RockEngine

Building small user-friendly boolean circuits

BENCHMARKING COMPUTATION ON PRIVATE DATA AGAINST OTHER OPEN SOURCE FRAMEWORKS

1. ABSTRACT

In this paper, we introduce *RockEngine Generic Privacy builder,* a Boolean circuit compiler meant to be part of the RockEngine Generic Privacy Engine developed by Rockchain.org. The *RockEngine Generic Privacy builder* is a complete garbled circuit generator that will be used in particular on the public network of Rockchain.org to perform Secure Function Evaluation on private dAppBoxes nodes.

The RockEngine Generic Privacy builder aims at being simple to use while providing performances comparable to state-of-the-art compilers which are currently developed for research purposes. It will be completed by other applications to perform garbling or execution of the system. Nonetheless, the *RockEngine Generic Privacy* has not been created to be used solely inside Rockchain's distributed infrastructure. It provides various features, and the circuits produced could easily be used by any developer, scientist or researcher. Its scripting language simplicity and genericity (JavaScript) abstracts all the difficulties of secured multipart computation circuits' designs away.

WHAT IS ROCKCHAIN?

Rockchain provides a second layer infrastructure on top of the Ethereum Blockchain, enabling data privacy on top of Ethereum, either by controlling access rights to distribute data, on each document which fields can be publicly disclosed, in computations what results can be broadcasted. Its technical distributed infrastructure is made of the DappBox, a

distributed file system, the RockEngine, whose abilities to perform computations on private data are described in this paper, and the DashRock, a real time reporting solutions on Ethereum smart contracts.

2. INTRODUCTION

Garbled circuits are today the most flexible solution to perform secure function evaluation (SFE), i.e. computation of a function whose inputs come from different parties which are not willing to share their data with one another.

The representation of the function as a Boolean circuit is a must-have condition for SFE to work. The first principle of garbled circuits' algorithms is to have a high-level representation of a pre-compiled function into a clear Boolean circuit.

The first party, Alice, turns this clear circuit into a garbled circuit and encrypts her own input. The second party, Bob, uses an oblivious transfer to get his encrypted input from Alice. They can finally evaluate the function: Alice sends the garbled circuit and her encrypted input to Bob so that he can perform the computation.

The garbling and execution of the circuit are usually done in a single step since traditional garbled circuits cannot be safely reused. Every time we need to evaluate a function, we use the same Boolean circuit to create a new garbled circuit with new keys.

The construction of Boolean circuits beings a necessary step, for optimization matters we are seeking optimal Boolean circuits, defined as the smallest circuits able to fully represent the given function, in order to minimize the computation time on both sides and the bandwidth used in transmitting the circuit.

As garbled circuits are still the subject of a booming and yet structuring research domain, there is still no comprehensive and user-friendly solution available to generate, optimize and use garbled circuits. Rockchain's RockEngine Generic Privacy builder is focusing on solving this issue while keeping the developers minds' sanity. In consideration, there will be no turn back to designing Boolean circuits with logical gates. The RockEngine compiles simple Javascript libraries into optimized Boolean circuits, without the developer having to trick is programming skills to understand underneath optimization tricks.

RockEngine Generic Privacy has both a compiler that generates the Boolean circuits and a runner process (RockEngine.GenericPrivacy.run).

3. STATE OF THE ART

Several solutions already exist to create Boolean circuits. There are namely:

- **Fairplay, 2004** [8], which was the first practical implementation of a generic Boolean circuit compiler and inspired subsequent papers later. It uses a custom C-like language as input: SFDL.
- **TASTY, 2009** [3], it contains tools for both garbled circuits and homomorphic encryption. It uses the Free XOR and row reduction techniques which have become a standard optimization in garbled circuit framework implementations.
- **PAL, 2012,** it uses SFDL, too, with a considerable number of optimizations compared to Fairplay.
- **CBMC-GC, 2012** [4], it uses ANSI C as input and outputs circuits in a text format. This gives it a compatibility advantage, but it generates large output files.
- **KSS, 2012** [5], authors built their garbled circuit software to show that some evaluations of billion-gate circuits are feasible in the malicious model (where specifically the sent circuit could not be the right one in a hacker attempt to alter computation results).
- **PCF, 2013** [6], which has LCC as input format and outputs circuit using a compact format.
- **OblivVM, 2015** [7], which compiles inputs into Java files and then uses the Java virtual machine optimizer.

- **OblivC**, 2015 [11], aims at providing an almost exhaustive range of programming tools to compile Boolean circuits, like oblivious RAM, while remaining accessible to standard developers.

