TZSlicer: Security-Aware Dynamic Program Slicing for Hardware Isolation

Mengmei Ye, Jonathan Sherman, Witawas Srisa-an, and Sheng Wei

Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE, USA
Email: {mye, jsherman, witty, swei}@cse.unl.edu

***Abstract*—To address security issues related to information leakage, microprocessor designers and manufacturers such as ARM and Intel have introduced hardware isolation-based technologies to support secure software execution. However, utilizing such technologies often requires significant efforts to design new applications or refactor existing applications to adhere to the usage protocols. Developers also need to clearly distinguish code sections that can manipulate sensitive data and relocate them to the secure execution environment. These processes can be laborious and error-prone, since over-protection can result in poor application performance and high resource usage, and under-protection may cause exploitable security vulnerabilities.**

In this paper, we introduce TZSlicer, a framework to automatically identify code that must be protected based on a sensitive variable list provided by developers. TZSlicer automatically identifies code sections that can process sensitive data, extracts those sections from the original program, and creates harness in the original and extracted code sections so that they can interface with each other. We develop a prototype of TZSlicer to support slicing of C programs at function, code block, and code line levels. Also, we identify optimization opportunities to improve the context switching overhead of TZSlicer via applying loop unrolling and variable renaming. We evaluate TZSlicer using seven real-world programs, and the evaluation results indicate that TZSlicer is effective in protecting sensitive data without incurring significant runtime and resource usage overheads.

I. Introduction

Embedded systems, such as Internet of Things (IoT) devices, have gained a great deal of popularity recently. Currently, there are tens of billions of embedded systems in deployment, which perform various tasks ranging from controlling basic appliances to handling sensitive information via various forms of data processing and routing. Unfortunately, many of these deployed devices are inadequately protected from malicious acts such as leaking confidential information and modifying critical data [1], [2].

Since these embedded systems often have low power budget and limited computing capability, it is not applicable to deploy complex security mechanisms. Therefore, hardware vendors and manufacturers have developed hardware measures to provide the necessary security. One notable technology for secure data processing is hardware-based isolation. For example, ARM introduced TrustZone technology [3] for hardware isolation on embedded devices; Intel developed Software Guard Extensions (SGX) to provide isolated execution environment for Intel-based platforms [4].

While such technologies have been available, employing them in general embedded computing software is not straightforward. When the data that needs to be protected is only processed by a small region of code, designing a program to ensure that the data is processed in full isolation by trusted hardware is relatively simple. However, when sensitive information flows across multiple methods or in a much larger code region, securing the processing of such data can be quite complex. Furthermore, for legacy programs that have long been deployed, refactoring them to take advantage of technologies like TrustZone may require significant efforts. The resulting software can also be inefficient or erroneous. For example, manually identifying possible weak links along a processing chain that a piece of sensitive data must flow through can lead to *over-protection*, resulting in waste of precious hardware resources as excessive code is being protected. On the other hand, if the data flow is complex, manual analysis can lead to *under-protection*, resulting in weak links that can be exploited by adversaries.

We develop TZSlicer, a new framework to automatically identify code statements in a program that must be protected based on a sensitive, must-protect variable list provided by developers. As such, the inputs of TZSlicer are a program that contains sensitive data and a list of variables that must be protected. TZSlicer analyzes the program and applies taint analysis of sensitive variables. It then analyzes the taint analysis results and performs program slicing to extract statements that must be executed in isolation. The extracted statements are then used to form new methods that are executed in isolation. The remaining code from the original program is then instrumented so that code statements that make calls to the protected methods can be inserted. The outputs of TZSlicer include a "normal world" program that executes without hardware protection and a "secure world" program that executes under protection.

A major challenge in the design of TZSlicer is the fact that the capacity of the "secure world" must be kept low. It is because all the "secure world" resources are considered as part of the *trusted computing base* (*TCB*), which, once compromised, would propagate the security threats to the entire system. As having been a consensus in the security community, minimizing the size of TCB is an important requirement to security design [5]. We address this challenge

978-1-5386-4731-8/18/$31.00 ©2018 IEEE

by developing multiple program slicing schemes targeting the method (i.e., *TZ-M*), code block (i.e., *TZ-B*), and code line (i.e., *TZ-L*) levels. While *TZ-L* (i.e., the finest granularity in slicing) achieves the minimum TCB size, it introduces an increased number of context switches between the two worlds. To further address this issue, we develop an advanced version of *TZ-L* (i.e., *TZ-L+*) that conducts loop unrolling and rescheduling on the results of *TZ-L* to optimize the context switching overhead.

We evaluate the effectiveness and efficiency of TZSlicer using 7 real-world applications that perform common tasks in embedded systems, such as signal processing, cryptographic operations, and statistical computations. Our evaluation results indicate that TZSlicer can successfully achieve the program slicing objectives for hardware isolation and meet the security requirement. Furthermore, the multi-level slicing and the loop unrolling mechanisms are capable of achieving controllable TCB sizes and context switching times, respectively, which are essential to accommodate for various application requirements.

