PAL: A Pseudo Assembly Language for Optimizing Secure Function Evaluation in Mobile Devices[☆]

Benjamin Mood[a,*], Kevin R. B. Butler[b,**]

[a]*Point Loma Nazarene University, San Diego, CA, USA*
[b]*Florida Institute for Cybersecurity Research, University of Florida, Gainesville, FL, USA*

Abstract

Secure function evaluation (SFE) on mobile devices, such as smartphones, allows for the creation of compelling new privacy-preserving applications. Generating garbled circuits on smartphones to allow for executing customized functions, however, is infeasible for all but the most trivial problems due to the high memory overhead incurred. We develop a new methodology of generating garbled circuits that is memory-efficient. Using the standard language (SFDL) for describing secure functions as input, we design a new pseudo-assembly language (PAL) and a template-driven compiler, generating circuits that can be evaluated with the canonical Fairplay framework. We deploy this compiler for Android devices and demonstrate that a large new set of circuits can now be generated on smartphones, with memory overhead to generate circuits solving the set intersection problem reduced by 95.6% for the 2-set case. We show our compiler's ability to interface with other execution systems and perform mobile phone specific optimizations on that execution system. We develop a password vault application to show how runtime generation of circuits can be used in practice. We also show that our circuit generation techniques can be used in conjunction with other SFE optimizations. These results demonstrate the feasibility of generating garbled circuits on mobile devices while maintaining the convenience of high-level function specification.

1. Introduction

Mobile phones are extraordinarily popular, with rates of adoption by consumers that are unprecedented in history. Smartphones have been particularly embraced, with the number of devices shipped skyrocketing from 296 million in

[☆]This manuscript has been accepted for publication in the Journal of Information Security and Applications. ©2018. This manuscript version is made available under the CC-BY-NC-ND 4.0 license `http://creativecommons.org/licenses/by-nc-nd/4.0/`

[*]Principal corresponding author
[**]Corresponding author
Email addresses: `bmood@pointloma.edu` (Benjamin Mood), `butler@ufl.edu` (Kevin R. B. Butler)

2010 [1] to over 355 million in the third quarter of 2015 alone [2]. The increasing importance of the mobile computing environment requires functionality tailored to the limited available resources. Concerns of portability and battery life necessitate design compromises for mobile devices compared to servers, desktops, and even laptops. In short, mobile devices will always be resource-constrained compared to their larger counterparts. However, through careful design and implementation, they can provide equivalent functionality while retaining the advantages of ubiquitous access. They have the ability to perform financial transactions like e-commerce as long as there is cell service.

Privacy-preserving computing is particularly well suited to deployment on mobile devices. For example, two parties may be bartering in a marketplace but they do not want others finding out the nature of their transaction. Furthermore, they do not want to reveal unnecessary information to each other. Such a transaction is ideally suited for *secure function evaluation*, or SFE. Recent work, such as by Huang et al. [3], demonstrates the myriad applications that may be seen through deployment of SFE on smartphones. These applications may perform computations between two smartphones or between a smartphone and server. An example of such an application would be a password vault performing encryption under SFE. This style of application has advantages over standard encryption techniques and is discussed in Section 6. However, because of the computational and memory requirements associated with garbled circuit evalution, it is infeasible to perform circuit compilation and evaluation solely within the mobile environment. Solutions have appeared that consider server-aided, or outsourcing methods of performing garbled circuit computation by pushing operations to third-party servers [4, 5, 6, 7, 8]; however, strong assumptions are made that no collusion occurs between the powerful servers, which may not hold true in real-world usage.

While numerous research initiatives have considered how to evaluate these circuits more efficiently [9, 10], there has been little work in determining how to *generate* the circuits in a memory-efficient manner. Two parties are often interested in customizing functions to be evaluated based on their particular requirements, but having to outsource the circuit generation to a third party can reveal information about the computation to be performed, which can be a privacy compromise; hence, it is important to be able to perform this compilation on devices that will also evaluate the functions. Our port of the canonical Fairplay [11] compiler for SFE to the Android mobile operating system revealed that because of intensive memory requirements, the majority of circuits could not be compiled in this environment. As a result, our main contribution is a novel design to compile the high-level Secure Function Definition Language (SFDL) used by Fairplay and other SFE environments into garbled circuits (GCs) with minimal memory usage. We created Pseudo Assembly Language (PAL), a mid-level intermediate representation (IR) compiled from SFDL, where each instruction represents a pre-built circuit, including providing production rules for the transformation of expressions. We created a Pseudo Assembly Language Compiler (PALC), which takes in a PAL file and outputs the corresponding circuit in Fairplay's syntax. We then created a compiler, Fairplay Pseudo Assembly Language Compiler (FPPALC), to compile SFDL files into PAL and then, using PALC, to the Secure Hardware Definition Language (SHDL) used by Fairplay for circuit evaluation.

Using these compilation techniques, we are able to generate circuits that were previously infeasible to create in the mobile environment. For example, the set intersection problem with sets of size two requires 469 KB of memory with our techniques versus over 10667 KB using a direct port of Fairplay to Android, a reduction of 95.6%. We are able to evaluate results for the set intersection problem using four and eight sets, as well as other problems such as Levenshtein distance; none of these circuits could previously be generated at all on mobile devices due to their memory overhead. We have also demonstrated how our compiler can be integrated into other techniques, particularly the pipelined execution framework developed by Huang et al. [10]. We also provide a new analysis on how runtime performance can be improved by examining smartphone operations and developing a new memory management system for dealing with the allocation of Java `BigInteger` objects used during the creation of garbled circuits. Our new approach can further reduce memory and performance overheads. These techniques provide a new arsenal in conjunction with improved evaluation techniques to make privacy-preserving computing on mobile devices a feasible proposition.

In this paper, we extend the results of our preliminary work developing a memory-efficient compiler, presented at FC'12 [12]. In this expanded version, we provide an enlarged description of our memory-efficient technique for generating the circuits needed to perform secure function evaluation on smartphones and also show how optimized memory usage on smartphones can affect runtime. We include additional details about the PAL language (Backus-Naur form (BNF) grammar, a full list of operations and operators, and many transformation rules). Additionally, we implemented an interpreter to integrate PAL with a more efficient SFE execution system [10] and then performed memory optimizations to improve performance of that system. We also include additional discussion points and an impact section.

The rest of this paper is organized as follows. Section 2 provides background on secure function evaluation and the garbled circuits used for this evaluation, as well as the Fairplay SFE compiler. Section 3 describes the design of PAL, our pseudo assembly language, and PALC, our compiler to convert PAL into SHDL. We also describe FPPALC, which converts SFDL to PAL. We also combined FPPALC and PAL for full translation form SFDL to SHDL. Section 4 describes our testing environment and methodology, and provides benchmarks on memory and execution time. Section 5 lists a mobile specific optimization we make to an execution system. Section 6 describes applications that demonstrate circuit generation in use, while Section 8 describes related work and Section 9 concludes.

2. Background

2.1. Secure Function Evaluation with Fairplay

The origins of SFE trace back to Yao's pioneering work on garbled circuits [13]. While many approaches to performing SFE use Yao's protocol, including the Fairplay system (described below), alternative methods exist, such as Kruger et al.'s use of ordered binary decisions diagrams [14]. SFE enables two parties to jointly compute a function without knowing each other's input and without the presence of a trusted third party. More formally, given participants Alice and Bob with input vectors $\vec{a} = a_0, a_1, \cdots a_{n-1}$ and $\vec{b} = b_0, b_1, \cdots b_{m-1}$ respectively, they wish to compute a function $f(\vec{a}, \vec{b})$ without revealing any information about the inputs that cannot be gleaned from observing the function's output. Fundamentally, SFE is predicated on two cryptographic primitives. *Garbled circuits* allow for the evaluation of a function without either party gaining any information about the participants' input or output, which is not learned by their own output. This is possible since one party creates the garbled circuit and the other party evaluates it without knowing what the internal circuit values represent. Secondly, an *Oblivious Transfer* (OT) allows the party executing the garbled circuit to obtain the correct garbled values for that party's inputs from the other party without gaining or leaking information about the other's input values; in particular, a 1-out-of-n OT protocol allows Alice to learn about one piece of data without gaining any information on the remaining $n-1$ pieces.

Fairplay consists of two components. The first, a compiler, reads in a program written in a language that describes the circuit operations. Its output resembles a hardware description language and acts as input to the second component, the execution system. This piece is responsible for performing the garbled circuit protocol operations between the two parties.

2.2. Garbled Circuits

A garbled circuit is an encrypted version of the Boolean circuit representation of the function to be evaluated. Both the gates and wires of the circuit are garbled. The gates take in wires as input; the wires' values are represented by two pseudo-random fixed-length strings, which are representations of 0s or 1s. Similar to a standard Boolean gate, the garbled gate evaluates the inputs and gives a single output, but the garbled gate's truth table is encrypted. For a two-input garbled gate, the truth table entries TT are computed using the formula $TT_{i,j} = Enc(x_i||y_j||g) \oplus w_{i,j}$ where x is one of the two input wires with the value representing i, y is the other input wire with the value representing j, g is the gate number, and $w_{i,j}$ is the non-encrypted wire representing the i, j^{th} value of the truth table. Given a set of input wires x_i and y_j and output g, wire $w_{i,j}$ can be determined from the TT values.

