit Mult.	300,634	1,032,416	20%
1024-bit Mult.	8,301,962	19,209,120	21%

Table 2: Non-XOR gates in circuits computed by the interpreter with and without the application of simplification rules by the runtime system.

specify functions with the same language used internally to translate bytecode operations into circuits; an example of this language is shown in Section 5.1. This allows for possible combinations of our compiler with other circuit generation and optimization tools.

4.6 PCF Interpreter

To use a PCF description of a circuit in a secure protocol, an interpreter is needed. The interpreter simulates the execution of the PCF file for a single-bit machine, emitting gates as needed for the protocol. Loops are not explicitly unrolled; instead, PCF branch instructions are conditionally followed, based on the logic value of some wire, and each wire identifier is treated as an address in memory. This is where the requirement that loop bounds be independent of both parties' inputs is ultimately enforced: the interpreter cannot determine whether or not to take a branch if it cannot determine the condition wire's value.

For testing purposes, we wrote two PCF interpreters: one in C, which is packaged as a reusable library, and one in Java that was used for tests on smartphones. The C library can be used as a simulator or for full protocol execution. As a simulator it simply evaluates the PCF file without any garbling to measure the size of the circuit that would have been garbled in a real protocol. This interpreter was used for the LAN tests, using an updated version of the KSS12 protocol. The Java interpreter was incorporated into the HEKM system for the smartphone experiments, and can also be used in a simulator mode.

4.7 Threat Model

The PCF system treats the underlying secure computation protocol as a black box, without making any assumptions about the threat model. In Section 6, we present running times for smaller circuits in the malicious model version of the KSS12 protocol. This malicious model implementation simply invokes multiple copies of the same PCF interpreter used for the semi-honest version, one for each copy of the circuit needed in the protocol.

Function	With (s)	Without (s)
16384-bit Comp.	4.41 ± 0.3%	4.44 ± 0.3%
128-bit Sum	0.0581 ± 0.3%	0.060 ± 2%
256-bit Sum	0.103 ± 0.3%	0.105 ± 0.3%
1024-bit Sum	0.365 ± 0.3%	0.367 ± 0.2%
128-bit Mult.	0.892 ± 0.1%	0.894 ± 0.1%
256-bit Mult.	3.02 ± 0.1%	3.04 ± 0.1%
1024-bit Mult.	39.7 ± 0.2%	39.9 ± 0.06%

Table 3: Simulator time with simplification rules versus without, using the C interpreter. Times are averaged over 50 samples, with 95% confidence intervals, measured using the *time* function implemented by SBCL.

4.8 Runtime Optimization

Some optimizations cannot be performed without unrolling loops, and so we defer these optimizations until the PCF program is interpreted. As an example, logic simplification rules that eliminate gates whose output values depend on no more than one of their input wires can only be partially applied at compile time, as some potential applications of these rules might only be possible for some iterations of a loop. While it is possible to compute this information at compile time, in the general case this would involve storing information about each gate for every iteration of every loop, which would be as expensive as unrolling all loops at compile time.

A side-effect of applying such logic simplification rules is copy propagation. A gate that always takes on the same value as one of its inputs is equivalent to a copy operation. The application of logic simplification rules to such a gate results in the interpreter simply copying the value of the input wire to the output wire, without emitting any gate. As there is little overhead resulting from the application of simplification rules at runtime, we are able to reduce compile times further by not performing this optimization at compile time.

For each gate, the interpreter checks if the gate's value can be statically determined i.e. if its output value does not rely on either party's input bits. This is critical, as some of the gates in a PCF file are used for control flow, e.g. to increment a loop index. Additionally, logic simplification rules are applied where possible in the interpreter. This allows the interpreter to not emit gates that follow an input or which have static outputs even when their inputs cannot be statically determined. As shown in Table 2, we observed cases where up to 80% of the gates could be removed in this manner. Even in a simulator that performs no garbling, applying this runtime optimization not only shows no performance overhead, but actually a very slight performance gain, as shown in

Function	This Work	KSS12	HFKV
16384 Comp.	32,229	49,149	-
RSA 256	235,925,023	332,085,981	-
Hamming 160	880	-	3,003
Hamming 1600	9,625	-	30,318
3x3 Matrix	27,369	160,949	47,871
5x5 Matrix	127,225	746,177	221,625
8x8 Matrix	522,304	3,058,754	907,776
16x16 Matrix	4,186,368	24,502,530	7,262,208

Table 4: Comparisons between our compiler's output and the output of the KSS12 and Holzer et al. (HFKV) compilers, in terms of non-XOR gates.

