Memory-Efficient Garbled Circuit Generation for Mobile Devices

Benjamin Mood, Lara Letaw, and Kevin Butler

Department of Compter & Information Science
University of Oregon, Eugene, OR 97405 USA
`{bmood,zephron,butler}@cs.uoregon.edu`

Abstract. Secure function evaluation (SFE) on mobile devices, such as smartphones, creates compelling new applications such as privacy-preserving bartering. Generating custom garbled circuits on smartphones, however, is infeasible for all but the most trivial problems due to the high memory overhead incurred. In this paper, we develop a new methodology of generating garbled 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 that generates circuits which can be evaluated with Fairplay. 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 for the set intersection problem reduced by 95.6% for the 2-set case. 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 high-level function specification.

1 Introduction

Mobile phones are extraordinarily popular, with adoption rates unprecedented in the history of product adoption by consumers. Smartphones in particular have been embraced, with over 296 million of these devices shipped in 2010 [4]. The increasing importance of the mobile computing environment requires functionality tailored to the limited resources available on a phone. 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.

Privacy-preserving computing is particularly well suited to deployment on mobile devices. For example, two parties bartering in a marketplace may wish to conveal the nature of their transaction from others, and share minimal information with each other. Such a transaction is ideally suited for *secure function evaluation*, or SFE. Recent work, such as by Chapman et al. [6], demonstrates the myriad applications of SFE on smartphones.

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

In this paper, we describe a memory-efficient technique for generating the garbled circuits needed to perform secure function evaluation on smartphones. While numerous research initiatives have considered how to *evaluate* these circuits more efficiently [16, 7], little work has gone towards efficient *generation*. Our port of the canonical Fairplay [12] 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. Combined with more efficient evaluation, our techniques provide a new arsenal for making privacy-preserving computation feasible in the mobile environment.

The rest of this paper is organized as follows. Section 2 provides background on secure function evaluation, garbled circuits, and the Fairplay SFE compiler. Section 3 describes the design of PAL, our pseudo assembly language, and our associated compilers. Section 4 describes our testing environment and methodology, and provides benchmarks on memory and execution time. Section 5 describes applications that demonstrate circuit generation in use, while Section 6 describes related work and Section 7 concludes.

2   Background

2.1   Secure Function Evaluation with Fairplay

The origins of SFE trace back to Yao's pioneering work on garbled circuits [18]. 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 $\boldsymbol{a} = a_0, a_1, \cdots a_{n-1}$ and $\boldsymbol{b} = b_0, b_1, \cdots b_{m-1}$ respectively, they wish to compute a function $f(\boldsymbol{a}, \boldsymbol{b})$ 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, *oblivious transfer* 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 that both parties are able to provide inputs. 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 perform 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. The Fairplay group is notable for creating the abstraction of a high-level language, known as SFDL. This language describes secure evaluation functions and is compiled 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. [14] 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.

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

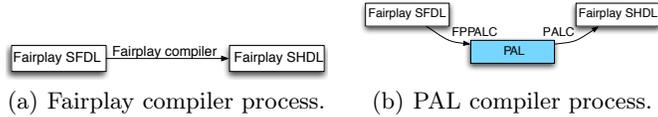(a) Fairplay compiler process.　　　(b) PAL compiler process.

Fig. 1: Compilation with Fairplay versus PAL.

unrolls variables, 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 examined the results of circuit compilation on a PC to determine the scope of memory requirements. From tests we performed on a 64-bit Windows 7 machine, we observed that Fairplay needed at least 245 MB of memory to run the compilation of the keyed database program, a program that matches database keys with values and employs SFE for privacy preservation (described further in Section 4). In order to determine the cause of this memory usage, we began by analyzing 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 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 is well-described by others considering secure function and secure multiparty computation, such as Kruger et al.'s OBDD protocol [10], Pinkas et al.'s SFE optimizations [16], the TASTY proposal for automating two-party communication [5], Jha et al.'s privacy-preserving genomics [8], Brickell et al.'s privacy-preserving classifiers [3] and Huang et al.'s recent improvements to evaluating SFE [6]. Similarly, we make the well-used assumption that parties enter correct input to the function.