II. Background and Motivation

A. Threat Models

We target two common types of threat models that have been used against embedded systems: information leakage attacks and data falsifying attacks. Both attacks have the potential of compromising the security or privacy of the critical data and programs in the embedded systems and must be prevented. Throughout the discussions in this paper, we employ a concrete example as shown in Fig. 1 to introduce the threat models, the countermeasures, as well as the design and implementation of TZSlicer. Without loss of generality, we assume that array x contains sensitive data that are subject to security or privacy breaches under the two threat models. The program involves typical code structures (e.g., loops and branches), data structures (e.g., arrays), and operations (e.g., assignments and arithmetic computations) that commonly appear in real world C programs.

```
1   void M1(bool flag) {
2     if(flag) {
3       for (i = 0; i < 20; i++) {
4         b[i] = b[i] + 1;
5         y[i] = x[i] + a[i] + 1;
6         b[i] = a[i] + 2;
7         a[i] = a[i] + 1;
8         x[i] = x[i] + a[i];
9       }
10    }
11    else {
12      for (i = 0; i < 20; i++) {
13        b[i] = a[i] + b[i];
14        y[i] = x[i] + a[i];
15      }
16    }
```

```
1   int M2(int m, int n) {
2     int s = 0;
3     s = *p + *q;
4     return s;
5   }
```

```
1   void main() {
2     bool flag = true;
3     int m = 5;
4     int n = 6;
5     M1 (flag);
6     int s = M2(m, n);
7   }
```

Fig. 1. Sample C code for the technical discussion of TZSlicer, where array x contains sensitive data that requires protection.

1) Information Leakage Attacks.: Information leakage is one of the major forms of threats that compromise the security and privacy of critical systems [6], wherein the adversary breaches into the system, identifies the secret data, and leaks the data via an overt or covert channel. To date, there are a variety of effective methods to issue information leakage attacks, such as malware [7], memory access pattern inference [8], and hardware Trojans [9]. In the example shown in Fig. 1, an information leakage attack can be conducted by the attacker hacking into the program call stacks involving lines 5, 8, and 14 via debugging and reading the array x values.

2) Data Falsifying Attacks.: Different from the information leakage attacks that read critical data from the system, data falsifying attacks aim to modify the critical data, which may cause system malfunction or denial of services. In particular, in security sensitive systems, such as authentication or monitoring systems, the falsified data can cause the system to bypass the otherwise mandatory security verification procedures and result in severe security compromises [1]. In the example shown in Fig. 1, the attacker can issue a data falsifying attack directly by manipulating the array x values (i.e., at lines 5, 8, and 14) or indirectly by compromising the variable $flag$ (i.e., at line 2). Both cases falsify the critical data x and thus endanger the security of the entire system.

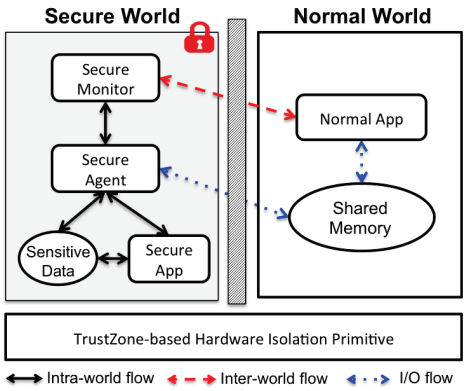

Fig. 2. TrustZone-based hardware isolation framework.

B. TrustZone Framework.

As a countermeasure to prevent the information leakage and data falsifying attacks, the security community has developed hardware isolation techniques that create trusted execution environment (TEE) for the protection of sensitive data and programs [3][4]. For example, Fig. 2 shows the hardware isolation framework we target in this paper based on ARM TrustZone [3]. The application runtime is split into two separate environments, namely the *secure* world and the *normal* world, which are isolated at the physical bus level using the TrustZone technology provided as a security extension in the ARM processor. The principle of hardware isolation is that the normal world cannot directly access the secure world as ensured by the bus level isolation. Instead, to issue an access to the secure world, the *normal app* must invoke a secure monitor call (SMC) that triggers the *secure monitor* to switch the CPU mode from normal to secure. Then, the *secure agent* in the secure world conducts security verifications and either grant or deny the access request. The data exchange between

the secure and normal worlds can be enabled by a *shared memory* allocated in the normal world, which is accessible to both worlds.

Fig. 3 shows the most straightforward way to deploy the target program to the TrustZone framework and gain immediate security benefits. The entire program is embedded in the secure world and, consequently, no accesses to any part of the program can be achieved without going through the security verifications at the *secure agent*.

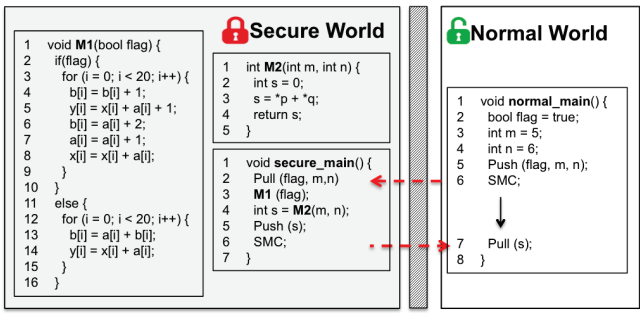

Fig. 3. Sliced programs by directly leveraging the ARM TrustZone framework.

