EvoIsolator: Evolving Program Slices for Hardware Isolation Based Security

Mengmei Ye(✉), Myra B. Cohen, Witawas Srisa-an, and Sheng Wei

Department of Computer Science and Engineering,
University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA
{mye,myra,witty,swei}@cse.unl.edu

Abstract. To provide strong security support for today's applications, microprocessor manufacturers have introduced hardware isolation, an on-chip mechanism that provides secure accesses to sensitive data. Currently, hardware isolation is still difficult to use by software developers because the process to identify access points to sensitive data is error-prone and can lead to under and over protection of sensitive data. Under protection can lead to security vulnerabilities. Over protection can lead to an increased attack surface and excessive communication overhead. In this paper we describe EvoIsolator, a search-based framework to (i) automatically generate executable minimal slices that include all access points to a set of specified sensitive data; and (ii) automatically optimize (for small code block size and low communication overhead) the code modules for hardware isolation. We demonstrate, through a small feasibility study, the potential impact of our proposed code optimizer.

Keywords: Software transplantation · Genetic algorithms
Hardware security

1 Introduction

Hardware isolation is growing as a way for software developers to secure sensitive program calculations and data. For instance, customized secure chips are being used to store fingerprints and payment data on mobile phones. Isolation must include not only the data, but the code that accesses that data to avoid leakage of sensitive information. One popular isolation technique that we will work with in this paper is the ARM TrustZone [1]. Sensitive data and associated program code are stored in the *secure world* while the rest of the code is placed in the *normal world*. Code in the secure world can access data in both environments, while the normal world cannot directly query the secure world.

While this type of isolation provides stronger security than traditional software only approaches, there are some potential pitfalls. A bug in the secure world can cause significant harm by leaking or corrupting sensitive information. This argues for placing only a limited amount of (well tested) code into the secure world. There is also communication overhead between the secure and normal

T. E. Colanzi and P. McMinn (Eds.): SSBSE 2018, LNCS 11036, pp. 377–382, 2018.
https://doi.org/10.1007/978-3-319-99241-9_24

worlds. This suggests that different slices of code, and different code interleavings can impact the performance of such an architecture.

Recently we developed TZSlicer [2], a technique that uses slicing and a taint analysis to carve a program for use in hardware isolation. TZSlicer, while a good first step, over approximates the amount of the code that needs to be isolated in the secure world. We also found the need to add optimizations. For example, statements that do not access sensitive data might be interwoven with statements that do. This will lead to the inclusion of statements that do not access sensitive data into the secure world. On the other hand, if we separate the sensitive and non-sensitive statements into their respective spaces, it will result in fewer statements in the secure world but may also incur a significantly higher communication overhead. It is desirable to develop an optimization approach, which can reorder the statements to achieve security-aware program slicing with low communication overhead.

In this paper we present our vision of a more flexible framework for hardware isolation, using a search-based approach [3,4] that can balance different objectives. Our framework applies evolutionary algorithms in two phases. First, we propose to create a secure slice and synthesize it into the secure world. We view this as a type of software transplantation (where we remove code from the normal world and place it into the secure world) [4,5]. Our goal is to include the smallest slice that passes a security test suite. Second, we use another evolutionary algorithm (a genetic algorithm) to re-order and optimize the synthesized code within the secure world, with the goal of reducing communication overhead. We call our framework EvoIsolator. To the best of our knowledge, this is the first search-based approach to hardware isolation for program security.

While our vision is not yet fully implemented, we present our idea and motivating examples in this paper, along with a feasibility study to demonstrate how the second part of the framework, the optimization to reduce communication overhead, can improve performance.

2 Background and Motivation

ARM TrustZone. The ARM TrustZone mechanism is a widely used security platform to prevent against threat models such as information leakage attacks [1,6]. The secure world and the normal world in the TrustZone framework are separated by a bus-level hardware isolation interface. The communication between the secure and normal worlds is conducted by a secure monitor in the secure world. In addition, there are secure apps used to execute the sensitive programs and a secure memory space to store the sensitive information. The normal world contains a normal app to execute non-sensitive programs and a shared memory space to store information that both the secure and normal apps can access.

In our hardware isolation framework based on TrustZone [1], the normal app first sends a request to the secure world and provides input data for the shared memory. Then, the secure monitor issues a secure monitor call (SMC) to switch

the CPU mode from the normal to the secure world. The secure app conducts computations for each request after reading the input data stored in the shared memory and writes the end results to the shared memory for the normal app to access prior to switching the CPU mode back to the normal world. Note that resources stored in the secure world are treated as a part of the trusted computing base (TCB), and any faults or security vulnerabilities in the secure world can compromise the entire system [7].