Table 3. The slight performance gain is a result of the transfer of control that occurs when a gate is emitted, which has a small but non-trivial cost in the simulator. In a garbled circuit protocol, this cost would be even higher, because of the time spent garbling gates.

5 Portability

5.1 Portability Between Bytecodes

Our compiler can be given a description of how to translate bytecode instructions into boolean circuits using a special internal language. An example, for the LCC instruction "ADDU," is shown in Figure 6. The first line is specific to LCC, and would need to be modified for use with other front-ends. The second line assumes a stack machine model: this instruction reads two instructions from the stack. Following that is the body of the translation rule, which can be used in general to describe circuit components and how the input variables should be connected to those components.

```
(``ADDU'' nil second normal nil nil
  (two-stack-arg (x y) (var var)
  (chain [o1 = i1 + i2 + i3,
    o2 = i1 + (i1 + i2) * (i1 + i3)]
  (o2 -> i3
   x -> i1
   y -> i2
   o1 -> stack)
  (0 -> i3))))
```

Figure 6: Code used in our compiler to map the bytecode instruction for unsigned integer addition to the subcircuit for that operation.

The description follows an abstraction similar to VMCrypt, in which a unit gadget is "chained" to create a larger gadget. It is possible to create chains of chains e.g. for a shift-and-add multiplier as well. For more complex operations, Lisp source code can be embedded, which can interact directly with the compiler's internal data structures.

5.2 Portability Between SFE Systems

Both the PCF compiler and the interpreter can treat the underlying secure computation system as a black box. Switching between secure computation systems, therefore, requires work only at the "back end" of the interpreter, where gates are emitted. We envision two possible approaches to this, both of which we implemented for our tests:

1. A single function should be called when a gate should be used in the secure computation protocol. The Java implementation of PCF uses this approach, with the HEKM system.

2. Gates should be generated as if they are being read from a file, with the secure computation system calling a function. The secure computation system may need to provide "callback" functions to the PCF interpreter for copying protocol-specific data between wires. The C implementation we tested uses this abstraction for the KSS12 system.

6 Evaluation

We compiled a variety of functions to test our compiler, optimizer, and PCF interpreter. For each circuit, we tested the performance of the KSS12 system on a LAN, described below. For the KSS12 timings, we averaged the runtime for 50 runs, and alternating which computer acted as the generator and which as the evaluator to account for slight configuration differences between the systems. Compiler timings are based on 50 runs of the compiler, on a desktop PC with an Intel Xeon 5560 processor, 8GB of RAM, a 7200 RPM hard disk, Scientific Linux 6.3 (kernel version 2.6.32, SBCL version 1.0.38). Source code for our compiler, our test systems, and our test functions is available at the authors' website.

6.1 Effect of Array Sizes on Timing

Some changes in compile time can be observed as some of the functions grow larger. The dataflow analysis deals with certain pointer operations by traversing the entire local variable space of the function and all global memory, which in functions with large local arrays or programs with large global arrays is costly as it increases the number of wires that optimizer must analyze. Reducing this cost is an ongoing engineering effort.

