ScaRR:
Scalable Runtime Remote Attestation for Complex Systems

Flavio Toffalini
SUTD

Eleonora Losiouk
University of Padua

Andrea Biondo
University of Padua

Jianying Zhou
SUTD

Mauro Conti
University of Padua

flavio_toffalini@mymail.sutd.edu.sg, jianying_zhou@sutd.edu.sg
elosiouk@math.unipd.it, andrea.biondo.1@studenti.unipd.it, conti@math.unipd.it

Abstract

The introduction of remote attestation (RA) schemes has allowed academia and industry to enhance the security of their systems. The commercial products currently available enable only the validation of static properties, such as applications fingerprint, and do not handle runtime properties, such as control-flow correctness. This limitation pushed researchers towards the identification of new approaches, called runtime RA. However, those mainly work on embedded devices, which share very few common features with complex systems, such as virtual machines in a cloud. A naive deployment of runtime RA schemes for embedded devices on complex systems faces scalability problems, such as the representation of complex control-flows or slow verification phase.

In this work, we present ScaRR: the first Scalable Runtime Remote attestation schema for complex systems. Thanks to its novel control-flow model, ScaRR enables the deployment of runtime RA on any application regardless of its complexity, by also achieving good performance. We implemented ScaRR and tested it on the benchmark suite SPEC CPU 2017. We show that ScaRR can validate on average 2M control-flow events per second, definitely outperforming existing solutions that support runtime RA on complex systems.

1 Introduction

RA is a procedure that allows an entity (*i.e.,* the *Verifier*) to verify the status of a device (*i.e.,* the *Prover*) from a remote location. This is achieved by having first the *Verifier* sending a challenge to the *Prover*, which replies with a report. Then, the *Verifier* analyzes the report to identify whether the *Prover* has been compromised [10]. In standard RA, usually defined as static, the *Prover* verification involves the integrity of specific hardware and software properties (*e.g.,* the *Prover* has loaded the correct software). On the market, there are already several available products implementing static RA, such as Software Guard Extensions (SGX) [18] or Trusted Platform Module (TPM) [42]. However, these do not provide a defence against runtime attacks (*e.g.,* the control-flow ones) that aim to modify the program runtime behaviour. Therefore, to identify *Prover* runtime modifications, researchers proposed runtime RA. Among the different solutions belonging to this category, there are also the control-flow attestation approaches, which encode the information about the executed control-flow of a process [8, 9].

In comparison to static RA, the runtime one is relatively new, and today there are no reliable products available on the market since researchers have mainly investigated runtime RA for embedded devices [8, 9, 21, 22, 50]: most of them encode the complete execution path of a *Prover* in a single hash [8, 22, 50]; some [9] compress it in a simpler representation and rely on a policy-based verification schema; other ones [21] adopt symbolic execution to verify the control-flow information continuously sent by the *Prover*. Even if they have different performances, none of the previous solutions can be applied to a complex system (*e.g.,* virtual machines in a cloud) due to the following reasons: (i) representing all the valid execution paths through hash values is unfeasible (*e.g.,* the number of execution paths tends to grow exponentially with the size of the program), (ii) the policy-based approaches might not cover all the possible attacks, (iii) symbolic execution slows down the verification phase.

The purpose of our work is to fill this gap by providing ScaRR, the first runtime RA schema for complex systems. In particular, we focus on environments such as Amazon Web Services [2] or Microsoft Azure [3]. Since we target such systems, we require support for features such as multi-threading. Thus, ScaRR provides the following achievements with respect to the current solutions supporting runtime RA: (i) it makes the runtime RA feasible for any software, (ii) it enables the *Verifier* to verify intermediate states of the *Prover* without interrupting its execution, (iii) it supports a more fine-grained analysis of the execution path where the attack has been performed. We achieve these goals thanks to a novel model for representing the execution paths of a program, which is based on the fragmentation of the whole path into meaningful sub-paths. As a consequence, the *Prover* can send

a series of intermediate partial reports, which are immediately validated by the *Verifier* thanks to the lightweight verification procedures performed.

