OPTIMIZING SECURE FUNCTION EVALUATION ON MOBILE DEVICES

by

BENJAMIN MOOD

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

September 2012

THESIS APPROVAL PAGE

Student: Benjamin Mood

Title: Optimizing Secure Function Evaluation on Mobile Devices

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Kevin Butler	Chair
Michal Young	Member
and	
Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded September 2012

THESIS ABSTRACT

Benjamin Mood

Master of Science

Department of Computer and Information Science

September 2012

Title: Optimizing Secure Function Evaluation on Mobile Devices

Secure function evaluation (SFE) on mobile devices, such as smartphones, allows for the creation of new privacy-preserving applications. Generating the circuits on smartphones which allow for executing customized functions, however, is infeasible for most problems due to memory constraints. In this thesis, we develop a new methodology for generating circuits that is memory-efficient. Using the standard SFDL language for describing secure functions as input, we design a new pseudo-assembly language (PAL) and a template-driven compiler, generating circuits which can be evaluated with the canonical Fairplay evaluation framework. We deploy this compiler and demonstrate larger circuits can now be generated on smartphones. We show our compiler's ability to interface with other execution systems and perform optimizations on that execution system. We show how runtime generation of circuits can be used in practice. Our results demonstrate the feasibility of generating garbled circuits on mobile devices.

This thesis includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Benjamin Mood

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon
Point Loma Nazarene University, Point Loma, California

DEGREES AWARDED:

Master of Science, Computer & Information Science, 2012, University of Oregon
Bachelor of Science, Computer Science, 2010, Point Loma Nazarene University

AREAS OF SPECIAL INTEREST:

Cryptography
Security
Computer Architecture
Compilers
Game Design

PROFESSIONAL EXPERIENCE:

Adjunct, Point Loma Nazarene University, Point Loma, California, Summer 2010

GRANTS, AWARDS AND HONORS:

Graduate Research Fellowship, Computer & Information Science, Summer 2011 to present

Graduate Teaching Fellowship, Computer & Information Science, Fall 2010 to Spring 2011

PUBLICATIONS:

A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler. Detecting Co-Residency with Active Traffic Analysis Techniques. Department of Computer and Information Science, Univesrity of Oregon, CIS-TR-2012-04, Jul. 2012.

B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security (FC)*, February 2012.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Kevin Butler, for the time he put into me and this project and thesis. I would like to thank Michal Young for being on my Master's Thesis committee. I would also like to thank Adam Bates for his help in formatting this thesis. I would like the thank the OSIRIS lab for their input on this project. And lastly, I would like to thank DARPA, whom funded much of this research.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. Lara and Kevin contributed to writing this Chapter.

Mobile phones are extraordinarily popular, with rates of adoption by consumers that are unprecedented in history. Smartphones have been particularly embraced, with over 296 million of these devices shipped in 2010 [6]. 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 preform financial transactions like e-commerce on the move 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 do not want others finding out the nature of their transaction, and 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 Chapman et al. [8], demonstrates the myriad applications that may be seen through deployment of SFE on smartphones. These applications may preform computations between two smartphones or between a smartphone and server. An example of an application would be a password vault

which does encryption under SFE. This style of application has some advantages over standard encryption techniques and is discussed in Chapter VI.

However, because of the computational and memory requirements, it is infeasible to perform many of these operations in the mobile environment; often, the only hope of performing these operations would be to outsource computation to a cloud or other trusted third party, thus raising concerns about the privacy of the computation.

Our thesis statement is this: *Secure Function Evaluation can be made a viable application on smartphones through optimizations of current systems.*

In this thesis, we describe a memory-efficient technique for generating the garbled circuits needed to perform secure function evaluation on smartphones and also show how optimized memory usage on smartphones can affect runtime. While numerous research initiatives have considered how to evaluate these circuits more efficiently [20, 9], there has been little work in determining how to compile and generate the circuits required for evaluation 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 [17] 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 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. 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 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. Consequently, these techniques provide a new arsenal in conjunction with improved evaluation techniques to make privacy-preserving computing on mobile devices a feasible proposition.

All Chapters, other than Chapter V, have been published in some form in *Financial Cryptography and Data Security 2012*.