The order of the entries in the table is permuted to prevent the order of the truth table from leaking information. Fairplay, rather than randomly permuting TT, permutes the truth table by using a specific bit, known as the *permute* *bit*, on both x and y. Given the permute bit l of x and permute bit r of y, the permuted truth table is setup such that $r * 2 + l$ will always map to the corresponding truth table entry.

Consequently, the only values saved for the truth table are the four encrypted output values, TT. A two-input gate is thus represented by the four encrypted output values. Given a set of input wires x_i and y_j and g, the output, wire $w_{i,j}$, can be determined from TT by using the permute bits (to find the correct truth table entry), and $w_{i,j}$ by $w_{i,j} = TT_{i,j} \oplus Enc(x_i || y_j || g)$.

The garbled circuit protocol requires that both parties are able to enter input into the circuit. If Bob creates the circuit and Alice evaluates it, Bob already knows the wire values that map to his input; however, Alice must perform an oblivious transfer with Bob to receive the wire values that map to her input values. Once she knows the wire values for her input, she evaluates each gate within the garbled circuit in order. To evaluate a gate, she uses the input values as the key to decrypt the corresponding output value, as described above.

Once all gates are evaluated, Alice will have the garbled wire versions of both parties' output. To understand her own output, Alice acquires a *translation table*, which is a hash of the wires within the garbled circuit corresponding to all possible values of her output, from Bob. She then hashes her output wires to see which wires are set. Alice sends Bob's output in garbled form and he is able to asynchronously interpret the wire values.

Briefly, we can describe the high-level operation of the Fairplay garbled circuit protocol as follows:

1. Bob creates N garbled circuits and sends them to Alice.
2. Alice picks one of these garbled circuits to evaluate and informs Bob, which circuit she will evaluate.
3. Bob sends Alice the secrets for all other circuits.
4. Using these secrets, Alice checks the correctness of these circuits and aborts if one of those circuits was found to be incorrect.
5. Bob sends his input to Alice in garbled form.
6. Alice performs an oblivious transfer with Bob to transform her plaintext input into garbled input so it can be entered into the garbled circuit.
7. Alice inputs both parties garbled input values into the garbled circuit and evaluates the circuit.
8. Both parties attain their outputs. Bob sends Alice a table, which maps her output to 1 and 0 values. Alice sends Bob his output values in its garbled form. Bob, once he receives these values can transform them back into its 1 or 0 values.

This protocol is an example of a cut-and-choose protocol [15, 16]. In a cut-and-choose protocol, many circuits are created and some number of those are evaluated to receive an output from the SFE computation, while the remaining circuits are checked to ensure Bob created the correct circuits. Fairplay only uses 1 evaluation circuit while more recently protocols, such as Kreuter et al. [17], use more than 1 evaluation circuit to maximize security.

2.2.1. Oblivious Transfer

The circuit evaluation portion of Fairplay provides for the execution of the garbled circuit protocol and uses oblivious transfer (OT) to exchange information. Fairplay uses the 1-out-of-2 OT protocols of Bellare et al. [18] and Naor et al. [19] which allows for Alice to pick one of two items that Bob is offering and also prevents Bob from knowing which item she has picked. These are secure in the random oracle model and secure against malicious users, as noted in the Fairplay paper. Also noted, the OT protocol of Bing et al. [20] takes into account other threat models, which include malicious and covert users.

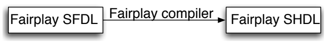
2.3. Fairplay Compiler

The Fairplay compiler is the canonical tool for generating circuits for secure function evaluation. It is notable for creating the abstraction of a high-level language, known as SFDL, for describing secure evaluation functions, and compiling them to SHDL, which is written in the style of a hardware description language such as Verilog or VHDL, which describe circuits.

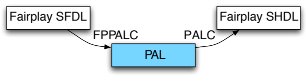
Examining the compiler in more detail, Fairplay compiles each instruction written in SFDL into a so-called *multi-bit instruction*. These multi-bit (e.g., integer) instructions are transformed to *single-bit instructions* (e.g., the 32 separate bits to represent that integer). From these single-bit instructions, Fairplay then unrolls variables and then transforms the instructions into SHDL and outputs the file, either immediately or after further circuit optimizations.

Fairplay's circuit generation process is very memory-intensive. We performed a port of Fairplay directly to the Android mobile platform (described further in Section 4) and found that a large number of circuits were completely unable to be compiled. We turned to examining the results of circuit compilation on a PC to determine the scope of memory requirements. From tests that we performed on a 64-bit Windows 7 machine, we noticed Fairplay needed at least 245 megabytes of memory to run the compilation of the keyed database program of size 16, an example of an SFE problem where a program matches keys with database lookups in a privacy-preserving manner (described further in Section 4). Our first task was to analyze the memory usage of Fairplay's compiler.

From our analysis, Fairplay uses the most memory during the mapping operation from multi-bit to single-bit instructions. During this phase, the memory requirements increased by 7 times when the keyed database program ran. We concluded that it would be easier to create a new system for generating the SHDL circuit file, rather than making extensive modifications to the existing Fairplay implementation. To accomplish this, we created an intermediate language that we called PAL, described in detail in section 3.



(a) Fairplay compiler process.



(b) PAL compiler process.

Figure 1: Compilation with Fairplay versus PAL.

2.4. Threat Model

There are two primary threat models in SFE literature. Protocols, which protect against honest-but-curious adversaries and protocols that protect against malicious adversaries. The stronger protocol (malicious) also protects against honest-but-curious adversaries as well.

In all cases (i.e., any number of circuits), Fairplay will protect against honest-but-curious adversaries. Honest-but-curious adversaries will obey the protocol but may look at any intermediate data the protocol produces in an attempt to gain additional information about the other party's input or output. The assumption of the honest-but-curious model is well-described by others considering secure function and secure multiparty computation, such as Kruger et al.'s OBDD protocol [14], Pinkas et al.'s SFE optimizations [9], the tools for Automating Secure Two-partY computations (TASTY) [21], Jha et al.'s privacy-preserving genomics [22], Brickell et al.'s privacy-preserving classifiers [23] and Huang et al.'s pipelined circuit execution techniques [3].

The authors of Fairplay also note that if either party were to deviate from this protocol and become malicious it may allow for one of the parties to get information from the other. Fairplay does have some defense for adversaries, which act maliciously. However, unlike more recent work, in which the probability a malicious adversary can succeed and attain extra information is $\frac{1}{2^s}$ [24], where s is the number of circuits, in Fairplay the probability is $\frac{1}{s}$. We do not attempt to compare Fairplay's definitions to more recent definitions, but note they the probability in succeeding in breaking Fairplay is substantially higher. We note that with some additions, the garbled circuit protocol of Fairplay may be modified to be more secure in the presence of malicious adversaries, as shown by Lindell et al. [25]. Other protocols, such as those proposed by Bing et al. [20], take other threat models into account. Our proof of concept tests adhere to the threat models as defined by Fairplay. We further discuss malicious adversaries in Section 6.4.

We are primarily interested protecting against honest-but-curious adversaries. However, we also want to show our implementation (discussed later) can also be used with more than a single circuit (as one circuit is all that is required for honest-but-curious adversaries). To this end, we run all our test cases with two circuits.

As with all of these protocols, we also assume the user enters in the correct (honest) input. Fairplay is secure in the random oracle model, implemented using the SHA-1 hash function. The oblivious transfer protocols in Fairplay (and hence, by PAL) are those from Bellare et al. [18] and Naor et al. [19] as described

Possible Operations		
Operation	Syntax	Usage
Variable Declarations	`Variables:`	Must be first
Procedure Declarations	`Procedure:` **NAME**	May be mixed with function declarations
Function Declarations	`Function:` **NAME** [takes paramName1 paramName2 ... paramNameN] [returns returnName1 returnName2 ... returnNameN] end	May be mixed with procedure declarations
Main Declaration	`Instructions:`	Must be last

Table 1: PAL headings

above. We assume that the generic case is that this is a programmable random oracle model, but Fairplay (and PAL) could be implemented using the Ishai et al. OT extensions [26] where proofs are performed against a non-programmable random oracle; we leave this for future work.

3. Design

To overcome the intensive memory requirements of generating garbled circuits within Fairplay, we designed a *pseudo assembly language*, or PAL, and a *pseudo assembly language compiler* called PALC. As noted in Figure 1, we change Fairplay's compilation model by first compiling SFDL files into PAL using our *fairplay pseudo assembly language compiler* or FPPALC, and generating the SHDL file which can then be run using Fairplay's circuit evaluator. Any runnable SFDL program can be represented in PAL.

3.1. PAL

We first describe PAL, our memory-efficient language for garbled circuit creation. PAL resembles an assembly language where each instruction corresponds to a pre-optimized circuit. PAL is composed of at least two parts: variable declarations and instructions. PAL files may also contain functions and procedures.

The heading syntax is defined in Table 1. Variable declarations or assembly instructions come after the headers.