ScaRR is designed to defend a *Prover*, equipped with a trusted anchor and with a set of the standard solutions (*e.g.,* W⊕X/DEP [34], Address Space Layout Randomization - ASLR [29], and Stack Canaries [14]), from attacks performed in the user-space and aimed at modifying the *Prover* runtime behaviour. The current implementation of ScaRR requires the program source code to be properly instrumented through a compiler based on LLVM [31]. However, it is possible to use lifting techniques [5], as well. Once deployed, ScaRR allows to verify on average $2M$ control-flow events per second, which is significantly more than the few hundred per second [21] or the thousands per second [9] verifiable through the existing solutions.

Contribution. The contributions of this work are the following ones:

- We designed a new model for representing the execution path for applications of any complexity.
- We designed and developed ScaRR, the first schema that supports runtime RA for complex systems.
- We evaluated the ScaRR performances in terms of: (i) attestation speed (*i.e.,* the time required by the *Prover* to generate a partial report), (ii) verification speed (*i.e.,* the time required by the *Verifier* to evaluate a partial report), (iii) overall generated network traffic (*i.e.,* the network traffic generated during the communication between *Prover* and *Verifier*).

Organization. The paper is organized as follow. First, we provide a background on standard RA and control-flow exploitation (Section 2), and define the threat model (Section 3). Then, we describe the ScaRR control-flow model (Section 4) and its design (Section 5). We discuss ScaRR implementation details (Section 6) and evaluate its performance and security guarantees (Section 7). Finally, we discuss ScaRR limitations (Section 8), related works (Section 9), and conclude with final remarks (Section 10).

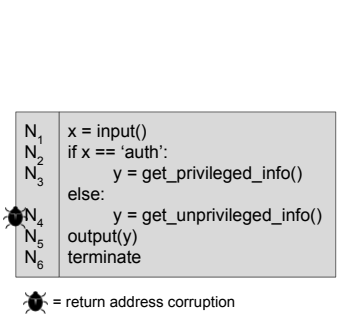
2 Background

The purpose of this section is to provide background knowledge about standard RA procedures and control-flow attacks.

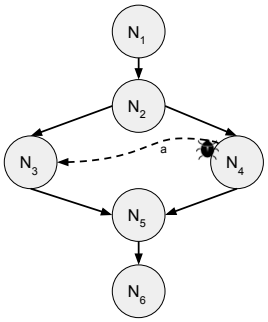
Remote Attestation. RA always involves a *Prover* and a *Verifier*, with the latter responsible for verifying the current status of the former. Usually, the *Verifier* sends a challenge to the *Prover* asking to measure specific properties. The *Prover*, then, calculates the required measurement (*e.g.,* a hash of the application loaded) and sends back a report R, which contains the measurement M along with a digital fingerprint F, for instance, $R = (M, F)$. Finally, the *Verifier* evaluates the report, considering its freshness (*i.e.,* the report has not been generated through a replay attack) and correctness (*i.e.,* the *Prover* measurement is valid). It is a standard assumption that the *Verifier* is trusted, while the *Prover* might be compromised. However, the *Prover* is able to generate a correct and fresh report due to its trusted anchor (*e.g.,* a dedicated hardware module).

Control-Flow Attacks. To introduce control-flow attacks, we first discuss the concepts of control-flow graph (CFG), execution-path, and basic-block (BBL) by using the simple program shown in Figure 1a as a reference example. The program starts with the acquisition of an input from the user (line 1). This is evaluated (line 2) in order to redirect the execution towards the retrieval of a privileged information (line 3) or an unprivileged one (line 4). Then, the retrieved information is stored in a variable (y), which is returned as an output (line 5), before the program properly concludes its execution (line 6).

A CFG represents all the paths that a program may traverse during its execution and it is statically computed. On the contrary, an execution path is a single path of the CFG traversed by the program at runtime. The CFG associated to the program in Figure 1a is depicted in Figure 1b and it encompasses two components: nodes and edges. The former are the BBLs of the program, while the latter represent the standard flow traversed by the program to move from a BBL towards the next one. A BBL is a linear sequence of instructions with a single entry point (*i.e.,* no incoming branches to the set of instructions other than the first), and a single exit point (*i.e.,* no outgoing branches from the set of instructions other than the last). Therefore, a BBL can be considered an atomic unit with respect to the control-flow, as it will either be fully executed, or not executed at all on a given execution path. A BBL might end with a control-flow event, which could be one of



(a) Pseudo-code of a program under a control-flow attack.