Based on the example shown in Fig. 3, we note that the developer must create two separate programs: the *secure app* and the *normal app*, and deploy them into the two worlds in order to leverage the TrustZone framework. This significantly increases the complexity of software development. In particular, the new hardware isolation architecture requires the developer to determine how to split the data and code based on their security properties, which is challenging for the general software developers who do not have security background or deep understanding about the underlying hardware framework. Furthermore, we note that the straightforward code deployment shown in Fig. 3 results in a big size TCB (i.e., the size of the secure world) in the TrustZone framework, which negatively impacts security. In this paper, we aim to address these challenges by developing TZSlicer, an automatic security-aware program slicing technique to achieve security guarantee while maintaining a small TCB.

III. TZSlicer System Framework

Fig. 4 shows the system architecture of the TZSlicer framework, which automatically generates the secure and normal slices required by TrustZone for traditional software developers who do not have deep knowledge about TrustZone. To achieve this goal, we develop two system components for TZSlicer: the *Taint Analyzer* and the *Slice Optimizer*. The *Taint Analyzer* determines the propagation of sensitive information (i.e., the user tainted variables) throughout the program following the system dependency graph (SDG) [10]. Based on the taint analysis results, TZSlicer deploys all the sensitive components into the secure world and the rest into the normal world. After that, the *Slice Optimizer* further reduces two types of system runtime overhead introduced by the sliced programs, namely the resource overhead and the communication overhead, by refining the deployed code in the two slices.

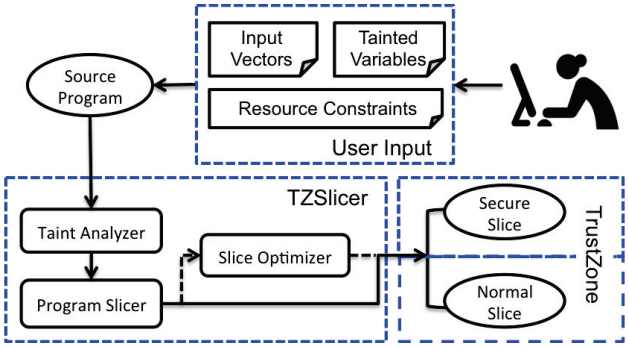

Fig. 4. TZSlicer system framework.

A. Taint Analyzer

To indicate the security requirement of the application for program slicing, the developer provides three data blocks as the inputs to TZSlicer: (1) a set of tainted variables that are the source of security sensitivities; (2) a set of test input vectors that provide a full coverage of the functional test to enable dynamic program slicing; and (3) the resource constraints that determine the key parameters in the slicing optimization.

Based on the user inputs, we employ TaintGrind, a dynamic taint analysis framework [11] to determine the propagation of taints in the SDG [10]. The originally tainted variables combined with those determined by the propagation analysis represent the critical program components that require protection. Depending on the granularity of the taint analysis, we develop the following three tainting methods.

- *Method-level Tainting* ***(TZ-M)***, which taints the program at the function level and generates security sensitive functions and non-sensitive functions;
- *Block-level Tainting* ***(TZ-B)***, which further taints the internal code blocks (e.g., branches) within the functions and generates security sensitive and non-sensitive blocks; and
- *Line-level Tainting* ***(TZ-L)***, in which each line of the program is labeled as either security sensitive or non-sensitive based on the taint propagation.

Fig. 5 shows the taint analysis procedure and results of *TZ-L* for the function $M1$ presented in Fig. 1, where Array x is the user tainted variable. We observe that the taint on x is propagated to 3 lines of code, including lines 5, 8, and 14, which are deemed security sensitive and subject to the information leakage and data falsifying threats discussed in Section II-A. The taint analysis fully identifies the attack surfaces that have the potential of exposing the sensitive data.

B. Program Slicer

The results of the taint analysis form two sets of program components: (1) the code that is tainted, either originally by the developer or by the propagation in the SDG; and (2) the code that is not tainted. By the definition of taint under this context, Component (1) indicates the security sensitive

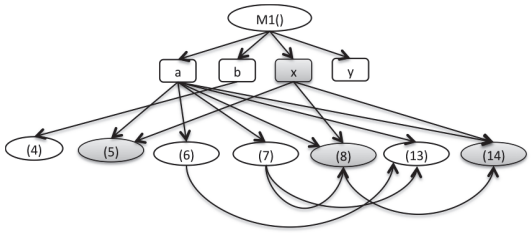

Fig. 5. System dependency graph for taint analysis for the Function $M1$ in Fig. 1. The shaded nodes represent the sensitive code (line numbers) identified by the taint analysis.

component that composes the *secure slice*, and Component (2) forms the *normal slice*, which are targeted to deploy in the secure world and the normal world, respectively.