Table 2 lists the set of operations that are available in PAL along with their instruction signatures. Each operation consists of a destination, an operator, and one to three operands. `DEST`, `V1`, `V2`, and `COND` are variables in our operation listing. PAL also has operations not found in Fairplay, such as shift and rotate. These two operations also take an N value, an integer, for the size of the shift or rotation. PAL does not have multiplication or division operators as SFDL does not have complete implementations of multiplication and division (the symbols exist in the compiler, but it errors when you use them).

Possible Operations	
Operation	Syntax
Addition	DEST + V1 V2
Subtraction	DEST - V1 V2
Less than	DEST < V1 V2
Greater than	DEST > V1 V2
Less than or Equal to	DEST <= V1 V2
Greater than or Equal to	DEST >= V1 V2
Equal to	DEST == V1 V2
Not Equal to	DEST != V1 V2
Bitwise AND	DEST & V1 V2
Bitwise OR	DEST | V1 V2
Bitwise XOR	DEST ^ V1 V2
Bitwise NOT	DEST ! V1
Shift Left	DEST << N V1
Shift Right	DEST >> N V1
Rotate Left	DEST ROT N V1
Set Equal	DEST = V1
If Conditional	DEST IF COND V1 V2
Input line	INPUT V1 a (or INPUT V1 b)
Output line	INPUT V1 a (or INPUT V1 b)
For loop	V1 FOR X (an integer) to Y (an integer)
Call a procedure	V1 PROC
Call a function	DEST,...,DEST = FunctionName(param, ... ,param)
Multiple Set Equals	DEST,...,DEST=V,...,V

Table 2: All PAL Operations.

The `IF` statement assigns either `V1` or `V2` to the destination based upon the rightmost bit of the `COND` variable. All IF operations in a high level language can be reduced to the `IF` conditional. Unlike in a program which jumps if the IF statement is not needed, in a circuit all parts of the IF statement must be executed every evaluation.

The first part of a PAL program is the set of variable declarations. These consist of a variable name and bit length, and the section is marked by a *Variables:* label. In this low-level language there are no structs or objects, only integer variables and arrays. Each variable in a PAL file must be declared before it can be used. Array indices may be declared at any point in the variable name. The `IN` and `OUT` operations, when used with arrays, take in full arrays and not just a single variable so the user does not have to write out all the input statements.

Figure 2 shows an example of variables declared in PAL. `Alicekey` and `Bobkey` have a bit length of 6, `Bobin` and `Aliceout` have a bit length of 32, `COND` is a boolean like variable and has a bit length of 1, and `Array[7]` is an array of seven elements with a bit length of 5. All declared variables are initialized to 0. After variable declarations, a PAL program can have function and procedure

```
Variables:
Alicekey    6
Bobin       32
Bobkey      6
Aliceout    32
COND        1
Array[7]    5
```

Figure 2: Example of variable declarations in PAL.

```
Instructions:
Bobin IN b
Bobkey IN b
Alicekey IN a
COND == Alicekey Bobkey
Aliceout IF COND
        Bobin Aliceout
Aliceout OUT a
```

Figure 3: Example of number comparison (for keyed database problem) in PAL.

definitions preceding the *Instructions:*, which is the main function. After a heading, any PAL instructions that follow it are part of that portion of the program: be it a function, procedure, or the instructions.

Figure 3 shows the PAL instructions for comparing two keys as used in the keyed database problem, described more fully below. The first two statements are input retrieval for Bob, while the third retrieves input for Alice. A boolean like variable `COND` is set based on a comparison and the output is set accordingly. Note that constants are allowed in place of `V1`, `V2`, or `COND` in any instruction.

PAL supports loops, functions, and procedures. Like other programming languages, a `FOR` loop only affects the next statement; otherwise, a procedure that contains multiple statements is needed, which `FOR` can loop over. We use `FOR` loops instead of *goto* to be consistent with SFDL. Functions are similar to those in other languages with the exception that they can return any number of variables. A function may only be called on the right side of a `set equal` statement. To deal with the equality of structures defined in the higher level SFDL language and with multiple returns from a function, we added the ability of `set equal` statements to have multiple left and right side variables where the corresponding leftmost variable on the left side is set to the variable on the leftmost side of the right side of the `set equal` statement. Figure 4 shows the BNF grammar for PAL.

To illustrate a full program, Figure 5 shows the keyed database problem in PAL, where a user selects data from another user's database without any information given about the item selected. In this program, Bob enters 16 keys and 16 data entries and Alice enters her key. If Alice's key matches one of Bob's then Alice's output of the program is Bob's data entry that held the corresponding key. The PAL program shows how each key is checked against Alice's key. If one of those keys matches, then the output is set.

3.2. PALC

Circuits generated by our PALC compiler, which generates SHDL files from PAL, are created using a database of pre-generated circuits matching instructions to their circuit representations. These circuits, with the exception of equality, were generated using simple Fairplay programs that represent an equivalent

```
<S> := <Var> <FP> <Main>
<Var> := <DeclareIdentifier> <Bitlength> <Var> | ϵ
<FP> := <Function> <FP> | <Procedure> <FP> | ϵ
<Function> := Function: <Identifier> <Takes> <Returns> end <AssemblyLines>
<Procedure> := Procedure: <Identifier> <AssemblyLines>
<Main> := Instructions: <InputLines> <AssemblyLines> <OutputLines>
<AssemblyLines>:= <AssemblyLine> <AssemblyLines> | ϵ
<AssemblyLine> := <BinaryOp> | <MonoOp> | <If> | <For> | <FunctionCall> | <ProcedureCall>

<InputLines> := <InputLine> <InputLines> | ϵ
<InputLine> := <Identifier> IN a | <Identifier> IN b
<OutputLines> := <OutputLine> <OutputLines> | ϵ
<OutputLine> := <Identifier> OUT a | <Identifier> OUT b

<BinaryOp> := <Identifier> <BinaryOperator> <Identifier> <Identifier>
<MonoOp> := <Identifier> <MonoOperator> <Identifier>
<If> :=<Identifier> IF <Identifier> <Identifier> <Identifier>
<For> := <Identifier> FOR <Number> <Number>
<FunctionCall>:= <IdentifierList> = <Identifier>(<Params>)
<ProcedureCall> := <Identifier> PROC
<BinaryOperator> := + | - | > | >= | < | <= | == | != | && | & | ’|’ ’|’ | ’|’ | ˆ
<MonoOperator> := = | ! | « | » | ROT

<DeclareIdentifier> := <Letter><DeclareName>
<DeclareName> := <DeclareArrayName> | <DeclareNotArray> | .<DeclareIdentifier>
<DeclareArrayName> := [<Number>]<DeclareName>
<DeclareNotArray> := <String><DeclareName> | <String> | ϵ

<IdentifierList> := <Identifier> | <Identifier>,<IdentifierList>
<Takes> := takes <SymbolList> | ϵ
<Returns> := returns <SymbolList> | ϵ
<SymbolList> := <StringStart> <SymbolList> | <StringStart>
<Params> := <ParameterList> | ϵ
<ParameterList> := <Identifier> | <Identifier>,<ParameterList>
<Identifier>:= <Letter><Name>
<Name> := <ArrayName> | <NotArray> | .<Identifier> | ϵ
<ArrayName> := [<StringStart>]<Name> | [<Number>]<Name>
<NotArray> := <String><Name>
<StringStart> := <Letter><String>
<String> := <Letter><String> | <Digit><String> | ϵ
<Bitlength> := <Digit><Number>
<Number>:= <Digit><Number>| ϵ
<Digit> := 0|1|2|3|4|5|6|7|8|9
<Letter> := a|b|...|z|A|...|Y|Z|$
```

Figure 4: BNF rules for PAL. For the | (or) symbol, literal uses are contained in ’ ’. White space is omitted from the above grammar, but required between words and symbols.

```
Variables:
i 6
in.a 6
in.b[16].data 24
in.b[16].key 6
out.a 24
$c0 1
$t0 1
DBsize 64

Procedure: $p0
$t0 == in.a in.b[i].key
$c0 = $t0
out.a IF $c0 in.b[i].data out.a

Instructions:
in.b[16].data IN b
in.b[16].key IN b
in.a IN a
DBsize = 16
i FOR 0 15
$p0 PROC
out.a OUT a
```

Figure 5: Representation of keyed database program in PAL.

functionality. We made our own optimized equality circuit. Variables hold integers signifying what gate they currently point to, meaning no gates need to be generated to represent them. Any operation that does not actually generate a gate is considered a *free* operation. Assignments, shifts, and rotates are free.

Variables in PALC have two possible states: they are either specified by a list of gate positions or they have a real numerical value. If an operation is performed on real value variables, the result is stored in the real value of the destination. These real value operations do not need a circuit to be created and are thus free.

When variables of two different sizes are used, the size of the operation is determined by the destination. If the destination is 24 bits and the operands are 32 bits, the operation will be done assuming the operands are 24 bits. This will not cause an error but may yield incorrect results if false assumptions are made.

Currently there are a number of known optimizations, such as removing static gates, which are not implemented inside PALC; these optimization techniques are a subject of future work. We did, however, add an optimization for dealing with arrays. When accessing an array variable where the index is based on user input the program must use equality statements to determine what value the index holds. This means that an array of size 16 requires 16 equals statements and 16 IF statements to determine which array index should be accessed. If the same variable is used twice in a statement then, naively, it requires 32 equal statements and 32 IF statements. If one of the pairs is the destination, then instead of 32 equal statements and 32 IF statements, each instruction is instead performed on the single array, resulting in 16 equals statements, 16 IF statements, and 16 operation statements.