This thesis is structured in the following way: Chapter II provides background on secure function evaluation, the garbled circuits used for this evaluation, as well as the Fairplay SFE compiler and the other work related to our own. Chapter III 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. Chapter IV describes our testing environment and methodology, and provides benchmarks on

memory and execution time. Chapter VI describes applications that demonstrate circuit generation in use, and Chapter VII concludes.

CHAPTER II

BACKGROUND

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. There was some writing contribution but I was the contributor for the relevant testing and understanding of Fairplay.

2.1. Secure Function Evaluation with Fairplay

The origins of SFE trace back to Yao's pioneering work on garbled circuits [22]. While many approaches to performing SFE use Yao's protocol, including the Fairplay compiler (described below), alternative methods exist, such as Kruger et al.'s use of ordered binary decisions diagrams [14]. SFE enables two parties to 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 any party gaining additional information about the participants. This is possible since one party creates a garbled circuit and the other party evaluates the circuit without knowing what the wires represent. Secondly, an *Oblivious Transfer* (OT) allows the party executing the garbled circuit to obtain the correct wires for setting inputs from the other party without gaining additional information about the circuit; in particular, a 1-out-of-

n OT protocol allows Bob to learn about one piece of data without gaining any information on the remaining $n - 1$ pieces.

A garbled circuit is composed of many garbled gates, with inputs represented by two random fixed-length strings. Like a normal boolean gate, the garbled gate evaluates the inputs and gives a single output, but alterations are made to the garbled gate's truth table: aside from the randomly chosen input values, the output values are uniquely encrypted by the input wires and an initialization vector. The order of the entries in the table is then permuted to prevent the order from giving away the value. Consequently, the only values saved for the truth table are the four encrypted output values. A two-input gate is thus represented by the two inputs and four encrypted output values.

The garbled circuit protocol requires both parties to be able to enter input into the circuit. If Bob creates the circuit and Alice receives it, Bob can determine which wires to set, and Alice performs an oblivious transfer to receive her input wires. Once she knows her input wires she runs the circuit by evaluating each gate in order. To evaluate a gate, she uses the input values as the key to decrypt the output value. To find the correct entry in the table, Alice uses a decryption step using the input wires as keys. To find her output, Alice acquires a translation table, a hash of the wires, from Bob for her possible output values. She then can preform the hash on her output wires to see which wires were set. Alice sends Bob's output in garbled form since she cannot interpret it.

Fairplay is the canonical tool for generating and evaluating garbled 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 VHDL and describes the garbled circuit. 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. [1] and Naor et al. [18] 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.

Both protocols work under the Diffie-Hellman assumption in cryptography. They are secure in the random oracle model. This model says that if a user is supposed to choose a random number, they actually choose a random number. The protocol of Bing et al. [3] takes into account other threat models, which include malicious and covert users, but for our purposes we use the assumptions and protocols under which Fairplay is used. Weakening these assumptions to include more robust adversarial threat models is an extension for future work.

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

1. Bob creates N garbled circuits.
2. Alice picks one of these garbled circuits to evaluate.
3. Bob reveals to Alice the secrets for all other circuits.
4. Alice checks the correctness of these other circuits.
5. Bob sends his input to Alice.
6. Alice uses an oblivious transfer with Bob to acquire the wires she needs to set for her input.
7. Alice evaluates the circuit.

8. Both parties attain their outputs.

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 Chapter IV) 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, an example of an SFE problem where a program matches keys with database lookups in a privacy-preserving manner (described further in Chapter IV). 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.

2.2. Threat Model

As with Fairplay, which is secure in the random oracle model implemented using the SHA-1 hash function, our threat model accounts for an honest-but-curious adversary. This means the participants will obey the given protocol but may look at any data the protocol produces. Note that this 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 [20], the TASTY proposal for automating two-party communication [7], Jha et al.'s privacy-preserving genomics [11], Brickell et al.'s privacy-preserving classifiers [4] and Huang et al.'s pipelined circuit execution techniques [8]. Similarly, we make the well-used assumption that parties enter correct input to the function. The authors of Fairplay also note that if either party were to deviate from this protocol and become malicious it would impact security and allow for one of the parties to get information from the other. With some additional complexity, the garbled circuit protocol may be modified to be more secure in the presence of malicious adversaries, as shown by Lindell et al. [15]. Other protocols, such as those proposed by Bing et al. [3], take other threat models into account. Our proof of concept tests adhere to the threat model as defined by Fairplay, and we leave a stronger attacker model for future work.