We develop a *Program Slicer* that conducts the slicing and deployment of the two program slices based on the results obtained from the *Taint Analyzer*. An important design principle of the *Program Slicer* is to ensure that the sliced programs are functionally equivalent to the original program. Such a principle cannot be achieved by directly slicing the original program into the secure and normal slices, because there exist communications between the secure and normal slices that are now split into two separate and non-concurrent CPU modes. To address the missing communication problem, the *Program Slicer* injects world switching code into the two slices where the communication with the other slice is needed. As discussed in Section II, the communication between the two worlds is through the shared memory and, therefore, the injected world switching code involves two parts:

- *shared memory access*, in which the program slice loads or stores data to or from the shared memory, enabling the communication with the other slice; and
- *world switching*, in which the program slice in one world issues a secure monitor call that causes the secure monitor in the TrustZone framework to switch the CPU mode to the other world.

Upon accomplishing the memory access and the world switching operations, both the shared data and the CPU mode are switched from the source world to the destination world, which concludes one complete world switching. In summary, the program slicing methods achieve secure world resource savings with the consequence of possibly increased communication overhead in certain cases. After slicing, the information flow of the sliced programs remains the same as the original program.

C. Slice Optimizer

1) Resource Optimization: To address the system resource challenge, we employ *dynamic taint analysis* that is dependent upon the specific executions of the sliced programs, which has been shown *to achieve significantly smaller slices than static slicing* [12]. The dynamic taint analysis method leverages the user provided test input vectors and propagates the taints only if the corresponding code exists in the execution path determined by the test input vectors. Consequently, the secure world only hosts sensitive data and code that will be executed

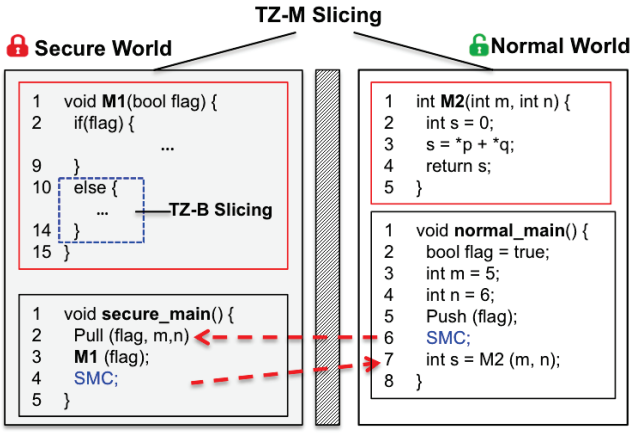

Fig. 6. *TZ-M* & *TZ-B* Slicing.

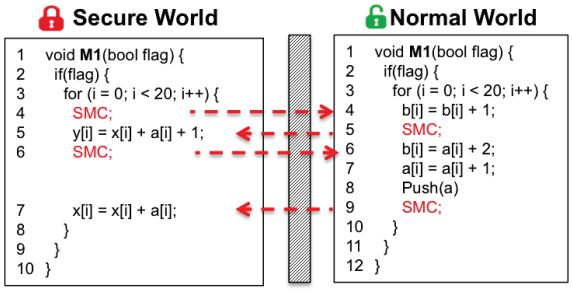

Fig. 7. *TZ-L* Slicing.

at system runtime, which reduces a significant amount of resource usage as compared to the static method. Fig. 6 and Fig. 7 demonstrate the complete program slicing process following the code example in Fig. 1, corresponding to the three tainting methods *TZ-M*, *TZ-B*, and *TZ-L*.

- *TZ-M Slicing*. As shown in Fig. 6, *TZ-M Slicing* applies the *TZ-M* tainting method, which places the tainted function $M1$ into the secure world and the non-tainted $M2$ function into the normal world. Compared to the baseline slicing method in Fig. 3, *TZ-M Slicing* reduces the resource overhead of the secure world by the size of the non-tainted $M2$ function. Consequently, since $M2$ is now in the normal world, a world switch together with shared memory accesses are required to maintain the original program flow and functionality.
- *TZ-B Slicing*. As shown in Fig. 6, *TZ-B Slicing* leverages *TZ-B* tainting that further carves the unexecuted "else" branch in *M1* out of the secure world, which can be achieved by considering the dynamic *flag* value provided by the user (*flag*=1). As can be observed from Fig. 6, *TZ-B Slicing* achieves direct resource savings in the secure world (by the size of the "else" branch in *M1*) without increasing the communication overhead.
- *TZ-L Slicing*. As shown in Fig. 7, *TZ-L Slicing* further reduces the secure world resource usage by tainting into the level of each line of code within the $M1$ function. Based on the *TZ-L* tainting method demonstrated in Fig. 5, the lines 5, 8, and 14 are tainted and migrated to the secure world. After

the slicing, in order to maintain the original program flow, the sliced programs must conduct 2 round-trip switches as shown in Fig. 7.

2) Communication Optimization: Isolation-aware Dynamic SMC Scheduling: To reduce the communication overhead, we employ a dynamic scheduling mechanism, which manipulates the order of the instruction execution at the runtime of each world to reduce the number of communications between the two worlds. To achieve this goal, the dynamic scheduling mechanism involves two systematic approaches, namely *loop unrolling* and *code reordering*. The *loop unrolling* scheme applies to the sliced program with loop structures, which fully or partially decomposes the loops and enables the merge of multiple iterations to reduce the number of world switches. The *code reordering* mechanism further reorders and merges the code within each iteration to reduce communications under the constraints of data dependencies.