Function	Total Gates	non-XOR Gates	Compile Time (s)	Simulator Time (s)
16384-bit Comp.	97,733	32,229	3.40 ± 4%	4.40 ± 0.2%
Hamming 160	4,368	880	9.81 ± 1%	0.0810 ± 0.3%
Hamming 1600	32,912	6,375	11.0 ± 0.4%	0.52 ± 8%
Hamming 16000	389,312	97,175	10.8 ± 0.2%	4.83 ± 0.5%
128-bit Sum	1,443	345	4.70 ± 3%	0.0433 ± 0.4%
256-bit Sum	2,951	721	4.60 ± 3%	0.0732 ± 0.4%
1024-bit Sum	11,999	2,977	4.60 ± 3%	0.250 ± 0.5%
64-bit Mult.	105,880	24,766	71.7 ± 0.2%	0.332 ± 0.4%
128-bit Mult.	423,064	100,250	74.9 ± 0.1%	0.903 ± 0.3%
256-bit Mult.	1,659,808	400,210	79.5 ± 0.9%	3.07 ± 0.2%
1024-bit Mult.	25,592,368	6,371,746	74.0 ± 0.2%	40.9 ± 0.4%
256-bit RSA	673,105,990	235,925,023	381. ± 0.2%	980. ± 0.3%
512-bit RSA	5,397,821,470	1,916,813,808	350. ± 0.2%	7,330 ± 0.2%
1024-bit RSA	42,151,698,718	15,149,856,895	564. ± 0.2%	56,000 ± 0.3%
3x3 Matrix Mult.	92,961	27,369	306. ± 1%	0.256 ± 0.5%
5x5 Matrix Mult.	433,475	127,225	343. ± 0.7%	0.94 ± 2%
8x8 Matrix Mult.	1,782,656	522,304	109. ± 0.1%	3.14 ± 0.3%
16x16 Matrix Mult.	14,308,864	4,186,368	109. ± 0.1%	23.7 ± 0.3%
4-Node Graph Iso.	482,391	97,819	684. ± 0.2%	3.63 ± 0.5%
16-Node Graph Iso.	10,908,749	4,112,135	1040 ± 0.1%	47.0 ± 0.1%

Table 5: Summary of circuit sizes for various functions and the time required to compile and interpret the PCF files in a protocol simulator. Times are averaged over 50 samples, with 95% confidence intervals, except for RSA-1024 simulator time, which is averaged over 8 samples. Run times were measured using the *time* function implemented in SBCL.

6.2 Experiments

We compiled and executed the circuits described below to evaluate our compiler and representation. Several of these circuits were tested in other systems, and present the non-XOR gate counts of the circuits generated by our compiler and other work in Table 4. The sizes, compile times, and interpreter times required for these circuits are listed in Table 5. By comparison, we show compile times and circuit sizes using the KSS12 and HFKV compilers in Table 6. As expected, the PCF compiler outperforms these previous compilers as the size of the circuits grow, due to the improved scalability of the system.

Arbitrary-Width Millionaire's Problem As a simple sanity check for our system, we tested an arbitrary-width function for the millionaire's problem; this can be viewed as a string comparison function on 32 bit characters. It outputs a 1 to the party which has the larger input. We found that for this simple function, our performance was only slightly better than the performance of the KSS12 compiler on the same circuit.

Matrix Multiplication To compare our system with the work of Holzer et al. [12], we duplicated some of their experiments, beginning with matrix multiplication on 32-bit integers. We found that our system performed favorably, particularly due to the optimizations our compiler and PCF interpreter perform. On average, our system generated circuits that are 60% smaller. We tested matrices of 3x3, 5x5, 8x8, and 16x16, with 32 bit integer elements.

Hamming Distance Here, we duplicate the Hamming distance experiment from Holzer et al. [12]. Again, we found that our system generated substantially smaller circuits. We tested input sizes of 160, 1600, and 16000 bits.

Integer Sum We implemented a basic arbitrary-width integer addition function, using ripple-carry addition. No array references are needed, and so our compiler easily handles this function even for very large input sizes. We tested input sizes of 128, 256, and 1024 bits.

Integer Multiplication Building on the integer addition function, we tested an integer multiplication function that uses the textbook shift-and-add algorithm. Unlike the integer sum and hamming distance functions, the multiplication function requires arrays for both input and output, which slows the compiler down as the problem size grows. We tested bit sizes of 64, 128, 256, and 1024.