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*

Possible Operations	
Operation	Syntax
Addition	DEST + V1 V2
Greater than or Equal to	DEST >= V1 V2
Equal to	DEST == V1 V2
Bitwise AND	DEST & V1 V2
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 1: PAL Operations

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 FP-PALC compiler, and generating the SHDL file which can then be run using Fairplay's circuit evaluator with our PALC compiler.

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. A full table showing all headings can be found in the full technical report [13] and is elided here because of space constraints.

Table 1 lists an abbreviated set of operations that are available in PAL along with their instruction signatures. The full set can be found in our technical report [13]. 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.

Note that conditionals can be reduced to the IF conditional. Unlike in regular programs, all parts of an IF circuit must be executed on every run.

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.

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 which has a bit length of 1, and Array[7] is an array of seven elements where each have a bit length of 5. All declared variables

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

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

Fig. 2: Example of variable declarations in PAL.

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

```
Variables:                          $c0 = $t0
i 6                                 out.a IF $c0 in.b[i].data out.a
in.a 6
in.b[16].data 24                    Instructions:
in.b[16].key 6                      in.b[16].data IN b
out.a 24                            in.b[16].key IN b
$c0 1                               in.a IN a
$t0 1                               DBsize = 16
DBsize 64                           i FOR 0 15
                                    $p0 PROC
Procedure: $p0                      out.a OUT a
$t0 == in.a in.b[i].key
```

Fig. 4: Representation of keyed database program in PAL.

are initialized to 0. After variable declarations, a PAL program can have function and procedure definitions preceding the instructions, which is the main function.

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.

To illustrate a full program, Figure 4 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, other than equality, were generated using simple Fairplay programs that represent equivalent functionality. Any operation that does not 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 as a real value. 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 performed 24-bit operands. This will not cause an error but may yield incorrect results if false assumptions are made.

There are currently 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.

3.3 FPPALC

To demonstrate the feasibility of compileing 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.

We note our compiler will not yield the same result 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. A more detailed explanation can be found in our technical report [13], To summarize, unoptimized constants in SFDL can cause the evaluator to crash, while programs consisting of a single `if` statement can produce inconsistent variable modifications. Apart from these differences, the generated circuits have equivalent functionality.

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 besides the parser. From the parser we built our own type system, 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.

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

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 how the circuits are created but on the way garbled circuit constructs work. Therefore, our circuits will provide the same security guarantees since our circuits also rely on using the garbled circuit protocol.

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 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 [11]. The other SFDL files that we have written can be found in the full technical report [13]. 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. We measured memory allocation and 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 FPPALC in this section.

Memory Usage & Compilation Time Table 2 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 2 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 compliation is related to the number of individual instructions.

Comparison to Fairplay Table 3 compares the Fairplay compiler with 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

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

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

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.

Circuit Evaluation Table 4 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. Described further below, we used both Fairplay's evaluator and an improved oblivious transfer (OT) protocol developed

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

by Nipane et al. [15]. Note that Fairplay's evaluator was unable to evaluate programs with around 20,000 mixed two and three input gates on the phone. This limit translates to 209 32-bit addition operations in our compiler.

While the circuits 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 5 shows the results of this evaluation. Programs denoted with a + required edits to the SHDL to run in the evaluator, in order to prevent their crashing due to the issues described in Section 3.3. 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 and discuss further in Section 5. Note, however, that these circuits are unable to be generated on the phone using Fairplay and require pre-compilation.