- **TinyGarble, 2015** [10], its compiler is an auxiliary application which generates circuits from verilog files. The TinyGarble executable is an improved version of the program JustGarble, which solely performs garbling (i.e. encryption of Boolean circuits).
- **Frigate, 2016** [9], which focuses on overcoming problems of speed and correctness encountered by previous compilers.
- **CBMC-GC v2.0, 2017** [2], is an optimized version of the software compiler released in 2012. It outputs less gates than before, and its extension ShallowCC provides "depth reduction" in the circuit.

Coding for Boolean circuits is particularly compelling in several ways that have not been overcome yet. For example, the circuit must be bounded, which implies that the number of iterations for a loop must be decided at compilation time.

Regarding the compilation of the circuit, the most important property is the size of the produced circuit. We can break down into these four parameters:

1. the number of XOR gates, which are quick to evaluate during the execution,
2. the number of non-XOR gates, which are more resources consuming,
3. the number of wires used, which will have a direct impact on the memory used by the machine performing the later evaluation of a garbled circuit,
4. the size of the line containing the circuit, which depends on the number of gates and the format used to encode the circuit.

When compiling Boolean circuits there is no possibility of hardware-based optimization as during a classical compilation. This is why the compilation of Boolean circuit operates on an abstract layer and is usually much slower; the speed factor is also relevant to compare different compilers.

In [9], Frigate's authors review PAL, KSS, CBMC, PCF, Obliv-C and OblivM. They highlight several flaws and compare them to their own solution. Frigate appears to be significantly faster than other solutions with a compile time improved up to 447x compared to the best previous results, while producing circuits with similar gate counts. Frigate also adds its own new functionalities.

The latest compiler released, CBMC-GC v2.0, focuses on reducing the number of gates and the depth of the circuit. It starts by writing the circuit as first time before performing an optimization phase. In [1], several authors of CBMC-GC benchmark their new version by comparing it to Frigate, TinyGarble and Obliv-C. They show significant improvements for some algorithms in terms of the total number of gates used and the size of the files produced.

In the remainder, we will compare our solution to Frigate and CBMC-GC when it is possible, given that those two compilers currently appear to be the most efficient and user-friendly existing solutions.

3. USE CASES

In a distributed infrastructure context, using Garbled circuits allow the perform search queries on distributed encrypted data silos. The algorithm encrypted as a garbled circuit is merely a string comparison or a distance computation between strings.

In a distributed infrastructure context always, along with secret sharing techniques on public nodes, it is possible to perform machine learning algorithm on a distributed network of private datasets.

Computing some algorithms on private datasets while only revealing the authorized computed outputs bring some new opportunities on models governed by a consensus: average-based group decision making, outlier detection without revealing datasets. We can think of improving house prices indexes, energy prices indexes.

Distributed governance, providing they get the correct tooling to extract data patterns without centralizing data, creates a new kind of regulatory practices that do not rely on a human trusted behavior. Data producers are controlled without information leaking about their strategic data. It can apply to many fields in finance.

In the health and bioscience area, privacy is a must that must not prevent quality datascience to be performed. By providing robust and fast matrix operations on private data, and controlling the output results, the RockEngine ensure numerous patient files analysis without compromising his right to privacy.

4. TECHNOLOGY

Rockchain's *Generic Privacy Engine* focuses mostly on usability, so that anyone could operate on a circuit without going himself through the decoding of a binary file or using a garbling specific language. The RockEngine is built upon the Go language efficiency optimizations for files encoding, and uses JavaScript as an input language to maximize the number of people able to use it.

4.1 COMPILING SCRIPTS FROM JAVASCRIPT INPUT FILES

JavaScript as an input language is the first strong point of RockEngine Generic Privacy builder: a program written to be used in a garbled circuit could also be compiled by any traditional JavaScript compiler. Indeed, DataScript is included in JavaScript excepted for a few specific functions which enable more efficient wire level operations.

Let us realize a quick syntax comparison between GPE.build, Frigate and CBMC-GC.

For each of the three compiler we provide a basic code to resolve Yao's millionaires problem. This problem, which is at the origin of garbled circuits is the following: two millionaires argue about which one of them is the richest, but they don't want to tell each other the exact amount of money the own.

This is typically a case where Secure Function Evaluation is useful. In this example it can be done with a simple comparison operator.