3.3. FPPALC

To demonstrate that it is feasible to compile non-trivial programs on a phone, we modified Fairplay's SFDL compiler to compile into PAL and then run PALC to compile to SHDL. This compiler is called FPPALC. Compiling in steps greatly reduces the amount of memory that is required for circuit generation.

$$
\begin{array}{rl}
Assembly; [\![V1\ V2\ \ +]\!] & \Rightarrow Assembly, (\$t_i\ +\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ -]\!] & \Rightarrow Assembly, (\$t_i\ -\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ \&]\!] & \Rightarrow Assembly, (\$t_i\ \&\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ |]\!] & \Rightarrow Assembly, (\$t_i\ |\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ \hat{}\,]\!] & \Rightarrow Assembly, (\$t_i\ \hat{}\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ ==]\!] & \Rightarrow Assembly, (\$t_i\ ==\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ !=]\!] & \Rightarrow Assembly, (\$t_i\ !=\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ >=]\!] & \Rightarrow Assembly, (\$t_i\ >=\ V1\ V2); \$t_i \\
Assembly; [\![V1\ V2\ \ <=]\!] & \Rightarrow Assembly, (\$t_i\ <=\ V1\ V2); \$t_i \\
Assembly; [\![V1\ \ \sim]\!] & \Rightarrow Assembly, (\$t_i\ !\ V1); \$t_i \\
Assembly; [\![V1\ \ -]\!](unary\ minus) & \Rightarrow Assembly, (\$t_i\ -\ 0\ V1); \$t_i
\end{array}
$$

Figure 6: Production rules transforming SFDL postfix expressions to PAL.

$$
Assembly; [\![For(i\ =\ x\ to\ y)\ Statement]\!] \Rightarrow \\
(Procedure\colon \$p_i), [\![Statment]\!], Assembly, (i\ FOR\ x\ to\ y), (\$p_i PROC);
$$

Figure 7: Rules for transforming `FOR` Loops from SFDL to PAL.

$$
Assembly; [\![if(expression)\ Statement\ [else\ StatementOfElse]]\!] \Rightarrow \\
Assembly, (\$c_i = expression\ result), [\![Statement[X = Y :=\ X\ IF\ \$c_i\ Y\ X]]\!]; \\
[Else, if\ needed]\ Assembly, (\$c_i =!\ expression\ result), [\![StatementOfElse[X = \\
Y :=\ X\ IF\ \$c_i\ Y\ X]]\!];
$$

Figure 8: Rules for transforming `IF` statements from SFDL to PAL.

We now describe our circuit transformation protocol for expressions and other operations. First, using the predefined order of operations in Fairplay, we represent the expression in postfix notation. As we consume the expression, we find the first operator and create the corresponding PAL based on the production rules shown in Figure 6. In the figure, `Assembly` represents the expression produced in PAL. We concatenate the new PAL instruction onto the end of the existing expression denoted by `Assembly`. We also note the transformations for the `FOR` and `IF` statements and for functions in Figures 7, 8, and 9 respectively. To transform the PAL to SHDL we use a case statements with prebuilt circuits. Parentheses () denote new assembly statements while brackets $[\![]\!]$ denote statements yet to be translated.

We note our compiler will not yield the same functionality as Fairplay's compiler in two cases, which we believe demonstrate erroneous behavior in Fairplay. In these instances, Fairplay's circuit evaluator will crash or yield erroneous results. They are as follow: (1) when a user leaves a constant in the SFDL file and not does not optimize or tries to output a constant, which caused a crash of the evaluator, and (2) when the program consists of a single `IF` statement with a single assignment inside it. The SFDL specfication calls for all variables to be initialized to zero so that program output where the guard is false should

Function Definitions:

$$\begin{gathered} \mathit{Assembly}; [\![\mathit{type}\ \mathit{functionName}(\mathit{param}_1 ... \mathit{param}_n)\ \mathit{Statement}]\!] \Rightarrow \\ (\mathit{Function}\colon \mathit{functionName}\ \mathit{takes}\ \mathit{param}_1 ... \mathit{param}_n\ \mathit{returns}\ \mathit{Var}_1 ... \mathit{Var}_m), [\![\mathit{Statement}]\!], \mathit{Assembly}; \end{gathered}$$

Function Calls:

Case 1: single equals statement:

$$\begin{gathered} \mathit{Assembly}; [\![\mathit{structVar} = \mathit{function}(\mathit{param}_1 ... \mathit{param}_n)]\!] \Rightarrow \\ \mathit{Assembly}, (\mathit{funcVar}_1 ... \mathit{funcVar}_m\ =\ \mathit{functionName}(\mathit{param}_1 ... \mathit{param}_m); \end{gathered}$$

Case 2: in an expression:

$$\begin{gathered} \mathit{Assembly}; [\![\mathit{function}(\mathit{param}_1 ... \mathit{param}_n)]\!] \Rightarrow \\ \mathit{Assembly}, (\$t_i\ =\ \mathit{functionName}(\mathit{param}_1 ... \mathit{param}_m); \$t_i \end{gathered}$$

Figure 9: Rules for transformation of functions from SFDL to PAL: definition and calls

be 0 for all variables modified inside of the `IF` statement. However, even if the guard is false they are modified inside the `IF` statement. We implemented our compiler to ensure all variables are initialized to 0 as per the specification. An example program of this error is found in the appendix.

Apart from these small differences, the functionality of the two circuits is equivalent. Both approaches of circuit generation rely on an equivalent SFDL specification. Since both approaches generate the circuit from the SFDL specification, FPPALC's corresponding output circuit has the same functionality as the Fairplay circuit.

For our implementation of the SFDL to PAL compiler we took the original Fairplay compiler and modified it to produce the PAL output by removing all elements other than the parser. From the parser we built our own type system, before building support for basic expressions, assignment statements, and finally `IF` statements and `FOR` loops. All variables are represented as unsigned variables in the output but input and other operations treat them as signed variables. Our implementation of FPPALC and PALC, which compile SFDL to PAL and PAL to SHDL respectively, comprises over 7500 lines of Java code.

3.4. Garbled Circuit Security

A major question posed about our work is the following: *Does using an intermediate metalanguage with precompiled circuit templates change the security guarantees compared to circuits generated completely within Fairplay?* The simple answer to this question is no: we believe that the security guarantees offered by the circuits that we compile with PAL are equivalent to those from Fairplay.

Because there are no preconditions about the design of the circuit in the description of our garbled circuit protocol, any circuit that generates a given result will work: there are often multiple ways of building a circuit with equivalent functionality. Additionally, the circuit construction is a composition of

	Memory (KB)			Time (ms)		
Program	Initial	SFDL→PAL	PAL→SHDL	SFDL→PAL	PAL→SHDL	Total
Millionaires	4931	5200	5227	90	29	119
Billionaires	4924	5214	5365	152	54	206
CoinFlip	5042	5379	5426	139	122	261
KeyedDB	4971	5365	5659	142	220	362
SetInter 2	5064	5393	5533	161	305	466
SetInter 4	5078	5437	5600	135	1074	1209
SetInter 8	5122	5542	5739	170	6659	6829
Levenshtein 2	5184	5431	5576	183	336	519
Levenshtein 4	5233	5436	5638	190	622	802
Levenshtein 8	5264	5473	5693	189	2987	3172

Table 3: FPPALC on Android: total memory application was using at end of stages and the time it took.

existing circuit templates that were themselves generated through Fairplay-like constructions. Note that the security of Fairplay does not rely on the way the circuits are created but on the way garbled circuit constructs work. Therefore, our circuits will provide similar security guarantees since our circuits also rely on using the garbled circuit protocol. We also note that Huang et al. [10] considered circuit templates in the evaluator for further composition, including adders, muxers, and other broad functions.

A second question which can be asked of our system is *Can we guarantee the same program will be executed with a different compiler?* Given the transformation rules shown previously in Figures 6, 7, 8, and 9, we know the circuits generated will be semantically the same to the specification of SFDL. Thus we know the circuits we generate will produce circuits which are correct.

4. Evaluation

In this section, we demonstrate the performance of our circuit generator to show its feasibility for use on mobile devices. We targeted the Android platform for our implementation, with HTC Thunderbolt smartphones as a deployment platform. These smartphones contain a 1 GHz Qualcomm Snapdragon processor and 768 MB of RAM, with each Android application limited to a 24 MB heap.

4.1. Testing Methodology

We benchmarked compile-time resource usage with and without intermediate compilation to the PAL language. We tested on the Thunderbolts; all results reported are from these devices. Memory usage on the phones was measured by looking at the PSS metric, which measures pages that have memory from multiple processes. The PSS metric is an approximation of the number of pages used combined with how many processes are using a specific page of memory.

Several SFDL programs, of varying complexity, were used for benchmarking. Each program is described below. We use the SFDL programs representing the Millionaires, Billionaires, and Keyed Database problems as presented in Fairplay [11]. The other SFDL files, written by us, are presented in the appendix. We describe these below in more detail.