(b) Control-flow graph of a program under a control-flow attack.

Figure 1: Illustrative example of a control-flow attack.

the following in a X86_64 architecture: procedure calls (*e.g.,* `call`), jumps (*e.g.,* `jmp`), procedure returns (*e.g.,* `ret`), and system calls (*e.g.,* `syscall`). During its execution, a process traverses several BBLs, which completely define the process execution path.

Runtime attacks, and more specifically the control-flow ones, aim at modifying the CFG of a program by tampering with its execution path. Considering Figure 1, we assume that an attacker is able to run the program (from the node N_1), but that he is not authorized to retrieve the privileged information. However, the attacker can, anyway, violate those controls through a memory corruption error performed on the node N_4. As soon as the attacker provides an input to the program and starts its execution, he will be redirected to the node N_4. At this point, the attacker can exploit a memory corruption error (*e.g.,* a stack overflow) to introduce a new edge from N_4 to N_3 (edge labeled as a) and retrieve the privileged information. As a result, the program traverses an unexpected execution path not belonging to its original CFG. Even though several solutions have been proposed to mitigate such attacks (*e.g.,* ASLR [29]), attackers still manage to perform them [43].

This illustrative example about how to manipulate the execution path of a program is usually the basic step to perform more sophisticated attacks like exploiting a vulnerability to take control of a process [49] or installing a persistent data-only malware without injecting new code, once the control over a process is taken by the attacker [44].

Runtime RA provides a reliable mechanism which allows the *Verifier* to trace and validate the execution path undertaken by the *Prover*.

3 Threat Model and Requirements

In this section, we describe the features of the *Attacker* and the *Prover* involved in our threat model. Our assumptions are in line with other RA schemes [8, 9, 18, 21, 47].

Attacker. We assume to have an attacker that aims to control a remote service, such as a Web Server or a Database Management System (DBMS), and that has already bypassed the default protections, such as Control Flow Integrity (CFI). To achieve his aim, the attacker can adopt different techniques, among which: Return-Oriented Programming (ROP)/ Jump-Oriented Programming (JOP) attacks [16, 17], function hooks [35], injection of a malware into the victim process, installation of a data-only malware in user-space [44], or manipulation of other user-space processes, such as security monitors. In our threat model, we do not consider physical attacks (our complex systems are supposed to be virtual machines), pure data-oriented attacks (*e.g.,* attacks that do not alter the original program CFG), self-modifying code, and dynamic loading of code at runtime (*e.g.,* just-in-time compilers [41]). We refer to Section 7.4 for a comprehensive attacker analysis.

Prover. The *Prover* is assumed to be equipped with: (i) a trusted anchor that guarantees a static RA, (ii) standard defence mitigation techniques, such as W⊕X/DEP, ASLR. In our implementation, we use the kernel as a trusted anchor, which is a reasonable assumption if the machines have trusted modules such as a TPM [42]. However, we can also use a dedicated hardware, as discussed in Section 8. The *Prover* maintains sensitive information (*i.e.,* shared keys and cryptographic functions) in the trusted anchor and uses it to generate fresh reports, that cannot be tampered by the attacker.

4 ScaRR Control-Flow Model

ScaRR is the first schema that allows to apply runtime RA on complex systems. To achieve this goal, it relies on a new model for representing the CFG/execution path of a program. In this section, we illustrate first the main components of our control-flow model (Section 4.1) and, then, the challenges we faced during its design (Section 4.2).

4.1 Basic Concepts

The ScaRR control-flow model handles BBLs at assembly level and involves two components: *checkpoints* and *List of Actions (LoA)*.

A *checkpoint* is a special BBL used as a delimiter for identifying the start or the end of a sub-path within the CGF/execution path of a program. A *checkpoint* can be: *thread beginning/end*, if it identifies the beginning/end of a thread; *exit-point*, if it represents an exit-point from an application module (*e.g.,* a system call or a library function invocation); *virtual-checkpoint*, if it is used for managing special cases such as *loops* and *recursions*.