Listing 1: Millionaires' problem with GPE.build

```
var $parties = 2
var $intsize = 16

var in_0 = 0
var in_1 = 0
var out_0 = in_0    > in_1
```

Listing 2: Millionaires' problem with Frigate

```
#define wiresize 32
#parties 2

typedef int_t wiresize int
typedefuint_t 1 bool

#input 1 int
#input 2 int
#output 1 bool

function void main() {
    output1 = input1 > input2;
}
```

Listing 3: Millionaires' problem with CBMC-GC

```
#include <inttypes.h>

typedef int32_t InputA;
typedef int32_t InputB;
typedef uint8_t Output;

Output mpc_main(InputA INPUT_A, InputB INPUT_B) {
    return INPUT_A > INPUT_B;
}
```

RockEngine Generic Privacy builder uses a "convention" syntax such that each input or output is prefixed by either in_x or out_x where x is the index of the involved party associated to this variable (either as a private data input provider, or as a computation result consumer).

In the current version of Rockchain Engine Generic Privacy Engine, the initialization of all variables before any computation is mandatory. Since JavaScript is not strongly typed, the RockEngine compiler has no other clue of each variable type than its initialization type.

In case of input variables, there is no effective assignment due to the initialization, as the initialization is merely a convention to define IN/OUT interfaces with the underlying Boolean circuit.

4.2 ROCKENGINE CIRCUIT'S USABILITY AND REUSABILITY

Our RockEngie provides a wide amount of scripting possibilities thanks to the Javascript support. Our compiler detects code patterns that are not compliant with the distribution of generated garbled circuits in a distributed multi-part computation environment. This can happen when the script has no predictable output, with conditional loops with external parameters used for limits, and so one. The direct interaction of the compiler with the Javascript developer in a programming editor makes it easy to understand the programming constraints provided by the need to have distributed computation.

Using the RockEngine Privacy Engine requires the following actions:

1. download the RockEngine.GenericPrivacy.build repository,
2. go to the RockEngine.GenericPrivacy.build folder and compile the main executable file with the following command

   go install
3. go back to your gopath (where you installed Go) and compile your first circuit like this:

   ./bin/RockEngine.GenericPrivacy.build/pathToMyFile/myFile.js

The last command line will output a single file with a *freeg* extension in the folder of your JavaScript line. We optimized the usability of the output file compared to other approaches generating several files, one file per function, as well as a statistics file for Frigate. In the CBMC-GC for example, nine text files are systematically generated along with the C++ executable code, representing the Boolean circuit.

The RockEngine Generic Privacy builder is designed for an optimal inter-operability with other GO programs. Any programmer can simply reuse any pre-compiles RockEngine Generic Privacy circuits with the following commands:

```
import      "github.com/AlphaDinoRC/RockEngine_GenericPrivacy.build/compiler";
var circuit pc.Circuit = compiler.CircuitFromJS("myFile.js");
```

The resulting circuit variable will contain the whole resulting circuit.

Similarly, if someone wants to create another Go program and needs to open a circuit stored as a *.freeg* file he simply has to use the following commands:

```
import      "github.com/AlphaDinoRC/RockEngine_GenericPrivacy.build/plainCircuit";
var circuit pc.Circuit = compiler.RetrieveFromFile("myFile.freeg");
```

5. THE BENCHMARK

5.1 Test panel

To assess the efficiency of the three selected algorithms we use a test-panel composed of standards algorithms: multiplication of large integers and multiplication of matrices.

The detail of those algorithms is as follows:

Algo. 1 16x16 matrices multiplication with 64 bits integers,

Algo. 2 32x32 matrices multiplication of 64 bits integers,

Algo. 3 64x64 matrices multiplication of 64 bits integers,

Algo. 4 256 bits integer multiplication,

Algo. 5 1024 bits integer multiplication

In the next section we only compare RockEngine Generic Privacy builder with Frigate given that CBMC-GC is not able to perform the required computations within a reasonable time limit. To compile the first algorithm alone, the time required is more than an hour. Moreover, it only accepts standard C types, which means that we cannot use integers of arbitrary length and algorithms 4 and 5 cannot be implemented.

5.2 Test results

Speed performances We analyze the speed of the initial circuit compilation (the build time).

The benchmark results are given in table 1.

Algorithm	RockEngine	Frigate
Algo. 1	0.159 s	0.344 s
Algo. 2	1.534 s	2.459 s
Algo. 3	18.63 s	18.36 s
Algo. 4	0.181 s	0.022 s
Algo. 5	2.51 s	0.179 s

Table 1: Compilation speed compared to Frigate

Testing machine used:

- CPU: Intel Core i5-6200U (Dual Core - 2.3 GHz / 2.8 GHz Turbo - Cache 3 MB - TDP 25W)
- RAM DDR4 8 GB, 2133 MHz
- SSD 120 GB, M.2 - SATA 6GB/s
- OS: GNU Linux x86_64, kernel 4.10.0-32-generic, Xubuntu 17.04

We can observe that Frigate in its current version is efficient for matrix calculus but not very efficient yet compared to Frigate in sheer unknown big numbers multiplication.