The *Millionaire's* problem describes two users who want to determine which has more money without either revealing their inputs. We used a 4-bit integer input for this problem. The *Billionaire's* problem is identical in structure but uses 32-bit inputs instead. The *CoinFlip* problem models a trusted coin flip where neither party can determine the program's outcome deterministically. It takes two inputs of 24-bit inputs per party. In the *Keyed database* program, a user performs a lookup in another user's database and returns a value without the owner being aware of which part of the database is looked up - we use a database of size 16. The keys are 6-bits and the data members are 24-bits. The *Set intersection* problem determines elements two users have in common, e.g., friends in a social network. We measured with sets of size 2, 4, and 8 where 24-bit input was used. Finally, we examined *Levenshtein distance*, which measures edit distance between two strings. This program takes in 8-bit inputs.

4.2. Results

Below the results of the compile-time tests performed on the HTC Thunderbolts are presented. We measured memory allocation and amount of time required to compile, for both the Fairplay and PAL compilers. In the latter case, we have data for compiling to and from the PAL language. Our complete compiler is referred to as FPPALC in this section.

4.2.1. Memory Usage & Compilation Time

Table 3 provides memory and execution benchmarks for circuit generation, taken over at least 10 trials per circuit. We measure the initial amount of memory used by the application as an SFDL file is loaded, the amount of memory consumed during the SFDL to PAL compilation, and memory consumed at the end of the PAL to SHDL compilation.

As an example of the advantages of our approach, we successfully compiled a set intersection of size 90 that had 33,000,000 gates on the phone. The output file was greater than 2.5 GB. Android has a limit of 4 GB per file and if this was not the case we believe we could have compiled a file of the size of the memory card (30 GB). This is because the operations are serialized and the circuit never has to fully remain in memory.

Although we did not focus on speed, Table 3 gives a clear indication of where the most time is used per compilation: the PAL to SHDL phase, where the circuit is output. The speed of this phase is directly related to the size of the program that is being output, while the speed of the SFDL to PAL compilation is based on how many individual instructions exist.

	Memory (KB)	
Program	Fairplay	FPPALC
Millionaires	658	296
Billionaires	1188	441
CoinFlip	1488	384
KeyedDB 16	NA	688
SetInter 2	10667	469
SetInter 4	NA	522
SetInter 8	NA	617
Levenshtein Dist 2	NA	392
Levenshtein Dist 4	NA	405
Levenshtein Dist 8	NA	429

Table 4: Comparison of memory increase by Fairplay and FPPALC during circuit generation.

4.2.2. Comparison to Fairplay

Table 4 shows the comparison of the Fairplay compiler and FPPALC. Where results are not present for Fairplay are situations where it was unable to compile these programs on the phone. For the set intersection problem with set 2, FPPALC uses 469 KB of memory versus 10667 KB by Fairplay, a reduction of 95.6%. Testing showed that the largest version of the keyed database problem that Fairplay could handle is with a database of size 10, while we easily compiled the circuit with a database of size 16 using FPPALC.

To determine just how large the programs we could compile were, we determined the maximum program size that the Fairplay compiler can compile on a phone. We used a program that adds single numbers together. We found we were able to have 342 addition operations when adding the constant 1. This compilation had about 20,000 gates. We should note this is the most generous possible program that could be constructed for Fairplay. Programs with array accesses (which the above did not have) require enormous amounts of memory, e.g. the keyed database, size 10 of which was able to successfully compile on the phone had 571 gates. Fairplay could not compile size 11 on the phone which had 629 gates.

4.2.3. Circuit Evaluation

Table 5 depicts the memory and time of the evaluator running the programs compiled by FPPALC. Consider again the two parties Bob and Alice, who create and receive the circuit respectively in the garbled circuit protocol. This table is from Bob's perspective, who has a slightly higher memory usage and a slightly lower run time than Alice. We present the time required to open the circuit file for evaluation and to perform the evaluation, or all operations other than the time it takes to load the initial non-garbled circuit into memory, using two different oblivious transfer protocols. As we describe in more detail below, we used both Fairplay's evaluator and an improved oblivious transfer (OT) protocol

	Memory (KB)			Time (ms)		
Program	Initial	Open File	End	Open File	Fairplay	Nipane
Millionaires	5466	5556	5952	197	533	406
Billionaires	5451	5894	6287	579	1291	981
CoinFlip	5461	5933	6426	789	1795	1320
KeyedDB 16	5315	6197	7667	1600	1678	1593
SetInter 2	5423	5993	6932	1511	2088	1719
SetInter 4	5414	7435	11711	8619	7714	7146
Levenshtein Dist 2	5617	6134	7162	1799	2220	2004
Levenshtein Dist 4	5615	7215	10787	7448	6538	6150
Levenshtein Dist 8	5537	12209	20162	29230	29373	27925

Table 5: Evaluating FPPALC circuits on Fairplay's evaluator with both Nipane et al.'s OT and the suggested Fairplay OT.

	Memory (KB)			Time (ms)	
Program	Initial	After File Opening	End	File Opening	Evaluating
Millionaires	5640	5733	5995	194	302
Billionaires	5536	5885	6303	631	958
+CoinFlip	5528	5796	6280	428	1062
KeyedDB 16	5551	6255	7848	2252	1955
SetInter 2	5439	6018	7047	1663	2131
SetInter 4	5553	7708	13507	10540	9555
+Levenshtein Dist 2	5568	5872	6316	529	781
+Levenshtein Dist 4	5577	6088	7178	1704	2213
Levenshtein Dist 8	5488	7670	13011	9745	8662

Table 6: Results from programs compiled with Fairplay on a PC evaluated with Nipane et al.'s OT.

developed by Nipane et al. [27]. Note that Fairplay's evaluator was unable to evaluate programs with around 20,000 mixed two and three input gates on the phone. This translates to 209 32-bit addition operations for our compiler.

Since the circuits that we generate are not optimized in the same manner as Fairplay's circuits, we wanted to ensure that their execution time would still be competitive against circuits generated by Fairplay. Because of the limits of generating Fairplay circuits on the phone, we compiled them using Fairplay on a PC, then used these circuits to compare evaluation times on the phone. Table 6 shows the results of this evaluation. Programs denoted with a + required edits to the SHDL to run in the evaluator to prevent their crashing due to the issues described in Section 3.3. By comparing the results in the Fairplay column of Table 5 and the Evaluating column of Table 6 we show the difference between the time Fairplay and FPPALC circuits took to evaluate. In many cases, evaluating the circuit generated by FPPALC resulted in faster evaluation. One anomaly to this trend was Levenshtein distance, which ran about three times slower using FPPALC. We speculate this is due to the optimization of constant addition operations. For instance, the optimizer knows if it is not possible for a specific variable's value will be over five then it does not need a full addition circuit and it can optimize the operation into a smaller circuit. Note, however, that these circuits are incapable of being generated on the phone and require pre-compilation. The size difference of the circuits can be extrapolated from the Tables by looking at time difference between Fairplay and FPPALC - since the amount of time a program takes is directly proportional to the circuit size.

4.3. Interoperability

To show that our circuit generation protocol can be easily used with other improved components for SFE, we used the faster oblivious transfer protocol of Nipane et al. [27], who replace the OT operation in Fairplay with 1-out-of-2 OT scheme based on a two-lock RSA cryptosystem. Shown in Table 6, this provides a speedup of over 24% for the Billionaire's problem mechanisms and 26% for the Coin Flip protocol. On average, there was a 13% speedup in evaluation time across all problems. with larger programs having a 5% reduction in evaluation time. For the *Millionaires*, *Billionaires*, and *CoinFlip* programs we disabled Nagle's algorithm as described by Nipane et al., leading to better performance on these problems. The magnitude of improvement decreased as circuits increased in size, a situation we continue to investigate. Our main findings, however, are that our memory-efficient circuit generation is complementary to other approaches that focus on improving execution time and can be easily integrated.

We speculate that the reason our findings for Nipane et al.'s oblivious transfer were not the results they achieved is due to the fact the bottlenecks on a mobile device are different from the bottlenecks on a desktop PC. In Section 5 we show how memory allocation and deallocation is much slower in proportion to a standard addition operation on a Phone as one example of different bottlenecks.

4.4. Pipelined Execution

To further extend the ability of our circuit creation scheme we created an interpreter to use the execution system of Huang et al. [10] with our language. Their system uses a pipelined execution which does not need the complete circuit to be stored in memory at the same time during the execution process. However they did not provide a way to generate circuits dynamically from a generalized language. It is possible to change the sizes of programs at runtime but once the program was compiled, using Java's compiler, it could not be structurally changed. We combined our compiler with their execution system. We were able to execute larger circuits on the phone which previously ran out of memory when executed with Fairplay.

While circuits are claimed to be created at runtime with the pipelined execution scheme, we found that the actual circuit generation worked differently.

	Exec. time (ns)		Proportion	
Program	PC	Phone	PC	Phone
Simple recursion of depth 200	16260	72297	903.4X	556.1X
1 addition	18	130	1.0X	1.0X
1 multiplication	18	154	1.0X	1.1X
1 string addition of two characters	2570	13183	142.8X	101.4X
Allocation of 1024 bytes	859	84618	47.8X	651.0X
Allocation of 10240 bytes	6044	571684	335.8X	4397.6X
Allocation of 8 bytes	97	2449	5.4X	18.8X
Creation of an 80 bit BigInteger	4946	44717	274.8X	344.0X

Table 7: Compares the time memory operations take on the phone compared to a PC and then compared with how many times slower that instruction is compared with an addition on the corresponding device.