2.3. 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. [12] demonstrated a "free XOR" evaluation technique to improve execution speed, while Pinkas et al. [20] 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. [9] 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. [21] 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 [7] 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 [2]

showed a method of secure multiparty computation. We are examining how to extend our compiler to become multiparty capable.

CHAPTER III

DESIGN

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. I created PAL, PALC, and FPPALC. I wrote this Chapter. Lara and Kevin helped edit this Chapter.

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 3.1., we change Fairplay's compilation model by first compiling SFDL files into PAL using our FPPALC compiler, and generating the SHDL file which can then be run using Fairplay's circuit evaluator.

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 3.1.. Variable declarations or assembly instructions come after the headers.

Table 3.2. 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.

(a) Fairplay compiler process. (b) PAL compiler process.

FIGURE 3.1.: Compilation with Fairplay versus PAL.

Possible Operations		
Operation	Syntax	Comment about use
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 3.1.: PAL headings

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.

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

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 3.2.: PAL Operations

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 3.2. 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 definitions preceding

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

FIGURE 3.2.: 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 3.3.: Example of number comparison in PAL.

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

Figure 3.3. 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 loops over the next statement. To loop over more than one statement a procedure is needed. The for loop loops over the procedure which contains multiple statements. We use for loops instead of *goto* since SFDL uses for loops. Functions are like other language's functions 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 gets set to the variable on the leftmost side of the right side of the set equal statement.

Figure 3.4. shows the BNF grammar for PAL.

To illustrate a full program, Figure 3.5. 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 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

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

<InputLines> := <InputLine> <InputLines>|ϵ
<InputLine> := <Identifier> IN a|<Identifier> IN b
<OutputLines> := <OutputLine> <OutputLine>|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> := +|-|>|>=|<|<=|==|!=|&& |& |‘|’ ‘|’|‘|’|^
<MonoOperator> := = |!||«|»|ROT

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

<IdentifierList> := <Identifier>|<Identifier>,<IdentifierList>
<Takes> := takes <SymbolList>|ϵ
<Returns> := returns <SymbolList>|ϵ
<SymbolList> := <SymbolList> <StringStart>|<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> := <Number>
<Number>:= <Digit><Number>|ϵ
<Digit> := 0|1|2|3|4|5|6|7|8|9
<Letter> := a|b|...|z|A|...|Y|Z
```

FIGURE 3.4.: BNF rules for PAL. For the | (or) symbol, literal uses are contained in ’ ’.

```
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 3.5.: Representation of keyed database program in PAL.

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 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

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

FIGURE 3.6.: Production rules transforming SFDL postfix expressions to PAL.

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

FIGURE 3.7.: Rules for transforming `FOR` Loops from SFDL to PAL.

compile to SHDL. This compiler is called FPPALC. Compiling in steps greatly reduces the amount of memory that is required for circuit generation.

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 3.6.. 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 3.7., 3.8., and 3.9. 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.

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

FIGURE 3.8.: Rules for transforming `IF` statements from SFDL to PAL.

Function Definitions:

$$Assembly; [\![type\ functionName(param_1...param_n)\ Statement]\!] \Rightarrow$$
$$(Function:\ functionName\ takes\ param_1...param_n\ returns\ Var_1...$$
$$Var_m), [\![Statement]\!], Assembly;$$

Function Calls:

Case 1: single equals statement:

$$Assembly; [\![structVar = function(param_1...param_n)]\!] \Rightarrow$$
$$Assembly, (funcVar_1...funcVar_m\ =\ functionName(param_1...param_m);$$

Case 2: in an expression:

$$Assembly; [\![function(param_1...param_n)]\!] \Rightarrow$$
$$Assembly, (\$t_i\ =\ functionName(param_1...param_m); \$t_i$$

FIGURE 3.9.: Rules for transformation of functions from SFDL to PAL: definition and calls

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 complete program is a single `IF` statement with a single assignment inside it. The spec called for all variables to be initialized to zero so the output of such a program where the guard is false should be 0 for all variables modified inside of the if. However, even if the guard is false they are modified inside of the if. It should be

noted 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 differences, the functionality of the two circuits are the same. Both approaches of generating the circuits 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 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. [9] 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 3.6., 3.7., 3.8., and 3.9., 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.