Communication Optimization 1: Loop Unrolling. Similar to the benefits obtained in other domains [13], [14], loop unrolling in the context of TZSlicer has the potential of breaking the restricted boundaries enforced by the iterations, which enables flexible and thus less frequent world switches. Fig. 8 demonstrates the benefit of loop unrolling using the same example in Fig. 1 while unrolling 2 iterations of the loop at a time. We observe that although the number of switches per iteration remains 4, the total number of iterations is reduced by half and, therefore, the total number of switches is reduced by half after unrolling. Without loss of generality, we define an unrolling parameter x to represent the number of iterations being unrolled. In other words, the total number of switches is reduced by x times in the ideal case.

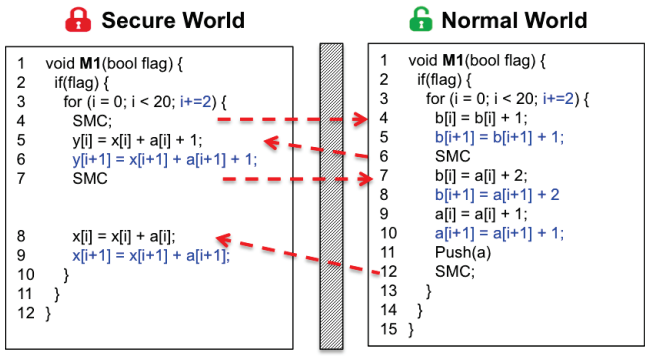

Fig. 8. *TZ-L+* Slicing using loop unrolling.

The applicability of loop unrolling is restricted by several factors. First, the loop must have been split partially into the secure and partially into the normal world with SMCs for communication. Second, the code in the next iteration must be predictable; in other words, there are no branch statements present in the loop. Third, there are no data dependencies in the loop body, as otherwise it is not possible to execute the code line of next iteration right after the one in the current iteration. For the programs that do not meet the first two requirements, we downgrade *TZ-L+* to *TZ-L*, while for those that do not meet the third requirement, we employ a variable renaming technique to eliminate the dependency.

Communication Optimization 2: Variable Renaming. In the cases of data dependency, we employ a variable renaming approach to eliminate the dependency and maintain the capability of unrolling and scheduling. The variable renaming approach works in the following two steps:

- *Step 1: Dependency Detection*, wherein we find variables that are subject to data dependencies, including read after write (RAW), write after read (WAR), and write after write (WAW) in the current iteration. Fig. 9 shows an example (extracted from the *DAXPY* test program discussed in Section IV), where line #4 has an RAW dependency on lines #2 and #3. Consequently, the original loop unrolling method will fail as the lines #2 and #3 cannot be re-scheduled to the current iteration due to the data dependency.
- *Step 2: Variable Renaming*. Once a data dependency is detected, we rename the variables subject to the dependency by expanding the variable name with iteration numbers, as shown in Fig. 9. The renaming keeps the variables from being overwritten and thus eliminates the dependency.

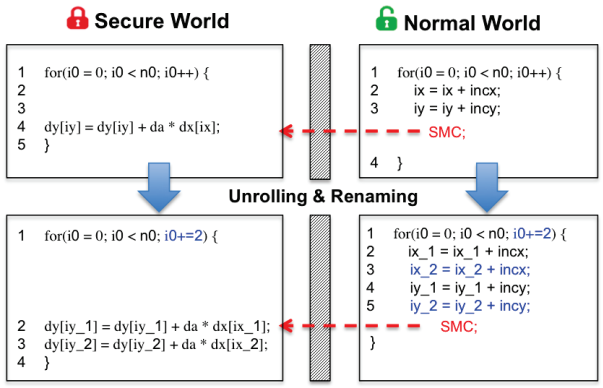

Fig. 9. *TZ-L+* variable renaming under data dependency.

IV. Empirical Evaluation

A. Experimental Setup

We adopt 7 real-world C programs to evaluate TZSlicer, as shown in TABLE I. The programs involve the functionalities of signal processing, cryptography, and statistical computations, where there exist security sensitive variables that are subject to information leakage and/or data falsifying attacks. TABLE I also summarizes the statistics of the programs. We observe that the programs cover a diverse set of test cases with varying lines of code (LoC), branch statements, loops, and functions, which collectively provide a comprehensive test set for evaluating different slicing methods supported by TZSlicer.

Furthermore, we develop a bare-metal TrustZone framework in our lab targeting Xilinx Zynq platforms. As shown in Fig. 10, the framework enables the TrustZone functionality on a Xilinx Zedboard by configuring the corresponding ARM CPU and the Xilinx specific registers following [15]. In our evaluation, we employ the 256KB on-chip memory (OCM) to

TABLE I
TEST PROGRAMS ADOPTED TO EVALUATE TZSLICER.

Test Cases	# Lines	Branches	Loops	Functions
FFT	83	3	7	1
Sobel_Filter	121	8	5	6
Matrix_Multiplication	26	1	3	1
AES_KeyExpansion	81	2	2	4
Linear_Regression	40	1	1	1
Shift_Cipher	57	8	0	2
DAXPY	33	5	2	1

deploy the secure world and normal world applications, each of which is assigned 128 KB of space. The hardware isolation is enforced at the physical bus level by the non-secure (NS) bit settings on the AXI interconnect, which ensures that normal world cannot access secure world resources as supported by the AMBA3 AXI bus protocol.