TZSlicer. TZSlicer [2] uses a dynamic taint analysis to slice a small part of the program into the TrustZone framework that meets the security requirements and maintains the original program functionality. The developer provides an original program, the input data, and the tainted (secure) variables to TZSlicer. Then, TZSlicer generates a system dependency graph (SDG) [8] and extracts the propagation flow for the sensitive computations. It then slices the program, synthesizes the secure and normal slices, and deploys them into the TrustZone system. TZSlicer then attempts to optimize the slices using loop unrolling and variable renaming. However, the applicability and the room for optimization by adopting these simple strategies are limited [2]. We believe search-based techniques can help develop more applicable and effective optimization strategies including code reordering, demonstrated in this work.

3 EvoIsolator

Figure 1 shows our vision of EvoIsolator. It has two primary optimization steps. First, it determines and synthesizes the slice (TZSurgeon). Then, it optimizes that slice for performance (TZOptimizer). EvoIsolator starts with the original program and a set of sensitive variables. It generates random initial secure and normal slices that pass a security test suite. It then transplants the sensitive computations into the secure world and the remaining non-sensitive computations into the normal world using genetic programming to find the best code configuration. In TZOptimizer it reorders the code to reduce communication overhead.

The EvoIsolator chromosome includes both secure and normal code blocks. The code blocks are split by the TrustZone SMC. A test suite used for fitness contains input-output pairs designed to detect information leakage and other traditional bugs. A second fitness function is added in TZOptimizer for reordering which counts the number of switches between the secure and normal worlds.

Fitness Function. We present a prototype fitness function for TZOptimizer:

$$f = \begin{cases} -1000 & \text{if the program does not compile} \\ w_1 * P_{IO} - w_2 * S & \text{if the program successfully compiles} \end{cases}$$

P_{IO} indicates the number of the input-output pairs that pass the test suite, and S indicates the number of world switches. w_1 and w_2 indicate the weights for P_{IO} and S, respectively. We leave normalization and tuning as future work.

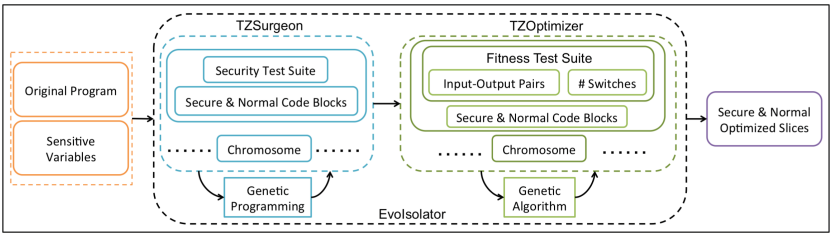
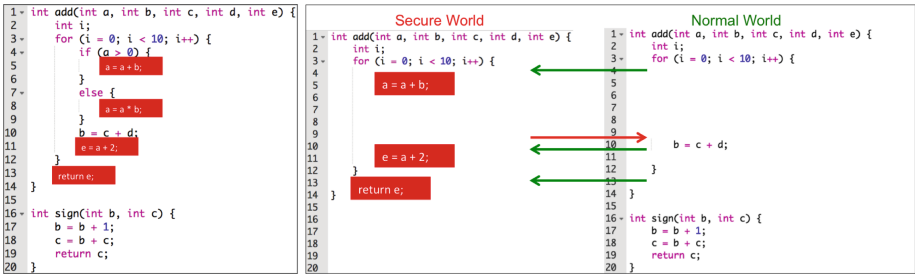


Fig. 1. System architecture and workflow of EvoIsolator

4 Feasibility Study

To evaluate the feasibility of EvoIsolator, we implemented a version of the second phase, TZOptimizer, using TZSlicer as input. We leave TZSurgeon as future work. We demonstrate our approach with an example (Fig. 2(a)).



(a) Original Example (b) Secure and Normal Slices

Fig. 2. Preparation for EvoIsolator (Color figure online)

4.1 Setup

This example contains the *add* and *sign* functions. The variable *a* in the *add* function is the secure (tainted variable). TZSlicer treats the lines 5, 8, 11, and 13 as sensitive computations (shown with the red text boxes). TZSlicer partitions the program as is seen in Fig. 2(b), placing the lines 5, 11, and 13 from the original program into the secure world and removing the redundant/non-executed code (e.g., line 8). The arrows indicate the world switching flow.