CHAPTER IV

EVALUATION

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. I did all of the experiments. Lara and Kevin helped edit this Chapter. Kevin contributed ideas for what experiments could be useful.

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 Thunderbolts 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 [16]. 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 4.1. 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

	Memory (KB)			Time (ms)		
Program	Initial	PAL	SHDL	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 Dist 2	5184	5431	5576	183	336	519
Levenshtein Dist 4	5233	5436	5638	190	622	802
Levenshtein Dist 8	5264	5473	5693	189	2987	3172

TABLE 4.1.: FPPALC on Android: total memory application was using at end of stages and the time it took.

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 4.1. gives a clear indication of where the most time is used per compilation: the 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 4.2.: Comparison of memory increase by Fairplay and FPPALC during circuit generation.

4.2.2. Comparison to Fairplay

Table 4.2. 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

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

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 4.3. 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 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 developed by Nipane et al. [19]. Note that Fairplay's evaluator was unable to evaluate programs with around 20,000 mixed two and three input gates on the phone. We assume the mixes of gates will vary, and translates for our compiler to 209 32-bit addition operations.

	Memory (KB)			Time (ms)	
Program	Initial	Open File	End	Open File	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 4.4.: Results from programs compiled with Fairplay on a PC evaluated with Nipane et al.'s OT.

While 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 4.4. 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 Chapter 3.3.. By comparing the results in the Fairplay column of Table 4.3. and the Evaluating column of Table 4.4. 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 approaches to SFE, we used the faster oblivious transfer protocol of Nipane et al. [19], who replace the OT operation in Fairplay with 1-out-of-2 OT scheme based on a two-lock RSA cryptosystem. Shown in Table 4.4., 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 Chapter V 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. [9] 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.

One point in their paper we would disagree with is the notion that circuits are created at runtime. Although the circuits are instantiated (read, allocated into memory), the actual circuit structure is hand coded and optimized by hand into the Java program. FPPALC is different; given a source file our interpreter does not need the Java compiler to execute the program. On a mobile phone this would also take a reinstallation of the application.

Our interpreter also allows a user to write a program and execute it without examining the circuit level program. More recently a group working on the pipelined execution from the same university has created a intermediate language that looks remarkably similar to our own - though without some of our program control structures and without a higher level language [5].

CHAPTER V

MEMORYMANAGEMENT

5.1. PC vs Phone Instruction Speed

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

Table 5.1. shows the amount of time a instruction took execute on a phone and PC for an average of 100,000 instructions in a row. This table also shows how much time the instructions took to execute compared with the most basic instruction.

The timing results also include the time it took for the loop to check as well.

The *s in the table represent the expected values since Java prevented us from acquiring the correct values at 100,000 iterations. The values we listed are our approximations based upon the values seen at lower iteration numbers. We do not know what Java did, but it prevented those three operations from being correctly timed.

5.2. Design

To take advantage of non-memory allocation instructions we implemented our own buffer pool and big integer system to remove the need for the large

	Execution time (ns)		Proportion increase	
Program	PC	Phone	PC	Phone
Simple recursion of depth 200	462	36454	28.9	1719.6
1 addition	16*	21.2	1	1
1 multiplication	21*	21	1.1	1
1 string addition of two characters	406	14947	25.4	705.1
Allocation of 1024 bytes	270	55438	16.9	2615
Allocation of 10240 bytes	1180	524595	73.8	24745
Allocation of 8 bytes	100*	3487	6.3	164.5
Creation of an 80 bit BigInteger	389	10334	24.3	487.5

TABLE 5.1.: 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. The *'s are our approximations based times at other iteration numbers.

number of allocation and deallocation steps for BigIntegers. We call our system *MemoryManagement*. The variables used in programs to hook into the memory are integers. The data in the memory itself we call *MMints*. We describe the MemoryManagement system below.

The MemoryManagement system is primarily a large integer array for the actual memory. There is also a circular queue to keep track of which spots in the memory are free. If the queue is ever empty then the memory is full.

Each variable which was previously a BigInteger in the original execution system is now an integer. These integers are the hooks into the MMints in the memory. For any MMint operation, the operation takes in a set of integers and preforms the operation on the set of corresponding MMints in memory.