RSA (Modular Exponentiation) In the KSS12 system [18], it was possible to compile an RSA circuit for toy problem sizes, and it took over 24 hours to compile a circuit for 256-bit RSA. This lengthy compile time and large memory requirement stems from the fact that all

	HFKV			KSS12		
Function	Total Gates	non-XOR gates	Time (s)	Total Gates	non-XOR gates	Time (s)
16384-bit Comp.	330,784	131,103	105. ± 0.1%	98,303	49,154	4.66 ± 0.5%
3x3 Matrix Mult.	172,315	47,871	2.2 ± 4%	424,748	160,949	10.5 ± 0.5%
5x5 Matrix Mult.	797,751	221,625	8.40 ± 0.3%	1,968,452	746,177	48.2 ± 0.2%
8x8 Matrix Mult.	3,267,585	907,776	59.4 ± 0.3%	8,067,458	3,058,754	210 ± 2%
16x16 Matrix Mult.	26,140,673	7,262,208	2,600 ± 7%	64,570,969	24,502,530	2,200 ± 1%
32-bit Mult.	65,121	26,624	6.43 ± 0.3%	15,935	5,983	0.55 ± 5%
64-bit Mult.	321,665	126,529	71.4 ± 0.3%	64,639	24,384	1.6 ± 2%
128-bit Mult.	1,409,025	546,182	999. ± 0.1%	260,351	97,663	6.10 ± 0.6%
256-bit Mult.	5,880,833	2,264,860	16,000 ± 2%	1,044,991	391,935	24.5 ± 0.2%
512-bit Mult.	-	-	-	4,187,135	1,570,303	105. ± 0.2%
1024-bit Mult.	-	-	-	16,763,518	6,286,335	430. ± 0.3%

Table 6: Times of HFKV and KSS12 compilers with circuit sizes. The Mult. program uses a Shift-Add implementation. All times are averaged over 50 samples with the exception of the HFKV 256-bit multiplication, which was run for 10 samles; times are given with 95% confidence intervals.

loops are unrolled before any optimization is performed, resulting in a very large intermediate representation to be analyzed. As a demonstration of the improvement our approach represents, we compiled not only toy RSA sizes, but also an RSA-1024 circuit, using only modest computational resources. We tested bit sizes of 256, 512, and 1024.

Graph Isomorpism We created a program, which allows two parties to jointly prove the zero knowledge proof of knowledge for graph isomorphism, first presented by Goldreich et al. [9]. In Goldreich et al.'s proof system, the prover has secret knowledge of an isomorphism between two graphs, g_1 and g_2. To prove this, the prover sends the verifier a random graph g_3 that is isomorphic to g_1 and g_2, and the verifier will then choose to learn either the $g_1 \rightarrow g_3$ isomorphism or the $g_2 \rightarrow g_3$ isomorphism. We modify this protocol so that Alice and Bob must jointly act as the prover; each is given shares of an isomorphism between graphs g_1 and g_2, and will use the online protocol to compute g_3 and shares of the two isomorphisms.

Our implementation works as follows: the program takes in XOR shares of the isomophism between g_1 and g_2 and a random seed from both participants. It also takes the adjacency matrix representation of g_1 as input by a single party. The program XORs the shares together to create the $g_1 \rightarrow g_2$ isomorphism. The program then creates a random isomorphism from $g_1 \rightarrow g_3$ using AES as the PRNG (to reduce the input sizes and thus the OT costs), which effectively also creates g_3.

Once the random isomorphism $g_1 \rightarrow g_3$ is created, the original isomorphism, $g_1 \rightarrow g_2$, is inverted to get an isomorphism from $g_2 \rightarrow g_1$. Then the two isomorphisms are "followed" in a chain to get the g_2 to g_3 isomorphism, i.e. for the ith instance in the isomorphic matrix, $iso_{2\rightarrow3}[i] = iso_{1\rightarrow3}[iso_{2\rightarrow1}[i]]$. The program outputs shares of both the isomorphism from g_1 to g_3 and g_2 to g_3 to both parties.

An adjacency matrix of g_3 is also an output for the party which input the adjacency matrix g_1. This is calculated by using the g_1 and the $g_1 \rightarrow g_3$ isomorphism.

6.3 Online Running Times

To test the online performance of our new format, we modified the KSS12 protocol to use the PCF interpreter. Two sets of tests were run: one between two computers with similar specifications on the University of Virginia LAN, a busy 100 megabit Ethernet network, and one between two smartphones communicating over a wifi network.