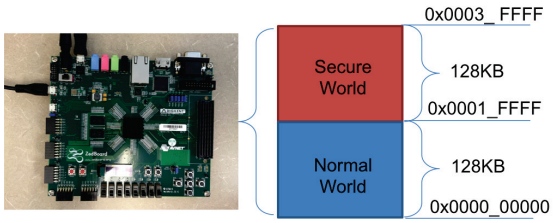

Fig. 10. TrustZone framework implemented on Xilinx Zynq Zedboard serving as the evaluation platform for TZSLICER.

B. Security Analysis

Technically, TZSLICER is capable of defending against the aforementioned information leakage and data falsifying attacks benefiting from the strong hardware bus-level isolation provided by ARM TrustZone. This is possible because attackers are now restricted from accessing the resources deployed in the secure world based on the user-specified tainted variables. Without loss of generality, we discuss the following three cases regarding taint propagation for security analysis in TZSLICER.

- *Assignment Statements*, which is the most straightforward way of taint propagation, where the taint propagates from the variable(s) being read (i.e., the right side of the assignment statement) to the those being written (i.e., the left side of the assignment statement). Based on our observation on the implemented prototype system, the *Taint Analyzer* is able to capture and taint all the variables being written, and the program slicer is thus able to place the code lines containing all those variables in the secure world (i.e., in the *TZ-L* case). Therefore, there are no security vulnerabilities that can be exploited due to the assignment statements.
- *Sub-function Calls*, which is also a common way of taint propagation. In this case, TZSLICER taints the arguments of the function that obtain assigned values from the user tainted variables, which ensures that the operations related to sensitive data is still protected (i.e., placed in the secure world) in the sub-functions.
- *Pointer Propagation*, where a tainted pointer instead of the value it points to is assigned to a new variable, via either assignment statements or sub-function calls. In this case, TZSLICER taints the new variable only if it is referencing the values of the corresponding pointer other than the pointer itself. In this way, the value of the sensitive variable will be protected, and even if the attacker has access to the pointer value, it is still impossible to access the value it points to.

In summary, we observe that TZSLICER is able to handle all the above channels where a sensitive variable has the potential of being leaked.

C. Quantitative Security Evaluation: TCB Size

We further evaluate the security of TZSLICER quantitatively by measuring the size of TCB, which is represented by the lines of code deployed in the secure world. TABLE II summarizes the results we obtained by running TZSLICER with the 7 test programs shown in TABLE I. We first taint a random input variable in each program. Then, we evaluate all the proposed slicing methods, including *TZ-M*, *TZ-B*, *TZ-L*, and *TZ-L+* (with various unrolling parameters) and compare them with the baseline approach in which the entire program is placed in the secure world. We adopt the relative saving value described in Equation (1) to quantify the improvement in each method (shown as the percentage numbers in parenthesis) and thus enable the comparison across examples.

$$Savings = \frac{Original\ Lines - Secure\ Lines}{Original\ Lines} \quad (1)$$

We observe from TABLE II that *TZ-M* achieves significant savings in the *AES_KeyExpansion* test case that involves non-tainted functions, while it does not contribute to the other cases where there are either too few functions or all of the functions are tainted. *TZ-B* reduces the TCB sizes for 6 out of the 7 programs. *TZ-L* achieves additional improvement by cutting down the TCB size at the level of code lines, which results in TCB savings in all 7 examples (ranging from 12.4% to 73.7%) compared to the baseline. *TZ-L+* increases the TCB sizes compared to *TZ-L* due to the code unrolling, which grows linearly with the unrolling parameter. Note that in programs where there are no loops or no world switching within the loops, including *Matrix_Multiplication*, *AES_KeyExpansion*, and *Shift_Cipher*, *TZ-L+* do not apply and thus result in the same TCB sizes as compared to *TZ-L*.

D. Performance Evaluation: World Switches

The migration of the partial program into the secure world results in additional communications (i.e., world switches) between the two worlds in order to maintain the original functionality, which introduces additional timing overhead compared to the original program without TrustZone protection. Therefore, we quantitatively evaluate the communication overhead of the programs generated by TZSLICER. TABLE III shows our evaluation results represented by the number of world switches (i.e., SMC calls) using the 7 test programs. Since *TZ-M* and *TZ-B* do not require world switches below the function level, we set

TABLE II
RESOURCE USAGE EVALUATION RESULTS REPRESENTED BY THE NUMBER OF LINES OF CODE. THE PERCENTAGE VALUE IN PARENTHESIS REPRESENT THE RELATIVE SAVINGS FROM THE BASELINE, AS DESCRIBED IN EQUATION (1).