For each insert in the program we also added a corresponding delete, or deallocation. Although this may seem like the memory system of a language like C, it is not since we do not deallocate the actual memory during each delete. This allows us to reuse the memory without a need for another allocation. We created

a set of functions for immutable operations and another for mutable operations for when mutable operations were applicable.

The downside to this implementation is that the MMints are of a fixed length. However, we have designed a method to take advantage of our system which would allow for a dynamic increase in the length of digits. We would use a second pool of memory as if it was a list of Inodes and then link multiple nodes together like they were the Inodes for a file system.

We also applied a few other memory optimizations we observed were possible during our conversion from BigIntegers to MMints. The primary change is the MemoryManagement system. Using the MemoryManagement system we also had a few ways to optimize the memory, such as primitive send and receive operations for network transmissions, which are made possible by our optimal system. We only applied our solution to the garbled circuit execution as opposed to oblivious transfers as well.

5.3. Evaluation

Table 5.2. shows the results of our MemoryManagement system applied to the Huang et al.'s system. The Table shows the results from our interpreter with the corresponding execution system. We had a speed increase during the execution phase by up to 4 times in the larger programs.

When we compared our interpreter to the custom circuits of Huang et al. we found even our optimized execution system was still slower than the custom circuits by a factor of 2 for the Levenstein distance. 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 apply our

	Execution time (ms)		
Program	Huang et al.	MMints	Percent 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 Dist 2	898	360	40.1
Levenshtein Dist 4	7105	2340	33.9
Levenshtein Dist 8	43999	11774	26.8
Levenshtein Dist 16	194067	48152	24.8

TABLE 5.2.: Comparison of the Huang et al.'s original execution phase and our own execution phase. Both execution systems use our interpreter.

MemoryManagement system to the hand created circuits and get a speedup since it would benefit from a better memory management strategy.

One of the more revealing results we experienced was the execution performance boost on a PC. The difference the MemoryManagement on a PC was negligibly faster or even slightly slower depending upon the input size used. This shows the optimizations which benefit the PC and phone are not the same.

CHAPTER VI

DISCUSSION

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. Lara and Kevin helped edit this Chapter. Kevin contributed ideas for what could be useful to create and talk about in this Chapter.

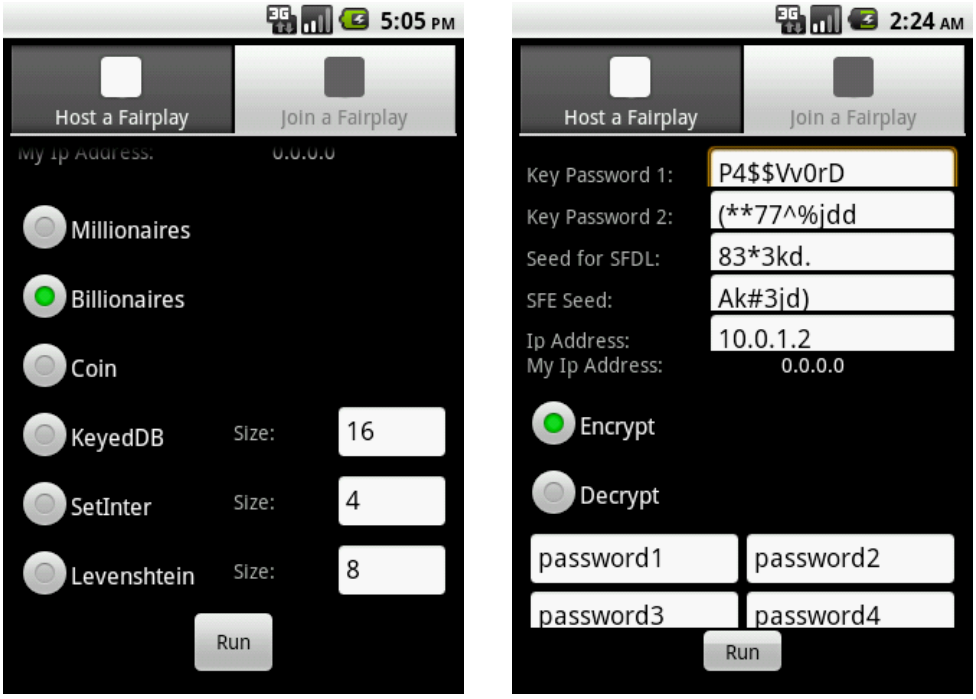
To demonstrate how our memory-efficient compiler can be used in practice, we developed Android apps capable of generating circuits at runtime. We describe these below.