For the LAN experiments, we used two computers running ScientificLinux 6.3, a four core Intel Xeon E5506 2.13GHz CPU, and 8GB of RAM. No time limit on computation was imposed on these machines, so we were able to run the RSA-1024 circuit, which requires a little less than two days. To compensate for slight configuration differences between the two systems, we alternated between each machine acting as the generator and acting as the evaluator.

We give the results of this experiment in Table 7. We note that while the simulator times given in Table 5 are more than half the CPU time measured, they are also on par with the time spent waiting on the network. Nonblocking I/O or a background thread for the PCF interpreter may improve performance somewhat, which is an ongoing engineering task in our implementation.

6.4 Malicious Model Tests

The PCF system is not limited to the semihonest model. We give preliminary results in the malicious model ver-

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit Comp.	99.8 ± 0.2%	5.63 ± 0.6%	26.0 ± 0.6%	79.4 ± 0.2%
Hamming 1600	9.13 ± 0.4%	0.64 ± 4%	2.9 ± 4%	6.87 ± 2%
Hamming 16000	91.2 ± 0.2%	5.67 ± 0.7%	28. ± 3%	69. ± 2%
64-bit Mult.	0.749 ± 0.3%	0.158 ± 0.7%	0.409 ± 0.3%	0.494 ± 0.6%
128-bit Mult.	2.04 ± 0.3%	0.52 ± 1%	1.25 ± 0.2%	1.31 ± 0.6%
256-bit Mult.	5.74 ± 0.5%	1.2 ± 2%	4.2 ± 2%	2.7 ± 3%
1024-bit Mult.	72.7 ± 0.2%	28. ± 4%	60. ± 2%	40. ± 3%
256-bit RSA	1940 ± 0.2%	767. ± 0.7%	1620 ± 2%	1080 ± 3%
1024-bit RSA	1.2×10^5 ± 7%	4.7×10^4 ± 9%	9.5×10^4 ± 20%	7×10^4 ± 33%
3x3 Matrix Mult.	5.33 ± 0.4%	0.403 ± 0.6%	1.45 ± 0.8%	4.28 ± 0.6%
5x5 Matrix Mult.	24.4 ± 0.2%	1.81 ± 0.4%	6.75 ± 0.9%	19.5 ± 0.4%
8x8 Matrix Mult.	100. ± 0.2%	7.39 ± 0.4%	26.8 ± 0.7%	81.1 ± 0.3%
4-node ISO	10.1 ± 0.1%	1.05 ± 0.7%	4.96 ± 0.3%	6.15 ± 0.4%
16-node ISO	116. ± 0.2%	15.7 ± 0.6%	71.6 ± 0.3%	60.3 ± 0.6%

Table 7: Total running time, including PCF operations and protocol operations such as oblivious transfer, for online protocols using the PCF interpreter and the KSS12 two party computation system, on two computers communicating over the University of Virginia LAN. With the exception of RSA-1024, all times are averaged over 50 samples; RSA-1024 is averaged over three samples. Running time is divided into time spent on computation and time spent on network operations (including blocking).

sion of KSS12. These experiments were run on the same test systems as above, using two cores for each party. We present our results in Table 9. The increased running times are expected, as we used only two cores per party. In the case of 16384-bit comparison, the increase is very dramatic, due to the large amount of time spent on oblivious transfer (as both parties have long inputs).

6.5 Phone Execution

We created a PCF interpreter for use with the HEKM execution system and ported it to the Android environment. We then ran it on two Galaxy Nexus phones where one phone was the generator and another phone was the evaluator. These phones have dual core 1.2Ghz processors and were linked over Wi-Fi using an Apple Airport.

6.6 Phone Trials

As seen in Table 8, we were able to run the smaller programs directly on two phones. Since the interpreter executes slower on a phone and what would have taken a week of LAN trials would have taken years of phone time, we did not complete trials of the larger programs. Not all of the programs had output for the generator, allowing the generator to finish before the evaluator. This leads to a noticeable difference in total running time between the two parties.

Mood's work on designing SFE applications for mobile devices [22] found that allocation and deallocation was a bottleneck to circuit execution. This issue was addressed by substituting the standard *BigInteger* type for a custom class that reduced the amount of allocation required for numeric operations, resulting in a fourfold improvement in execution time. The lack of this optimization in our mobile phone experiments may contribute to the reduced performance that we observed.