Test Cases	Original(Baseline)	TZ-M	TZ-B	TZ-L	TZ-L+(x=2)	TZ-L+(x=3)	TZ-L+(x=4)
FFT	83	83 (0%)	80 (-3.6%)	60 (-27.7%)	81 (-2.4%)	90 (+8.4%)	99 (+19.3%)
Sobel_Filter	121	121 (0%)	121 (0%)	106 (-12.4%)	131 (+8.3%)	140 (+15.7%)	149 (+23.1%)
Matrix_Multiplication	26	26 (0%)	17 (-34.6%)	19 (-26.9%)	19 (-26.9%)	19 (-26.9%)	19 (-26.9%)
AES_KeyExpansion	81	49 (-39.5%)	40 (-50.6%)	42 (-48.1%)	42 (-48.1%)	42 (-48.1%)	42 (-48.1%)
Linear_Regression	40	40 (0%)	27 (-32.5%)	24 (-40%)	41 (+2.5%)	45 (+12.5%)	51 (+27.5%)
Shift_Cipher	57	57 (0%)	15 (-73.7%)	15 (-73.7%)	15 (-73.7%)	15 (-73.7%)	15 (-73.7%)
DAXPY	33	33 (0%)	17 (-48.5%)	16 (-51.5%)	26 (-21.2%)	29 (-12.1%)	32 (-3.0%)

the results of *TZ-L* as the baseline for comparison. We observe that in 4 of the test programs, *TZ-L+* is capable of reducing the communication overhead significantly (ranging from 7.8% to 63.2%) thanks to the benefit of unrolling. The other 3 examples, namely *Matrix_Multiplication*, *AES_KeyExpansion*, and *Shift_Cipher*, do not show improvements because they either do not have loops or do not contain world switches within the loops.

E. Discussion: Trade-off between Security and Performance

Our evaluation results in TABLES II and III indicate that there is a trade-off between the TCB size (i.e., security) and the number of world switches (i.e., performance). To better understand the trade-off, we plot the correlations between the normalized values of the two metrics in Fig. 11 while varying the unrolling parameter in *TZ-L+* from 2 to 4. The curves for the 4 test programs confirm the general trade-off between security and performance, which provide reference to the developers to fine tune TZSLICER based on the application-specific security and performance requirements.

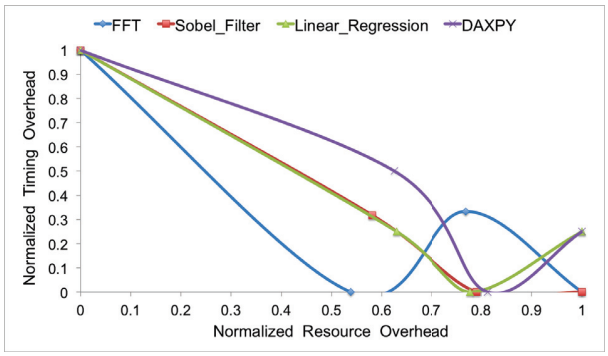

Fig. 11. Timing versus resource usage (normalized) under *TZ-L* and different unrolling parameters (x = 2, 3, 4) in *TZ-L+*.

V. RELATED WORK

A. Hardware Isolation

Hardware isolation techniques, such as ARM TrustZone [3] and Intel SGX [4], have become popular security primitives to prevent information leakage and data falsifying attacks. For example, ARM TrustZone [3] provides physical bus-level hardware isolation on ARM-based platforms, which has been adopted to protect a variety of software and hardware applications, such as programming runtime [16], operating system kernel [17], one-time password token [18], trusted platform module [19], and hardware Trojan defense [20]. Intel SGX [4] achieves hardware isolation (i.e., secure enclaves) by memory encryption, which has been employed to secure containers in the cloud [5]. Both hardware isolation primitives require an application-specific programming model that splits the runtime resources into isolated execution environments. However, most of the existing research has mainly focused on the applications of the isolation frameworks instead of the programming challenge. More recently, Lind et al. [21] proposed an automatic partitioning framework targeting Intel SGX. Our work differentiates from [21] by targeting a different hardware platform and hardware isolation mechanism (i.e., ARM TrustZone based on bus-level isolation), which introduces different context switching overhead and thus enables the opportunity of line-level slicing (i.e., *TZ-L*).

B. Program Slicing

Program Slicing [22] is a commonly used approach in software engineering, wherein a software program is partitioned into multiple pieces that satisfy certain slicing criteria to facilitate software testing, debugging, and maintenance. There are more than 30 types of program slicing methods that have been developed [22] to satisfy a variety of conditions, which fall under two basic categories: static slicing [10], [23] and dynamic slicing [24], [25]. Static slicing extracts slices from a program based on the static analysis without considering any specific input data. As such, they can suffer from imprecision that can lead to a large amount of code that must be protected in the context of our application. Dynamic slicing, on the other hand, considers the execution context and thus generates much smaller and more precise slices than static slicing. However, the generated slices are incomplete as the quality of the slices depends on the quality of the test inputs. Test inputs that provide higher code coverage would be likely to produce more slices.

TABLE III
EVALUATION OF CONTEXT SWITCHING OVERHEAD. THE PERCENTAGE VALUE IN PARENTHESIS REPRESENT THE RELATIVE SAVINGS FROM THE BASELINE.