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 6.1. (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 pseudo random



(a) (b)

FIGURE 6.1.: Screenshots of the GUI and password vault applications.

generator. 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 6.1.(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.

6.2.1. Security of the Password Vault

The security of the password vault application is dependent upon whether the output of the pseudo random 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 Garbled 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 n parameter increased the circuit size by a factor of about 4.5 (decreasing the larger n grows), when n is 8 there are 11,268 gates, 16 is 51,348 gates, 32 is 218,676 gates, and 64 is 902,004 gates.

The original PAL did not scale since it did not have loops, arrays, procedures, or functions. Once those programming structures were added the length of the PAL files were decreased dramatically. The resulting circuits generated from the new PAL were very similar to the original circuits.

6.4. Malicious Model

In a recent paper [10], Huang et al. implemented a process to achieve near malicious model security in SFE with only minimal changes. This is achieved by preforming the execution twice and the 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. They showed this process incurs a minimal throughput decrease when the execution is preformed on dual core machine. Most phone 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 "near" malicious model security instead of complete malicious model security. A party may gain a single bit of information they should not attain.

Kreuter et al. [13] implemented a system for preforming garbled circuits on cluster machines in the malicious model. They take garbled circuits to the limit by using state of the art execution systems and cluster machines.

6.5. Future Compiler Work

There are a number of optimizations that we would like to implement to create better circuits. Most notably free NOT and other gates which will always yield the same result. We are uncertain of the memory trade-off for these optimizations but it could be large depending on the size and structure of the program. For instance, using a NOT gate will then require inverting the table of a gate which used the NOT

gate's output. This will require keeping track of all gates which used a NOT gate, which may lead to a larger memory requirement. We also would like to add a way to keep track of the maximum value of a particular variable in the case, for instance, the full addition circuit is not needed.

Another optimization we plan to add is a way to deal with optimizing each part of the circuits. For instance, instead of having a static addition circuit we will add the ability to optimize a given addition circuit if one of the variables is a constant or if it is determined the maximum possible value of the variable will allow for a smaller addition circuit.

Given the large amount of time that is used to save and open complete circuit files we have speculated about combining our system with that of Huang et al.'s methodology of generating circuits directly in Java [9]. This would allow for programs to be compiled into PAL. Then a PAL interpreter could run the programs based on pre-built pieces. This system would allow users to write their own programs in SFDL and not have to worry about the memory limitations of SHDL.

CHAPTER VII

CONCLUSION

This Chapter was previously published in *Financial Cryptography and Data Security 2012*. The authors were Benjamin Mood, Lara Letaw, and Kevin Butler. This conclusion was edited by Lara and Kevin.

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 pre-generated 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 use of this compiler with a GUI editor and a password vault application. We interfaced our compiler with another execution system and then applied an optimization to that system specific to the mobile platform. Future work includes incorporating further optimizations in the circuit evaluator and determining whether the pre-generated templates may work with other approaches to both SFE and other privacy-preserving computation primitives.

APPENDIX

PROGRAMS

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};
  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};
  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;
    }
  }
}
```

REFERENCES CITED

[1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *International Crytology Conference*, 1990.

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

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

[4] J. Brickell and V. Shmatikov. Privacy-Preserving Classifier Learning. In *Proceedings of Financial Cryptography and Data Security*, Feb. 2009.

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

[6] 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.

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

[8] Y. Huang, P. Chapman, and D. Evans. Privacy-Preserving applications on smartphones: Challenges and opportunities. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec '11)*, Aug. 2011.

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

[10] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. *IEEE Symposium on Security and Privacy*, (33rd), May 2012.

[11] S. Jha, L. Kruger, and V. Shmatikov. Towards Practical Privacy for Genomic Computation. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 216–230, Berkeley, CA, USA, Nov. 2008.

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

[13] B. Kreuter, abhi shelat, and C. hao Shen. Towards Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[14] 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.

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

[16] D. Malkhi, N. Nisan, and B. Pinkas. Fairplay project, http://www.cs.huji.ac.il/project/fairplay/.

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

[18] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of SODA '01*, Washington, DC, 2001.

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

[20] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Proceedings of ASIACRYPT*, Tokyo, Japan, 2009.

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

[22] A. C.-C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.