In future work, we will port the C interpreter and KSS12 system to Android and run the experiment with that execution system. Since overhead appears to be tied to Android's Dalvik Virtual Machine (DVM), running programs natively should reduce overhead and hence reduce the performance differential between the phone and PC environments. Additionally, the KSS12 system uses more efficient cryptographic primitives, potentially further improving performance.

7 Related Work

Compiler approaches to secure two-party computation have attracted significant attention in recent years. The TASTY system presented by Henecka et al. [11] combines garbled circuit approaches with homomorphic encryption, and includes a compiler that emits circuits that can be used in both models. As with Fairplay and KSS12, TASTY requires functions to be described in a

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit Comp.	163. ± 0.5%	12. ± 3%	142. ± 0.5%	68. ± 1%
128-bit Sum	5.8 ± 8.2%	1. ± 30%	5.6 ± 8%	3. ± 20%
256-bit Sum	7.3 ± 5.0%	1. ± 30%	6. ± 5%	4. ± 20%
1024-bit Sum	16. ± 3.1%	2. ± 20%	16. ± 3%	6.4 ± 7%
64-bit Mult.	63.3 ± 0.5%	1. ± 10%	66.3 ± 0.6%	5. ± 10%
128-bit Mult.	257. ± 0.2%	3.8 ± 5%	280. ± 0.3%	12. ± 6%
3x3 Matrix Mult.	76.9 ± 0.4%	12. ± 2%	82.0 ± 0.5%	8.5 ± 4%
5x5 Matrix Mult.	352. ± 0.3%	49. ± 2%	371. ± 0.3%	32. ± 4%
8x8 Matrix Mult.	1,588. ± 0.1%	82. ± 3%	1,550. ± 0.1%	120. ± 1%

Table 8: Execution results from the phone interpreter using the HEKM execution system on two Galaxy Nexus phones. Times are averages of 50 samples, with 95% confidence intervals.

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit comp.	3900 ± 3%	76 ± 4%	2820 ± 2%	1200 ± 10%
128-bit sum	23. ± 2%	21. ± 2%	33.3 ± 0.5%	11.2 ± 0.2%
256-bit sum	63.0 ± 0.4%	10 ± 20%	49. ± 6%	27. ± 4%
1024-bit sum	260 ± 10%	16 ± 6%	187. ± 2%	100 ± 40%
128-bit mult.	192. ± 0.3%	47.2 ± 0.6%	168. ± 0.4%	70.1 ± 1%
256-bit mult.	637. ± 0.5%	160 ± 1%	577. ± 0.3%	210 ± 2%

Table 9: Online running time in the malicious model for several circuits. Times are averaged over 50 samples, with 95% confidence intervals.

domain-specific language. The TASTY compiler performs optimizations on the abstract syntax tree for the function being compiled. Kruger et al. developed an ordered BDD compiler to test the performance of their system relative to Fairplay [19]. Mood et al. focused on compiling secure functions on mobile devices with the PALC system, which involved a modification to the Fairplay compiler [23].

Recently, a compiler approach based on bounded model checking was present by Holzer et al. [12]. In that work, the CBMC system [5] was used to construct circuits, which were then rewritten to have fewer non-XOR gates. This approach had several advantages over previous approaches, most prominent being that functions could be described in the widely used C programming language, and that the use of CBMC allows for more advanced software engineering techniques to be applied to secure computation protocols. Like KSS12, however, this approach unrolls all loops (up to some fixed number of iterations), and converts a high level description directly to a boolean circuit which must then be optimized.

In addition to SFE, work on efficient compilers for proof systems has also been presented. Almeida et al. developed a zero-knowledge proof of knowledge compiler for Σ-protocols, which converts a protocol specification given in a domain-specific language into a program for the prover and the verifier to run [1]. Setty et al. presented a system for verifiable computation that uses a modification of the Fairplay compiler, which computes a system of quadratic constraints instead of boolean circuits, and emits executables for the prover and verifier [28, 29]. Our system is somewhat similar to these approaches, in that the circuit representation we present can be viewed as a program that is executed by the parties in the SFE system; however, our approach is unique in its handling of control flow and iterative constructs.

