Opportunities for Genetic Improvement of Cryptographic Code

Chitchanok Chuengsatiansup, Markus Wagner, Yuval Yarom
The University of Adelaide
Adelaide, Australia

ABSTRACT

Cryptography is one of the main tools underlying the security of our connected world. Cryptographic code must achieve both high security requirements and high performance. Automatic generation and genetic improvement of such code are underexplored, making cryptographic code a prime target for future research.

ACM Reference Format:
Chitchanok Chuengsatiansup, Markus Wagner, Yuval Yarom. 2022. Opportunities for Genetic Improvement of Cryptographic Code. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3520304.3534049

1 INTRODUCTION

With the proliferation of computers into all aspects of human life, the amount of sensitive data being processed keeps increasing. As cryptography is one of the main tools underlying the security of our modern and connected world, cryptographic software must meet not only high security requirements, but also exhibit excellent non-functional properties, such as high performance and low energy consumption. Hence, we see cryptography as a prime target domain for single- and multi-objective code optimization.

Let us consider the hardening of code against implementation attacks. Such attacks exploit the interaction of software with the environment it executes in, and they pose a significant threat to the security of cryptographic software. For example, side-channel attacks [5, 10] observe effects such as power consumption, electromagnetic emanations, or even state change in the processor microarchitecture, to leak sensitive information, completely undermining the security of the software. To mitigate the risk of side-channel attacks, cryptographic engineers have developed a range of techniques. These, typically, restrict software development to only use safe constructs. For example, *constant-time programming*, a programming paradigm that protects against microarchitectural attacks, does not allow branches whose condition depends on secret data. Another example is the use of masking to prevent leakage through power consumption. With masking, each internal value of the algorithm is represented by two or more variables, such that observing subsets of these variables reveals no information on the secret internal value. Multiple tools have been proposed to assist cryptographers in developing cryptographic software, validate it, and verify that it meets the required security guarantees.

The net result of restricting programming constructs, using automated tools, and of the desire to support formal verification is that modern cryptographic software tends to have a relatively simple control-flow: conditional statements are often eliminated, loops are unrolled, and functions are often implemented as a linear sequence of arithmetic operations. Similarly, implementations tend to have simple memory access patterns, avoiding aliasing or excessive pointer manipulations.

We believe that such lowered complexity creates opportunities for using genetic improvement [13] in cryptographic software. In this brief position statement, we outline a few ideas of how this can be achieved: we focus on code that is resilient to power analysis attacks in Section 3 and on optimizing performance in Section 4.

2 RELATED WORK

Optimizing non-functional code properties encompasses many techniques and is known under different names in different fields. For example, software automatic tuning [11] deals with enabling software adaptation to the problem or the computing environment. Much of the development of the domain was in the context of highly parallel supercomputers. Hence, significant effort has been invested in tuning parameters for parallelization [7].

For generating optimized code, Frigo [4] combine short code templates to generate tuned programs for performing the FFT. Alternatively, Shackleton [12] uses genetic search to find the best sequence of compiler optimization passes for a target program.

Kri and Feeley [9] use genetic search for compiling code. Specifically, they model register allocation and instruction scheduling as a search problem and use genetic algorithms to search for a high performance solution. They demonstrate a performance improvement of 10–50% over the non-optimized version of the code. Unfortunately, they do not compare against optimized code.

While this is by no means a comprehensive review, the automatic generation of cryptographic code that performs well given multiple criteria is underexplored. We aim to change this.

3 DEFENSES FOR POWER ANALYSIS

The power consumption of a computing device correlates with the values it processes. To protect against power analysis attacks [8] software can *mask* intermediate values. For example, in order-p Boolean masking, an internal value v is represented using $p + 1$ values, $v_0, \ldots, v_p$. The values $v_0, \ldots, v_{p-1}$ are chosen uniformly and independently at random, whereas v_p is computed using $v_p = v_0 \oplus \cdots \oplus v_{p-1} \oplus v$. Observing any combination of p values out of $v_0, \ldots, v_p$ does not reveal information on the value of v [6].

Although theoretically secure, masking can fail in practice due to unintended interactions between values in the processor [1]. To protect against such leakage, Rosita [15] automatically identifies leaky instructions and applies code transformations to prevent the leakage. While effective at eliminating leakage, the security Rosita offers comes at a performance cost. Protecting code can add up to 190% to the code running time [14].

One of the reasons for the slowdown that code protected by Rosita incurs is that Rosita considers each leak in isolation rather than looking at the problem in a holistic way. As a result, it misses options to apply countermeasures that prevent multiple leakage points, opting instead for local fixes. To improve the performance of protected code, we propose to view the problem of which transformations to apply as a multi-objective combinatorial optimization problem that aims to reduce both leakage and running time.