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. [15], who replace the OT operation in Fairplay with 1-out-of-2 OT scheme based on a two-lock RSA cryptosystem. Shown in Table 5, these provide an over 24% speedup for the Billionaire's problem and 26% speedup for the Coin Flip protocol. On average, there was an 13% decrease in evaluation time across all problems. 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 find-
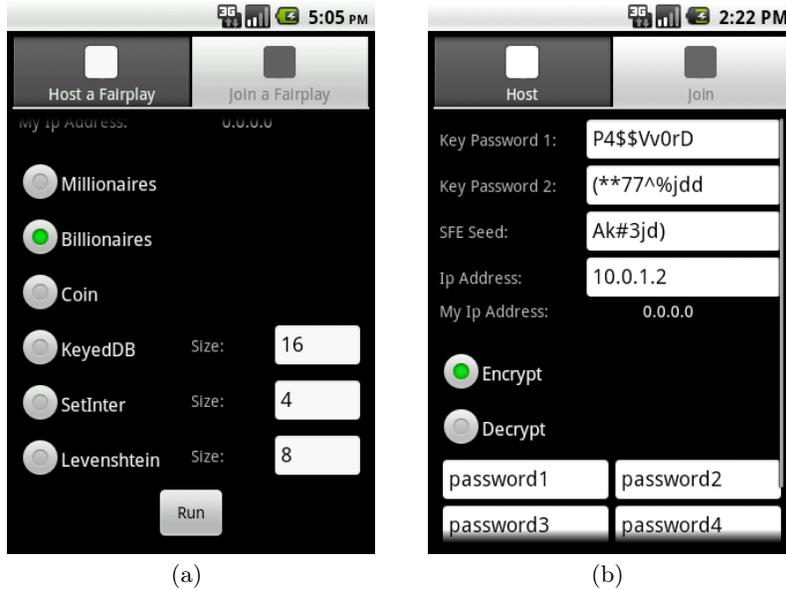
Fig. 5: Screenshots of editor and password wallet applications.

ings, however, are that our memory-efficient circuit generation is complementary to other approaches that focus on improving execution time and can be easily integrated.

5 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.

5.1 GUI Based Editor

To allow compilation on a phone we have to address one large problem. Our experience porting Fairplay to Android showed the difficulty of writing a program on the phone. Figure 5 (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.

5.2 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 inputs to a pseudorandom number generator. These passwords 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 and Bob can 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. Figure 5(b) shows a screenshot of this application. which can encrypt passwords from the user or decrypt those in the database.

Our evaluation shows that compiling the password SFDL program requires 915 KB of memory and approximately 505 ms, with 60% of that time involving 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.

5.3 Experiences with Garbled Circuit Generation

A major takeaway from our implementation was cognizance of 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 even if n statements are required.

The largest challenge for many other approaches 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 paremeter 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.

Our original version of PAL did not scale well due to its lack of loops, arrays, procedures, or functions. Once these structures were added, the length of PAL programs decreased dramatically. Instead of unrolling all programming control flow constructs we added them for smaller PAL programs, creating largely identical circuits with much fewer instructions.

6 Related work

Other research has primarily focused on optimizing the actual evaluation for SFE, while we focus on generating circuits in a memory efficient manor. Kolesnikov et al. [9] demonstrated a "free XOR" evaluation technique to improve execution speed, while Pinkas et al. [16] 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. [7] 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. [17] 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. [10] described a way to use ordered binary decision diagrams (OBDDs) to evaluate SFE, which can provide faster execution for certain problems. Our future work will involve determining whether the process of preparing OBDDs can benefit from our memory-efficient techniques. TASTY [5] 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.

7   Conclusion

We introduced a memory efficient technique for making SFE tractable on the mobile platform. We created PAL, an intermediate language, between SFDL and SHDL programs and showed that by using pre-generated circuit templates 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. Future work includes incorporating 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.

Acknowledgements

be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

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. J. Brickell and V. Shmatikov. Privacy-Preserving Classifier Learning. In *Proceedings of Financial Cryptography and Data Security*, Feb. 2009.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. D. Malkhi, N. Nisan, and B. Pinkas. Fairplay Project, http://www.cs.huji.ac.il/project/Fairplay/.
12. 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.
13. B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. Technical Report CIS-TR-2011-04, Department of Computer and Information Science, University of Oregon, Eugene, OR, USA, Sept. 2011.
14. M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *Proceedings of SODA '01*, Washington, DC, 2001.
15. 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.

16. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *Proceedings of ASIACRYPT*, Tokyo, Japan, 2009.

17. 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/.

18. 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.