Optimization of the output circuit

We measure the quality of the output of the three compilers according to the criteria defined in 2.

File size First we compare the sizes of the files produced which contain the circuits. The results are given in table 2. On that point it appears that RockEngine brings significant improvements compared to Frigate, in particular for matrices multiplication, which are frequently used in classification algorithms or data reduction techniques.

Algorithm	GPE.build	Frigate
Algo. 1	439 kb	3.4 Mb
Algo. 2	2.3 Mb	13.4 Mb
Algo. 3	14.1 Mb	55.3 Mb
Algo. 4	2.9 Mb	3.1 Mb
Algo. 5	46.7 Mb	50.3 Mb

Table 2: Compared size of the output on the test pane

Algorithm	RockEngine	Frigate
Algo. 1	66 050	147 630
Algo. 2	262 722	589 998
Algo. 3	1 049 154	2 359 476
Algo. 4	2 050	2 560
Algo. 5	8 194	10 240

Table 3: Compared numbers of wires used for each test

Number of wires used The second factor that we measure is the number of wires used. Those results are given in table 3. As we can see, the RockEngine Generic Privacy builder has been optimized in order to minimize those numbers, which are significantly lower than Frigate's.

Number of gates The number of gates is the most important factor to measure as it determines the number of operations to be transmitted and performed during the execution. This is also the most commonly used indicator.

The results are given in table 4. We can observe that RockEngine Generic Privacy builder and Frigate are generally similar on gate counts.

Algorithm	GPE.build		Frigate	
	XOR	Non-XOR	XOR	Non-XOR
Algo. 1	34 099 200	16 785 410	33 492 993	16 785 412
Algo. 2	272 793 600	134 283 266	267 157 505	134 283 268
Algo. 3	2 181 824 512	1 074 266 114	2 134 114 305	1 074 266 116
Algo. 4	130 052	65 285	131 333	65 285
Algo. 5	2 093 060	1 047 557	2 098 181	1 047 557

Table 4: Compared numbers of gates used for each test

6. CONCLUSION

As we saw in section 5, the first version of RockEngine Generic Privacy builder succeeded at providing performances comparable to one of the most recent and efficient garbled circuit compiler, Frigate.

On some matrixes operations, we are faster. Frigate is by far the fastest compiler available in open source. However we are improving memory usage significantly, which is important when considering that a garbled circuit must be generated for each usage, consuming a lot of memory for dataset independent usages.

The RockEngine Generic Privacy Engine is setting a high standard in usability through its Javascript input file definition, simplify the integration of SMC algorithms in any Go language by integrating smoothly as a package, and is simplifying the complexity of generated files for each Boolean circuit.µ

Our Research focus is now targeting on including new garbled circuits researchs in our framework, dedicated to be open source.

7. BIBLIOGRAPHY

[1] *Niklas Buescher et al. On Compiling Boolean Circuits Optimized for Secure Multi-party Com-putation. 2017.*

[2] *Martin Franz et al. "CBMC-GC: An ANSI C Compiler for Secure Two-Party ComputationsÆ". In: Compiler Construction: 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Vol. 8409. Springer. 2014, p. 244.*

[3] *Wilko Henecka et al. "TASTY: tool for automating secure two-party computations". In: Pro-ceedings of the 17th ACM conference on Computer and communications security. ACM. 2010, pp. 451–462.*

[4] *Andreas Holzer et al. "Secure two-party computations in ANSI C". In: Proceedings of the 2012 ACM conference on Computer and communications security. ACM. 2012, pp. 772–783.*

[5] *Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. "Billion-Gate Secure Computation with Malicious Adversaries." In: USENIX Security Symposium. Vol. 12. 2012, pp. 285–300.*

[6] *Benjamin Kreuter et al. "PCF: A Portable Circuit Format for Scalable Two-Party Secure Com-putation." In: USENIX Security Symposium. 2013, pp. 321–336.*

[7] *Chang Liu et al. "Oblivm: A programming framework for secure computation". In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE. 2015, pp. 359–376.*

[8] *Dahlia Malkhi et al. "Fairplay — a secure two-party computation system". In: USENIX Secu-rity Symposium. 2004, pp. 287–302.*

[9] *Benjamin Mood et al. "Frigate: A Validated, Extensible, and E cient Compiler and Inter-preter for Secure Computation". In: IEEE European Symposium on Security and Privacy. Mar. 2016.*

[10] *Ebrahim M Songhori et al. "Tinygarble: Highly compressed and scalable sequential garbled circuits". In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE. 2015, pp. 411–428.*

[11] *Samee Zahur and David Evans. "Obliv-C: A Language for Extensible Data-Oblivious Com-putation."* In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1153.