Test Cases	TZ-M	TZ-B	TZ-L (Baseline)	TZ-L+(x=2)	TZ-L+(x=3)	TZ-L+(x=4)
FFT	0	0	77	68 (-11.7%)	71 (-7.8%)	68 (-11.7%)
Sobel_Filter	0	0	729	471 (-35.4%)	351 (-51.9%)	351 (-51.9%)
Matrix_Multiplication	0	0	0	0 (0%)	0 (0%)	0 (0%)
AES_KeyExpansion	0	0	101	101 (0%)	101 (0%)	101 (0%)
Linear_Regression	0	0	19	10 (-47.4%)	7 (-63.2%)	10 (-47.4%)
Shift_Cipher	0	0	0	0 (0%)	0 (0%)	0 (0%)
DAXPY	0	0	20	14 (-30.0%)	8 (-60.0%)	11 (-45.0%)

VI. CONCLUSION AND DISCUSSIONS

We have developed TZSLICER, a security-aware dynamic program slicing framework, which automatically partitions the target program into a secure slice and a normal slice to work with the hardware isolation-based trusted execution environment. While designing TZSLICER, we focused on optimizing the trusted computing base of the system to ensure security while reducing the performance overhead via loop unrolling and scheduling. Our experimental results on seven real-world C programs justified the effectiveness and performance of TZSLICER. Although mainly targeting the ARM TrustZone platform in this work, TZSLICER can be adapted to other isolation-based frameworks, such as Intel SGX [4], with minimum engineering efforts on the wrapper of the program and the world switching operations.

The goal of TZSLICER is to bridge the gap between hardware security and software developers/end users, so that they can benefit from adopting hardware security primitives (e.g., TrustZone) in their system design without the huge burden of development or deep understanding of hardware security. The TZSLICER project is available at [26], where we plan to release the current and future versions of TZSLICER together with new evaluation results.

For future work, we will focus on addressing the limitation of dynamic tainting that produces incomplete coverage of execution paths, because only paths exercised by the test inputs would undergo the taint analysis. This can potentially leave vulnerable paths that were not exercised during the initial taint analysis. Our anticipated solution is to develop a dynamic code generation technique to identify additional paths that must be protected at runtime and employ the dynamic secure/normal migration feature of TrustZone to adapt to the runtime changes.

ACKNOWLEDGEMENT

We appreciate the constructive reviews provided by the anonymous reviewers. Also, we would like to thank Nianhang Hu who contributed to the implementation of the TrustZone framework used in this work.

REFERENCES

[1] V. van der Veen et al., "Drammer: Deterministic rowhammer attacks on mobile platforms," in *CCS*, 2016, pp. 1675–1689.
[2] R. G. Dutta et al., "Estimation of safe sensor measurements of autonomous system under attack," in *DAC*, 2017.
[3] "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.
[4] "Intel SGX," https://software.intel.com/en-us/isa-extensions/intel-sgx.
[5] F. S. et al., "VC3: Trustworthy data analytics in the cloud using SGX," in *S&P*, 2015, pp. 38–54.
[6] "Yahoo says 1 billion user accounts were hacked," https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html.
[7] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *S&P*, 2012, pp. 95–109.
[8] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Steal this movie: Automatically bypassing DRM protection in streaming media services," in *USENIX Security*, 2013.
[9] M. Tehranipoor and F. Koushanfar, "A survey of hardware Trojan taxonomy and detection," in *IEEE Design & Test of Computers*, 2010, pp. 10–25.
[10] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *PLDI*, 1988, pp. 35–46.
[11] "Taintgrind," https://github.com/wmkhoo/taintgrind.
[12] J. Newsome, D. Song, J. Newsome, and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *NDSS*, 2005.
[13] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in HLS," in *DATE*, 2015, pp. 163–168.
[14] J. Cong and C. H. Yu, "Impact of loop transformations on software reliability," in *ICCAD*, 2015, pp. 278–285.
[15] Xilinx Inc., "Programming ARM TrustZone architecture on the Xilinx Zynq-7000 all programmable SoC," in *UG1019 (v1.0)*, 2014.
[16] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *ASPLOS*, 2014, pp. 67–80.
[17] A. Azab et al., "SKEE: A lightweight secure kernel-level execution environment for arm," in *NDSS*, 2016.
[18] H. Sun et al., "TrustOTP: Transforming smartphones into secure one-time password tokens," in *CCS*, 2015, pp. 976–988.
[19] H. Raj et al., "fTPM: A software-only implementation of a TPM chip," in *USENIX Security*, 2016, pp. 841–856.
[20] N. Hu, M. Ye, and S. Wei, "Surviving information leakage hardware Trojan attacks using hardware isolation," *IEEE TETC*, 2017.
[21] J. Lind et al., "Glamdring: Automatic application partitioning for intel SGX," in *USENIX ATC*, 2017, pp. 285–298.
[22] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 12:1–12:41, Jun. 2012.
[23] J.-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1097–1113, 1994.
[24] D. Binkley et al., "A formalisation of the relationship between forms of program slicing," *Sci. Comput. Program.*, vol. 62, no. 3, pp. 228–252, 2006.
[25] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, 1990.
[26] "TZSlicer github repository," https://github.com/hwsel/tzslicer.