Closely related to our work is the Sharemind system [3, 14], which uses secure computation as a building block for privacy-preserving distributed applications. As in our approach, the circuits used in the secure computation portions of Sharemind are not fully unrolled until the protocol is actually run. Functions in Sharemind are described using a domain-specific language called SecreC. Although there has been work on static analysis for SecreC [26], the SecreC compiler does not perform automatic optimizations. By contrast, our approach is focused on allowing circuit optimizations at the bit-level to occur without having to unroll an entire circuit.

Kerschbaum has presented work on automatically optimizing secure computation at the protocol level, with an approach based on term and expression rewriting [15, 16]. This approach is based on maximizing the use of offline computation by inferring what each party can compute without knowledge of the other party's input, and does not treat the underlying secure computation primitives as a black box. It therefore requires additional work to remain secure in the malicious model. Our techniques could conceivably be combined with Kerschbaum's to reduce the overhead of online components.

8 Future Work

Our compiler can conceivably read any bytecode representation as input; one immediate future direction is to write translations for the instructions of another bytecode format, such as LLVM or the JVM, which would allow functions to be expressed in a broader range of languages. Additionally, we believe that our techniques could be combined with Sharemind, by having our compiler read the bytecode for the Sharemind VM and compute optimized PCF files for cases where garbled circuit computations are used in a Sharemind protocol.

The PCF format does not convey high-level information about data operations or types. Such information may further reduce the size of the circuits that are computed. Static analysis of such information by compilers has been widely studied, and it is possible that our compiler could be extended to support further reductions in the sizes of circuits emitted by the PCF interpreter. High-level information about data structures could also be used to improve the generation of circuits prior to optimization, using techniques recently presented by Evans and Zahur [6].

Our system and techniques can likely be generalized to the multiparty case, and to other representations of functions, such as arithmetic circuits. This would require significant changes to the optimization strategies and goals in our compiler, but fewer changes would be necessary for the PCF interpreter. Similar modifications to support homomorphic encryption systems are also possible.

9 Conclusion

We have presented an approach to compiling and storing circuits for secure computation systems that requires substantially lower computational resources than previous approaches. Empirical evidence of the improvement and utility of our approach is given, using a variety of functions with different circuit sizes and control flow structures. Additionally, we have presented a compiler for secure computation that reads bytecode as an input, rather than a domain-specific language, and have explored the challenges associated with such an approach. We also presented interpreters, which evaluate our new language on both PCs and phones.

The code for the compiler, PCF interpreters, and test cases will be available on the authors' website.

Acknowledgments This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A Certifying Compiler For Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 151–167, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, 2008.

[4] J. Boyar and R. Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2010.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] D. Evans and S. Zahur. Circuit structures for improving efficiency of security and privacy tools. In *IEEE Symposium on Security and Privacy (to appear)*, 2013.

[7] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985.

[8] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[10] V. Goyal, P. Mohassel, and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *Proceedings of 27th annual international conference on Advances*

in cryptology, EUROCRYPT'08, pages 289–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security*, 2010.

[12] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-Party computations in ANSI C. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

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

[14] R. Jagomägis. SecreC: a Privacy-Aware Programming Language with Apllications in Data Mining. Master's thesis, University of Tartu, 2010.

[15] F. Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 703–714, New York, NY, USA, 2011. ACM.

[16] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Conference on Data and Application Security and Privacy*, 2013.

[17] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ALP 2008*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[18] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[19] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, Alexandria, VA, Oct. 2006.

[20] L. Malka. VMCrypt: modular software architecture for scalable secure computation. In *ACM Conference on Computer and Communications Security*, pages 715–724, 2011.

[21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: A Secure Two-Party Computation System. In *13th Conference on USENIX Security Symposium*, volume 13, pages 287–302. USENIX Association, 2004.

[22] B. Mood. Optimizing Secure Function Evaluation on Mobile Devices. Master's thesis, 2012, University of Oregon.

[23] B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*, volume 7397. Springer Berlin Heidelberg, 2012.