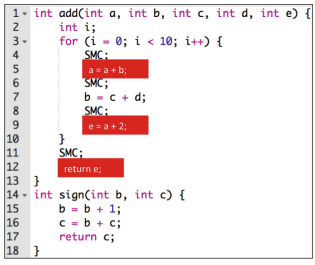


Fig. 3. Sliced example

Using the secure and normal slices generated by TZSlicer, EvoIsolator moves the slices to the secure and normal code blocks. To generate the fitness test suite, the code blocks are merged into a sliced program that is executable in a regular C environment (shown in Fig. 3). The SMC lines in the sliced program count the number of world switches in each loop iteration. In all, there are 31 switches in the initial sliced example.

TZOptimizer then tries to optimize solutions that pass all the test cases and achieves the minimum number of world switches. Figure 4 shows one of the crossover operations and one of the mutation operations for this phase. In Fig. 4(a), assume that TZOptimizer randomly picks two secure slices from the two chromosomes. By randomly selecting a crossover point, it swaps the code blocks in the chromosome parents and generates the offspring. In Fig. 4(b), assume that the mutation point is a line of the sensitive computation. TZOptimizer splits the target code block to two code blocks.

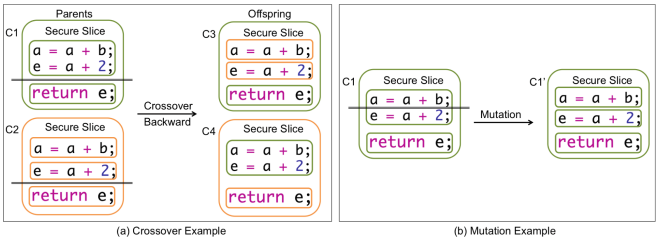


Fig. 4. Crossover and mutation examples

4.2 Evaluation

We built a version of TZOptimizer as a genetic algorithm in Python. We first generate 50 test cases (input-output pairs) based on the original program. Then, we input the secure and normal slices generated by TZSlicer to the TZOptimizer part of EvoIsolator. We use a population size of 12 based on some initial experiments. We set $w_1 = 10$ and $w_2 = 1/50$ for the weights in the fitness function. After executing multiple runs, EvoIsolator outputs two solutions, which reduce the original 31 switches (from TZSlicer) to 21 switches as is shown in Fig. 5. Solution 1 moves the line $e = a + 2$ backward and still keeps this line within the loop computation. In addition, EvoIsolator detects that it is unnecessary to place this line inside of the loop. Therefore, the second solution moves this line forward to the outside of the loop, which further reduces the resource usage during the computation and improves the efficiency of the program execution.

We ran the program 100 times to understand if it converges on a solution each time. We found that the number of generations to find this solution was usually less than 3, and in all cases we found a better solution. While this is a simple example we believe this can scale to larger programs.

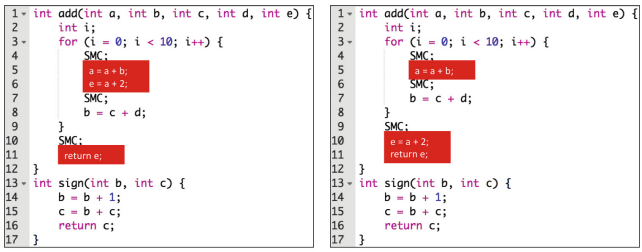


Fig. 5. Optimized example generated by EvoIsolator

5 Conclusions and Future Work

In this paper we proposed a search-based framework for hardware isolation, EvoIsolator. It optimizes slices for TrustZone applications to achieve the security of data and code with low communication overhead. We performed a feasibility study on phase II of EvoIsolator (TZOptimizer), which optimizes programs generated by Phase I (TZSurgeon). In future work we will implement the full-fledged EvoIsolator, tune the fitness function, and perform a comprehensive evaluation. We will also explore the use of multi-objective optimization.

Acknowledgments. This work was supported in part by National Science Foundation Grants CNS-1750867 and CCF-1745775.

References

1. ARM security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html
2. Ye, M., Sherman, J., Srisa-an, W., Wei, S.: TZSlicer: security-aware dynamic program slicing for hardware isolation. In: HOST (2018)
3. Mark Harman, S., Mansouri, A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012)
4. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Trans. Evol. Comput. **22**(3), 415–432 (2018)
5. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Trans. Softw. Eng. **44**, 574–594 (2017)
6. Hu, N., Ye, M., Wei, S.: Surviving information leakage hardware Trojan attacks using hardware isolation. IEEE TETC (2017)
7. Schuster, F., et al.: VC3: trustworthy data analytics in the cloud using SGX. In: S&P, pp. 38–54 (2015)
8. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, pp. 35–46 (1988)