Although the circuits are instantiated (i.e., read and allocated into memory), the actual circuit structure is hand coded and hand-optimized into the Java program. Our solution, FPPALC, is different; given a source file our interpreter does not need the Java compiler to execute the program, nor a reinstallation of the application as would be necessary on a mobile device. Additionally, our interpreter allows a user to write a program and execute it without examining the circuit level program.

5. MobileMem

We studied the runtime performance incurred by Java on our test phones by comparing the proportion which instructions took to execute on a PC vs a phone. We found that memory allocation and deallocation was several times slower on a phone than it was on a PC. The pipelined execution system uses BigIntegers to hold the numerical values. Since the BigInteger class is immutable most operations must allocate memory. We examined the time memory allocation and deallocation takes on the PC and phone.

Table 7 shows elapsed execution time for instructions on a phone and a PC over 100,000 instructions. Note that while certain operations take approximately the same amount of time between a phone and a PC, certain instructions such as memory allocation are considerably slower on the mobile device - note that allocation of 10,240 bytes incurs a slowdown of almost 4,400X compared to the simple instructions. This demonstrates to us that there are opportunities to significantly improve performance in the mobile environment. As described below, we implemented a new memory allocation scheme for the mobile environment, called MobileMem. We created our own custom class rather than reusing a built-in mutable class. Our class performs exactly the operations it needs and no more, which is important as we desire efficiency.

	Execution time (ms)		
Program	Huang et al.	MMInts	Reduction
Millionaires	70	40	57.7%
Billionaires	314	83	26.5%
CoinFlip	335	193	57.7%
KeyedDB 16	2135	691	32.3%
SetInter 2	1576	600	38.1%
SetInter 4	10375	2907	28.0%
SetInter 8	72058	18521	25.7%
SetInter 16	536565	129050	24.1%
Levenshtein 2	898	360	40.1%
Levenshtein 4	7105	2340	33.9%
Levenshtein 8	43999	11774	26.8%
Levenshtein 16	194067	48152	24.8%

Table 8: Comparison of the Huang et al.'s original execution phase and our own execution phase. Both execution systems use our interpreter.

5.1. Design

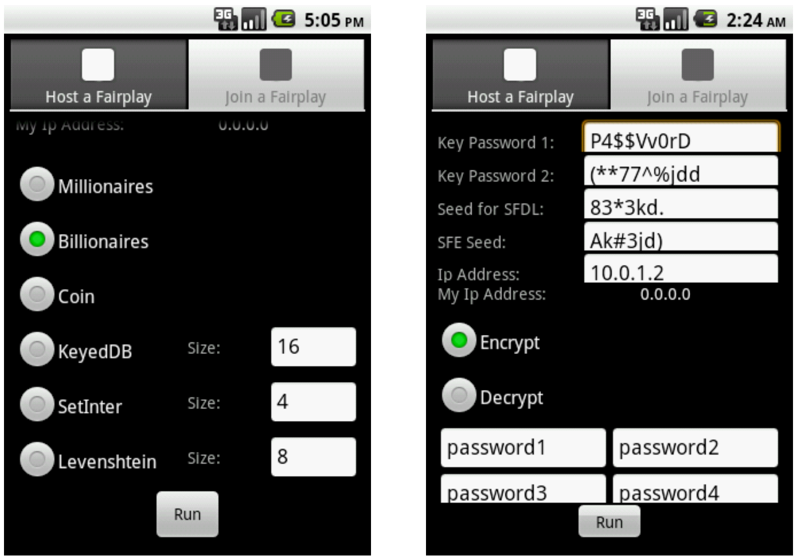
The MobileMem system comprises a customized buffer pool and system for handling large integers, removing the need for using the Java BigInteger class by replacing them with our own representations called MMInts. MobileMemis primarily a large integer array for addressing memory. A circular queue to keep track of which spots in the memory are free. Each variable previously represented as a Java BigInteger is now an integer pointing to its corresponding MMInt in memory. MMInt operations use a set of integers as input and execute on the set of corresponding MMInts in memory.

A form of deallocation also appears in MobileMem, but rather than performing the actual memory deallocation during a delete, we keep the MMInt structure in place in the buffer pool. This allows us to reuse the memory without a need for another allocation. We created functions for operations over both immutable and mutable variables.

A limitation to MobileMem is that MMInts are fixed in length. However, we designed a modification to allow dynamically increasing the length of digits. We use a second pool of memory containing a set of nodes, and link multiple nodes together, similar to indirect inode access in a file system.

We also applied other memory optimizations during conversion from BigIntegers to MMInt. Using MobileMem, we could optimize memory used for operations such as send and receive for network transmissions. Our solution was only applied to the circuit garbling and execution phase of the two-party computation, not to oblivious transfers.

Table 8 shows the results of our MobileMem system applied to the Huang et al. system with the corresponding execution system. In the execution phase, there was a speedup of up to 4X in larger programs. When we compared our interpreter to the custom circuits of Huang et al. we found even our optimized



My Ip Address: 0.0.0.0

( ) Millionaires
[✓ Billionaires]
( ) Coin
( ) KeyedDB Size: 16
( ) SetInter Size: 4
( ) Levenshtein Size: 8

Key Password 1: P4$$Vv0rD
Key Password 2: (**77^%jdd
Seed for SFDL: 83*3kd.
SFE Seed: Ak#3jd)
Ip Address: 10.0.1.2
My Ip Address: 0.0.0.0

[✓ Encrypt]
( ) Decrypt

password1 password2
password3 password4

(a) (b)

Figure 10: Screenshots of the GUI and password vault applications.

execution system was still slower than custom circuits by a factor of 2 for the Levenshtein distance problem. However, this does not diminish the value of our optimizations since many users, if not most, will not write the circuit level optimizations required for the improved efficiency. We could also apply our MobileMem system to the hand created circuits and get a speedup since they may also benefit from a better memory management strategy.

A particularly revealing result was desktop execution speed. The MobileMem system on a PC was negligibly faster or even slightly slower depending upon the input size used, showing that optimizations benefiting the mobile environment can differ considerably from those for desktops and servers.

6. Discussion

To demonstrate how our memory-efficient compiler can be used in practice, we developed Android apps capable of generating circuits at runtime. We describe these below.

6.1. GUI Based Editor

To allow use of the compiler on a phone we have to address one large problem. Our experience porting Fairplay to Android port showed the difficulty of writing a program on the phone. Figure 10 (a) shows an example of a GUI front-end for picking and compiling given programs based on parameters. A list of programs is given to the user who can then pick and choose which program they wish to run. For some of the programs there is a size variable that can also be changed.

6.2. Password Vault Application

We designed an Android application that introduces SFE as a mechanism to provide secure digital deposit boxes for passwords. In brief, this "password vault" can work in a decentralized fashion without reliance on the cloud or any third parties. If Alice fears that her phone may go missing and wants Bob to have a copy of her passwords, she and Bob can use their "master" passwords, as input to a pseudorandom generators. These master inputs are not revealed to either party, nor is the output of the generators, which is used to encrypt the password. If the passwords are ever lost, Alice can call Bob and jointly recover the passwords; both must present their master passwords to decrypt the password file, ensuring that neither can be individually coerced to retrieve the contents. This application also allows us to not need the cloud to store the information. Figure 10(b) shows a screenshot of this application.

Our evaluation shows that compiling the password SFDL program requires 915 KB of memory and approximately 505 ms, with 60% of that time is the PAL to SHDL conversion. Evaluating the circuit is more time intensive. Opening the file takes 2 seconds, and performing the OTs and gate evaluation takes 6.5 seconds. We are exploring efficiencies to reduce execution time.

The security of the password vault application is dependent upon whether the output of the pseudorandom number generator can be guessed. The keys are incremented for each password used to prevent same key attacks. The maximum length for a password in our program is 24 characters.

6.3. Experiences with Circuit Generation

One of the most important lessons from our implementation efforts was observing the large burden on mobile devices caused when complete circuits must be kept in memory. Better solutions only use small amounts of memory to direct the actual computation, for instance, one copy of each circuit instead of N for N of the same type of statement.

The largest difficulty of the full circuit approach is the need for the full circuit to be created. Circuits for $O(n^2)$ algorithms and beyond scale extremely poorly. A different approach is needed for larger scalability. For instance, doubling the Levenshtien distance parameter n increased the circuit size by a factor of about 4.5 (decreasing as n increases). For n of value 8, there are 11,268 gates; corresponding, n of size 16 yields 51,348 gates, $n = 32$ yields 218,676 gates, and $n = 64$ yields 902,004 gates.

In our first iteration of developing PAL, the compiler was limited in scalability, since it did not have loops, arrays, procedures, or functions. Once those programming structures were added, the length of PAL circuit files decreased dramatically. The resulting circuits generated from our improved version of PAL were syntactically similar to the original circuits and semantically equivalent.