4 CRYPTOGRAPHIC CODE OPTIMIZATION

Optimizing cryptographic code requires both skill in implementation and knowledge in cryptography. Most speed records in the past have been achieved by experts who manually implemented the schemes. Nonetheless, we believe genetic algorithms are suitable for optimizing cryptographic code. The reason is that cryptographic code tends to display a lower complexity than general programs, where significant parts of the code are in basic blocks without any control flow. Mainstream compilers tend to use windowing methods that are less effective for long basic blocks. At the same time the simplicity of control flow reduces the analysis efforts and simplifies searching the space of possible implementations.

Bosamiya et al. [2] apply GI to optimizing cryptographic code. Specifically, they start from assembly code that implements a cryptographic primitive. They use a genetic algorithm to apply transformations to the code in the search for a fast implementation. Finally, they prove that the resulting code is equivalent to the original code. They report performance improvement of up to 27% over the original code. Starting from assembly code makes it difficult to change register allocation and spill decision, limiting of their approach.

Our approach to improve the performance of cryptographic code generation is similar in the sense that we adapt combinatorial optimization and treat code generation as a search problem. However, our strategy eliminates their limitation by starting the optimization process directly from the *intermediate representation* instead of assembly code. This way, we can directly optimize register allocation and memory spills.

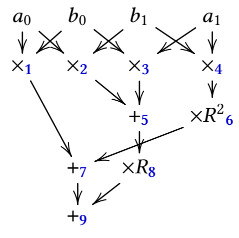

Figure 1: Data Flow Graph of Schoolbook Multiplication

An outline of our strategy is as follows. We start with an abstract representation of the code as a data flow graph, with nodes being operations and edges representing values. This representation could, for example, be obtained from a developer, extracted from an existing implementation, or generated by a tool such as Fiat cryptography [3]. Figure 1 shows an example of such a graph that implements a 2-digit by 2-digit schoolbook multiplication. Numbers are represented as digits in some radix R, and we want to multiply $a = a_0 + Ra_1$ by $b = b_0 + Rb_1$.

From this abstract representation we generate an initial arbitrary implementation. Then, we "mutate" the code, measure its performance, and keep a better version between the original and the mutated ones. We repeat these processes for a certain pre-defined computation budget. At the end of the process, we obtain a best-performing implementation.

While the outline of the processes seems to be simple, there are several challenges that we need to address. We need to transform abstract operations into code. To do so, we define templates to match operations with available instructions. Once we have an initial implementation, we can now perform mutations to move to an improved implementation.

We consider three types of mutations: using different available instructions, rescheduling instruction sequences, and changing decision of register to "spill" to memory. The first is achieved by using multiple possible templates for each operation. A change of a template is then a mutation. For the second, we move an operation forward or backwards, while observing the implied dependencies between operations. Finally, when the code uses more values than the number of available registers, there is a need to store some values in memory. Thus, the third type of mutation chooses a different register to store to memory. With this change, there is a need to fix all future references to the spilled value.

5 CONCLUSIONS

To ensure security, cryptographic code tends to use simple control flow and memory access patterns. This simplicity creates an opportunity for using GI for performance improvements.

REFERENCES

[1] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2015. On the Cost of Lazy Engineering for Masked Software Implementations. In *CARDIS*. 64–81.

[2] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. 2020. Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language. In *VSTTE*. 106–123.

[3] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE SP*. 1202–1219.

[4] Matteo Frigo. 1999. A Fast Fourier Transform Compiler. In *PLDI*. 169–180.

[5] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27.

[6] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*. 463–481.

[7] Herbert Jordan, Peter Thoman, Juan Jose Durillo Barrionuevo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A multi-objective auto-tuning framework for parallel codes. In *SC*.

[8] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*. 388–397.

[9] Fernanda Kri and Marc Feeley. 2004. Genetic Instruction Scheduling and Register Allocation. In *SCCC*. 76–83.

[10] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks - revealing the secrets of smart cards.* Springer.

[11] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda (Eds.). 2010. *Software Automatic Tuning From Concepts to State-of-the-Art Results.* Springer Science+Business Media.

[12] Hannah Peeler, Shuyue Stella Li, Andrew N. Sloss, Kenneth N. Reid, Yuan Yuan, and Wolfgang Banzhaf. 2022. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework. arXiv 2201.13305.

[13] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432.

[14] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. 2021. Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In *CCS*. 685–699.

[15] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2021. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *NDSS*.