[24] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-Party Computation Is Practical. In M. Matsui, editor, *Asiacrypt*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[25] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[26] J. Ristioja. An analysis framework for an imperative privacy-preserving programming language. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[27] T. Schneider. *Engineering Secure Two-Party Computation Protocols - Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012.

[28] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making Argument Systems for Outsourced Computation Practical (Sometimes). In *NDSS*, 2012.

[29] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX conference on Security symposium*, Berkeley, CA, USA, 2012.

[30] A. Yao. Protocols for Secure Computations. In *23rd Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

A PCF Semantics

The PCF file format consists of a header section that declares the input size, followed by a list of operations that are divided into subroutines. At runtime, these operations manipulate the internal state of the PCF interpreter, causing gates to be emitted when necessary. The internal state of the PCF interpreter consists of an instruction pointer, a call stack, an array of wire values, and an array of pointers. The pointers are positive integers. Wire values are 0, 1, or $\perp$, where $\perp$ represents a value that depends on input data, which is supplied by the code that invokes the interpreter. Each position in the wire table can be treated as a stack.

Each PCF instruction can take up to 3 arguments. The instructions and their semantics are as follows:

CLABEL/SETLABELC Appears only in the header, used for setting the input size for each party. CLABEL declares the bit width of a value, SETLABELC sets the value.

FUNCTION Denotes the beginning of a subroutine. When the subroutine is called, the instruction pointer is set to the position following this instruction.

GADGET Denotes a branch target

BRANCH Takes two arguments: a target, declared with GADGET, and a location in the wire table. In the wire value is 0, the instruction pointer is set to the instruction following the target. If the wire value is 1, the instruction pointer is incremented. If the wire value is $\perp$, evaluation halts with an error.

FUNC Calls a subroutine, pushing the current instruction pointer onto the call stack.

PUSH Pushes a copy of the wire value at a specified position onto the stack at that position.

POP Pops a stack at a specified position. If there is only one value on that stack, evaluation halts with an error.

ALICEIN32/BOBIN32 Fetches 32 input bits from one party, beginning at a specified *bit* position in that party's input. The bit position is specified by an array of 32 values in the wire table. If any of the values is ⊥, evaluation halts with an error. The input values will all have the value ⊥, and will be stored in the wire table at positions 0 through 31.

SHIFT OUT Outputs a single bit for a given party

RETURN Return from a subroutine. The instruction pointer is repositioned to the value popped from the top of the call stack.

STORECONSTPTR Sets a value in the pointer table

OFFSETPTR Adds a value to a pointer, specified by an array of 32 wire values starting at a position in the wire table. If any value in the array is ⊥, evaluation halts with an error.

PTRTOWIRE Saves a pointer value as a 32 bit unsigned integer. Each of the bits is pushed onto the stack at a location in the wire table.

PTRTOPTR Copies a value from one position in the pointer table to another.

CPY121 Copy a wire value from a position specified by a pointer to a statically specified position.

CPY32 Copy a wire value from a statically specific position to a position specified by a pointer.

$g_{0,0}g_{0,1}g_{1,0}g_{1,1}$ Compute a gate with the specified truth table on two input values from the wire table, with output stored at a specified position. Logic simplification rules are applied when one or both of the input values is ⊥. If no simplification is possible, then the output will be ⊥ and the interpreter will emit a gate. This is used for both local computations such as updating a loop index, and for computing the gates used by the protocol.

A.1 Example PCF Description

Below is an example of a PCF file. It iterates over a loop several times times, XORing the two parties' inputs with a bit from the internal state.

```
GADGET: main
CLABEL ALICEINLENGTH 32
CLABEL BOBINLEGNTH 32
CLABEL xxx 32
SETLABELC ALICEINLENGTH 128
SETLABELC ALICEINLENGTH 128
FUNCTION: main
1111 32 0 0
0000 33 0 0
0000 34 0 0
0000 35 0 0
GADGET: L
0110 36 35 34
0001 35 36 36
0110 36 34 33
0001 34 36 36
0110 36 33 32
0001 33 36 36
ALICEINPUT32 0 0
0001 36 0 0
BOBINPUT32 0 0
0001 37 0 0
0110 38 37 36
0110 39 33 38
SHIFT OUT ALICE 39
BRANCH L 35
RETURN xxx
```