6.4. Malicious Model

Increasingly, SFE proposals have shown their security in not only the semi-honest adversarial model, but also the rigirous standards of the modern malicious model. As an example, Huang et al. [28] implemented a process to achieve near malicious model security in SFE with only minimal changes. This is achieved by preforming the execution twice and then preforming a secure equality. For the second execution both parties switch their roles; the creator is now the evaluator and the evaluator is now the creator. The process incurs a minimal throughput decrease when the execution is preformed on dual core machine. Most phones are not currently dual core but it is expected to be more prevalent as technology is used more for power constrained devices.

This enhancement to security was implemented in the pipelined system we have already adapted. Since we already adapted this implementation to our compiler we know it is possible to adapt their new execution system to work with our language. The one downside to this approach is that this is a ‘’near” malicious model of security, instead of a complete malicious model of security. A party may gain a single bit of information they should not attain.

We note Fairplay's protocol does not achieve this with two circuits, and instead, if there is a cheater, reveals all secret information if the incorrect circuit is used during the circuit evaluation phase.

7. Impact of PAL

Since the original publication of this work, the authors are aware of two works that were directly affected by it: the PCF [29] and Frigate [30] compilers. The PCF compiler used templates to generate its sub-circuits, but then applied simplification rules to the circuit during SFE evaluation that allowed the circuit size to be substantially reduced. Unlike PAL, PCF used a technique to reduce the size of the output circuit by including jumps in the output format, which allowed PCF to not to have to unroll loops as in PAL.

The Frigate compiler was a direct result of this work. It functions very similarly to the PAL compiler; it uses templates and does not attempt to perform any global optimizations. In contrast to this work, Frigate uses the simplification rules from PCF to reduce the size of the circuit output at runtime.

The Frigate work incorporated an SFE evaluation system (a modified version of Kreuter et al. [17]) for testing the Frigate circuits. During the creation of the Frigate execution system, they, aware of this work, hand optimized the execution system's treatment of SFE values in many of the same way as we did in order to improve the performance of the Kreuter et al. [17] system.

The Frigate work also noted an error from the original PAL compiler where structs did not work. This error has since been fixed.

8. Related work

Current research has primarily focused on optimizing the actual transaction or generation of smaller circuits for SFE, while we focus on creating a memory efficient compiler. Kolesnikov et al. [31] demonstrated a “free XOR” evaluation technique to improve execution speed, while Pinkas et al. [9] implement techniques to reduce circuit size of the circuits and computation length. We plan to implement these enhancements in the next version of the circuit evaluator.

Huang et al. [10] have similarly focused on optimizing secure function evaluation, focusing on execution in resource-constrained environments. The approach differs considerably from ours in that users build their own functions directly at the circuit level rather than using high-level abstractions such as SFDL. While the resulting circuit may execute more quickly, there is a burden on the user to correctly generate these circuits, and because input files are generated at the circuit level in Java, compiling on the phone would require a full-scale Java compiler rather than the smaller-scale SFDL compiler that we use.

Another way to increase the speed of SFE has been to focus on leveraging the hardware of devices. Pu et al. [32] have considered leveraging Nvidia's CUDA-based GPU architecture to increase the speed of SFE. We have conducted preliminary investigations into leveraging vector processing capabilities on smartphones, specifically single-instruction multiple-data units available on the ARM Cortex processing cores found within many modern smartphones, as a means of providing better service for certain cryptographic functionality.

Kruger et al. [14] described a way to use ordered binary decision diagrams (OBDDs) to evaluate SFE, which can provide faster execution for certain problems. Our future work may include determining whether the process of creating the OBDDs can benefit from our memory-efficient techniques. TASTY [21] also uses different methods of privacy-preserving computation, namely homomorphic encryption (HE) as well as garbled circuits, based on user choices. This approach requires the user to explicitly choose the computation style, but may also benefit from our generation techniques for both circuits and the homomorphic constructions.

FairplayMP [33] showed a method of secure multiparty computation. We are examining how to extend our compiler to become multiparty capable. More recently an intermediate language using a pipelined execution model has been developed [34], but does not include the program control structures that we have developed, nor the ability to compile from a higher level language to it.

There have been multiple other implementations since, in both semi-honest [35, 36, 37, 38, 39] and malicious settings [29, 40].

Optimizations for garbled circuits include the free-XOR technique [41], garbled row reduction [42], rewriting computations to minimize SFE [43], and pipelining [44]. Pipelining allows the evaluator to proceed with the computation while the generator is creating gates.

Kreuter et al. [17] included both an optimizing compiler and an efficient run-time system using a parallelized implementation of SFE in the malicious model from [40].

9. Conclusion and Future Research

We introduced a memory efficient means for creating garbled circuits for making SFE tractable on the mobile platform. We created PAL, an intermediate language, between SFDL and SHDL programs and showed by using pregenerated circuit templates that we could make previously intractable circuits compile on a smartphone, reducing memory requirements for the set intersection circuit by 95.6%. We demonstrate the compiler's practicaliy through two smartphone applications. Our compiler with other execution systems and can be optimized to that specific system and mobile platform.

The creation of circuits for SFE in a fast and efficient manner is one of the central problems in the area. Previous compilers, from Fairplay to Kreuter et al. [17], were based on the concept of creating a complete circuit and then optimizing it. PAL represents the first system to use a simple template circuit, reducing memory usage by orders of magnitude. This has spurred additional substantial work in how to create better representations, including PCF [29], which built from this and used a more advanced representation to reduce the disk space used and has become an important artifact in the SFE community. More recently, we used the lessons learned from PAL in the design of Frigate [45], which provide both optimized representation and a validated platform for compiler design. PAL thus represents an important milestone in SFE compilers and has many design decisions and lessons that are valuable for current and future researchers in the privacy-preserving computation area.

Acknowledgements

We would like to thank Lara Letaw for her work on an early version of this paper, as well as Patrick Traynor and Michal Young for their insights regarding the narrative of the paper, and Adam Bates and Hannah Pruse for their comments. This work is supported by the National Science Foundation under grants CNS-1540217 and CNS-1540218. This material is based on research sponsored by DARPA under agreement number 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.

[1] Gartner, Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010, `http://www.gartner.com/it/page.jsp?id=1543014` (2011).

[2] IDC, Smartphone Shipments Reach Second Highest Level for a Single Quarter as Worldwide Volumes Reach 355.2 Million in the Third Quarter, According to IDC, `http://www.idc.com/getdoc.jsp?containerId=prUS25988815` (Oct. 2015).

[3] Y. Huang, P. Chapman, D. Evans, Privacy-Preserving applications on smartphones: Challenges and opportunities, in: Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec '11), 2011. URL `http://www.cs.virginia.edu/\~{}evans/pubs/hotsec2011/smartphones.pdf`

[4] H. Carter, B. Mood, P. Traynor, K. Butler, Secure outsourced garbled circuit evaluation for mobile devices, in: Proceedings of the USENIX Security Symposium, 2013.

[5] S. Kamara, P. Mohassel, B. Riva, Salus: a system for server-aided secure function evaluation, in: CCS '12: Proceedings of the 19th ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 2012, pp. 797–808.

[6] B. Mood, D. Gupta, K. Butler, J. Feigenbaum, Reuse it or lose it: More efficient secure computation through reuse of encrypted values, in: CCS '14: Proceedings of the 21st ACM Conference on Computer and Communications Security, ACM, 2014, pp. 582–596.

[7] H. Carter, C. Lever, P. Traynor, Whitewash: Outsourcing garbled circuit generation for mobile devices, in: Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, ACM, New York, NY, USA, 2014, pp. 266–275.

[8] H. Carter, B. Mood, P. Traynor, K. Butler, Outsourcing secure Two-Party computation as a black box, in: Cryptology and Network Security, Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 214–222.

[9] B. Pinkas, T. Schneider, N. P. Smart, S. C. Williams, Secure two-party computation is practical, in: Proceedings of ASIACRYPT, Tokyo, Japan, 2009.

[10] Y. Huang, D. Evans, J. Katz, L. Malka, Faster Secure Two-Party Computation Using Garbled Circuits, in: Proceedings of the 20th USENIX Security Symposium, San Francisco, CA, 2011.

[11] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, Fairplay: a secure two-party computation system, in: Proceedings of the 13th USENIX Security Symposium, San Diego, CA, 2004.

[12] B. Mood, L. Letaw, K. Butler, Memory-Efficient garbled circuit generation for mobile devices, in: FC '12: Proceedings of the 16th IFCA International Conference on Financial Cryptography and Data Security, Bonaire, 2012.

[13] A. C.-C. Yao, How to generate and exchange secrets, in: Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, Washington, DC, USA, 1986, pp. 162–167.
URL `http://portal.acm.org/citation.cfm?id=1382439.1382944`

[14] L. Kruger, S. Jha, E.-J. Goh, 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, 2006.

[15] Y. Lindell, B. Pinkas, Secure two-party computation via cut-and-choose oblivious transfer, in: Proceedings of the conference on Theory of cryptography, 2011.

[16] R. Zhu, Y. Huang, J. Katz, A. Shelat, The cut-and-choose game and its application to cryptographic protocols, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, 2016, pp. 1085–1100.
URL `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhu`

[17] B. Kreuter, a. shelat, C.-H. Shen, Billion-gate secure computation with malicious adversaries, in: Proceedings of the USENIX Security Symposium, 2012.

[18] M. Bellare, S. Micali, Non-interactive oblivious transfer and applications, in: International Crytology Conference, 1990.

[19] M. Naor, B. Pinkas, Efficient oblivious transfer protocols, in: Proceedings of SODA '01, Washington, DC, 2001.
URL `http://portal.acm.org/citation.cfm?id=365411.365502`

[20] Z. Bing, T. Xueming, X. Peng, J. Jiandu, Practical frameworks for h-out-of-n oblivious transfer with security against covert and malicious adversaries, Cryptology ePrint Archive, Report 2011/001, http://eprint.iacr.org/ (2011).

[21] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, TASTY: tool for automating secure two-party computations, in: Proc. 17th ACM Symposium on Computer and communications security (CCS'10), Chicago, IL, 2010.

[22] S. Jha, L. Kruger, V. Shmatikov, Towards Practical Privacy for Genomic Computation, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2008, pp. 216–230.

[23] J. Brickell, V. Shmatikov, Privacy-Preserving Classifier Learning, in: Proceedings of Financial Cryptography and Data Security, 2009.

[24] Y. Lindell, Fast cut-and-choose based protocols for malicious and covert adversaries, in: Advances in Cryptology–CRYPTO, 2013.

[25] Y. Lindell, B. Pinkas, An efficient protocol for secure two-party computation in the presence of malicious adversaries, in: In EUROCRYPT 2007, 2007, pp. 52–78.

[26] Y. Ishai, J. Kilian, K. Nissim, E. Petrank, Extending oblivious transfers efficiently, in: Proceedings of the Annual International Cryptology Conference, 2003.

[27] N. Nipane, I. Dacosta, P. Traynor, "Mix-In-Place" Anonymous Networking Using Secure Function Evaluation, in: Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2011.

[28] Y. Huang, J. Katz, D. Evans, Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution, IEEE Symposium on Security and Privacy (33rd).

[29] B. Kreuter, B. Mood, a. shelat, K. Butler, PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation, in: Proceedings of the USENIX Security Symposium (SECURITY '13), 2013.

[30] B. Mood, D. Gupta, H. Carter, K. Butler, P. Traynor, Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation, in: IEEE European Symposium on Security and Privacy (Euro S&P '16), 2016.

[31] V. Kolesnikov, T. Schneider, Improved Garbled Circuit: Free XOR Gates and Applications, in: Proceedings of ICALP '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 486–498.

[32] S. Pu, P. Duan, J.-C. Liu, Fastplay–A Parallelization Model and Implementation of SMC on CUDA based GPU Cluster Architecture, Cryptology ePrint Archive, Report 2011/097, http://eprint.iacr.org/ (2011).

[33] A. Ben-David, N. Nisan, B. Pinkas, Fairplaymp: a system for secure multi-party computation, in: Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, ACM, New York, NY, USA, 2008, pp. 257–266.

[34] D. Evans, W. Melicher, S. Zahur, Garbled circuit intermediate language, http://www.mightbeevil.org/gcparser/.

[35] M. Burkhart, M. Strasser, D. Many, X. Dimitropoulos, Sepia: Privacy-preserving aggregation of multi-domain network events and statistics, in: Proceedings of the 19th USENIX Conference on Security, USENIX Security'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 15–15.
URL `http://dl.acm.org/citation.cfm?id=1929820.1929840`

[36] I. Damgård, M. Geisler, M. Krøigaard, J. B. Nielsen, Asynchronous multiparty computation: Theory and implementation, in: Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09, Irvine, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 160–179. doi:10.1007/978-3-642-00468-1_10.

[37] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, Tasty: tool for automating secure two-party computations, in: Proceedings of the ACM conference on Computer and Communications Security, 2010.

[38] A. Holzer, M. Franz, S. Katzenbeisser, H. Veith, Secure two-party computations in ansi c, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, ACM, New York, NY, USA, 2012, pp. 772–783. doi:10.1145/2382196.2382278.

[39] Y. Zhang, A. Steele, M. Blanton, PICCO: A General-purpose Compiler for Private Distributed Computation, in: Proceedings of the ACM Conference on Computer Communications Security (CCS), 2013.

[40] a. shelat, C.-H. Shen, Two-output secure computation with malicious adversaries, in: Proceedings of EUROCRYPT, 2011.

[41] V. Kolesnikov, T. Schneider, Improved Garbled Circuit: Free XOR Gates and Applications, in: Proceedings of the international colloquium on Automata, Languages and Programming, Part II, 2008.

[42] B. Pinkas, T. Schneider, N. P. Smart, S. C. Williams, Secure Two-Party Computation is Practical, in: ASIACRYPT, 2009.

[43] F. Kerschbaum, Expression rewriting for optimizing secure computation, in: Conference on Data and Application Security and Privacy, 2013.

[44] Y. Huang, D. Evans, J. Katz, L. Malka, Faster secure two-party computation using garbled circuits, in: Proceedings of the USENIX Security Symposium, 2011.

[45] B. Mood, D. Gupta, H. Carter, K. Butler, P. Traynor, Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation, in: Proc. 1st IEEE European Symposium on Security and Privacy (Euro S&P 2016), 2016.

APPENDIX

Keyed Database program

```
program Keyed_DB_Search {
  const DBsize = 16;
  type Key = Int<6>;
  type Data = Int<24>;
  type Pair = struct {Key key, Data data};
  type AliceInput = Key;
  type BobInput = Pair[DBsize];
  type AliceOutput = Data;
  type Output = struct {AliceOutput alice,
                        BobOutput bob);
  type Input = struct {AliceInput alice,
                       BobInput bob};

  function Output output(Input input) {
    var Key i ;
    for (i = 0 to DBsize-1)
      if (input.alice == input.bob[i].key)
        output.alice = input.bob[i].data;
  }
}
```

Coin Flip program

```
program Coin {
  const InputSize = 2;
  type Data = Int<24>;
  type AliceInput = Data[InputSize];
  type BobInput = Data[InputSize];
  type AliceOutput = Data;
  type BobOutput = Data;
  type Output = struct {AliceOutput alice,
                        BobOutput bob};
  type Input = struct {AliceInput alice,
                       BobInput bob};

  function Output output(Input input)
  {
    var Data temp;
    temp = input.alice[0] ^ input.bob[0];
    temp = temp ^ input.bob[1];
    temp = temp&1;
    if(temp== (input.alice[1] & 1))
    {
      output.alice = 1;
      output.bob = 0;
    }
    else
    {
      output.alice = 0;
      output.bob = 1;
    }
  }
}
```

Set Intersection program

```
program SetIntersetion {
  const Size = 8;
  type Key = Int<10>;
  type Data = Int<24>;
  type AliceInput = Data[Size];
  type BobInput = Data[Size];
  type AliceOutput = Data[Size];
  type Output = struct {AliceOutput alice,
                        BobOutput bob};
  type Input = struct {AliceInput alice, BobInput bob};
  function Output output(Input input)
  {
    var Key i,k,j,index ; index=0;
    for (i = 0 to Size-1)
    {
      for (k = 0 to Size-1)
      {
        if (input.bob[i] == input.alice[k] )
        {
          for (j = 0 to Size-1)
          {
            if (index == j )
            {
              output.alice[j]= input.alice[k];
            }
          }
          index= index+1;
        }
      }
    }
  }
}
```

Levenshtein Distance program

```
program LevenshteinDistance {
  const bit = 1;
  const size = 8;
  const inputsize = 2;
  const Asize = inputsize+1;
  type Num = Int<size>;
  type Bit = Int<bit>;
  type AliceInput = Num[inputsize];
  type BobInput = Num[inputsize];
  type AliceOutput = Num;
  type BobOutput = Num;
  type Input = struct {AliceInput alice,BobInput bob};
  type Output = struct {AliceOutput alice,
                        BobOutput bob};
  function Output output(Input input)
  {
    var Num i,k,j;
    var Num temp1,temp2,temp3, result;
    var Bit answer;
    var Num[Asize][Asize] D;
    for (k=0 to Asize-1)
    {
      D[k][0] = k;
      D[0][k] = k;
    }
    for (i=1 to Asize-1)
    {
      for (j=1 to Asize -1)
      {
        if(input.alice[j-1] == input.bob[i-1])
        {
          D[i][j] = D[i-1][j-1];
        }
        else
        {
          temp1 = D[i-1][j] + 1;
          temp2 = D[i][j-1] + 1;
          temp3 = D[i-1][j-1] + 1;
          answer = temp2 < temp3;
          result = temp1;
          if ((temp2 < temp1)&answer)
            result = temp2;
          if((temp3 < temp1)& (temp3<temp2))
            result = temp3;
          D[i][j] = result;
        }
      }
    }
    output.alice = D[Asize-1][Asize-1];
    output.bob = D[Asize-1][Asize-1];
  }
}
```

Fairplay error example program

```
program FairplayError {
  const N=8;
  type Byte = Int<N>;
  type AliceInput = Byte;
  type BobInput = Byte;
  type AliceOutput = Byte;
  type BobOutput = Byte;
  type Input = struct {AliceInput alice, BobInput bob};
  type Output = struct {AliceOutput alice,
        BobOutput bob};
  function Output output(Input input)
  {
    if(input.bob>input.alice)
    {
      output.alice = input.bob & input.alice;
      output.bob = input.bob ^ input.alice;
    }
  }
}
```