A *LoA* is the series of significant edges that a process traverses to move from a *checkpoint* to the next one. Each edge is represented through its source and destination BBL and, comprehensively, a *LoA* is defined through the following notation:

$$[(\mathrm{BBL}_{s1}, \mathrm{BBL}_{d1}), \ldots, (\mathrm{BBL}_{sn}, \mathrm{BBL}_{dn})].$$

Among all the edges involved in the complete representation of a CFG, we consider only a subset of them. In particular, we look only at those edges that identify a unique execution path: procedure call, procedure return and branch (*i.e.,* conditional and indirect jumps).

To better illustrate the ScaRR control-flow model, we now recall the example introduced in Section 2. Among the six nodes belonging to the CFG of the example, only the following four ones are *checkpoints*: N_1, since it is a *thread beginning*; N_3 and N_4, because they are *exit-points*, and N_6,

since it is a *thread end*. In addition, the *LoAs* associated to the example are the following ones:

$$N_1 - N_3 \Rightarrow [(N_2, N_3)]$$
$$N_1 - N_4 \Rightarrow [(N_2, N_4)]$$
$$N_3 - N_6 \Rightarrow []$$
$$N_4 - N_6 \Rightarrow [].$$

On the left we indicate a pair of *checkpoints* (*e.g.,* $N_1 - N_3$), while on the right the associated *LoA* (empty *LoAs* are considered valid).

4.2 Challenges

Loops, *recursions*, *signals*, and *exceptions* involved in the execution of a program introduce new challenges in the representation of a CFG since they can generate uncountable executions paths. For example, *loops* and *recursions* can generate an indefinite number of possible combinations of *LoA*, while *signals*, as well as *exceptions*, can introduce an unpredictable execution path at any time.

Loops. In Figure 2, we illustrate the approach used to handle *loops*. Since it is not always possible to count the number of iterations of a loop, we consider the conditional node of the *loop* (N_1) as a *virtual-checkpoint*. Thus, the *LoAs* associated to the example shown in Figure 2 are as follows:

$$S_A - N_1 \Rightarrow []$$
$$N_1 - N_1 \Rightarrow [(N_1, N_2)]$$
$$N_1 - S_B \Rightarrow [(N_1, N_3)].$$

Recursions. In Figure 3, we illustrate our approach to handle *recursions*, *i.e.,* a function that invokes itself. Intuitively, the *LoAs* connecting P_B and P_E should contain all the possible invocations made by `a()` towards itself, but the number of invocations is indefinite. Thus, we consider the node performing the recursion as a *virtual-checkpoint* and model only the path that could be chosen, without referring to the number of times it is really undertaken. The resulting *LoAs* for the

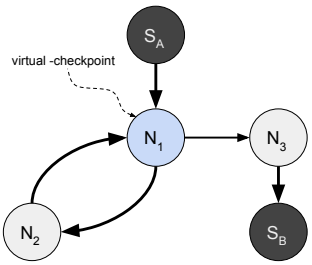


Figure 2: Loop example in the ScaRR control-flow model.

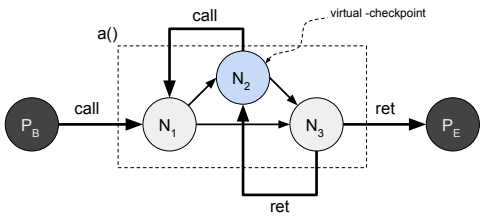


Figure 3: Recursion example in the ScaRR control-flow model.

example in Figuree 3 are the following ones:

$$P_B - N_2 \Rightarrow [(P_B, N_1), (N_1, N_2)]$$
$$N_2 - N_2 \Rightarrow [(N_2, N_1), (N_1, N_2)]$$
$$N_2 - N_2 \Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, N_2)]$$
$$N_2 - P_E \Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, P_E)]$$
$$P_B - P_E \Rightarrow [(P_B, N_1), (N_1, N_3), (N_3, P_E)].$$

Finally, the *virtual-checkpoint* can be used as a general approach to solve every situation in which an indirect jump targets a node already present in the *LoA*.

Signals. When a thread receives a *signal*, its execution is stopped and, after a context-switch, it is diverted to a dedicated handler (*e.g.,* a function). This scenario makes the control-flow unpredictable, since an interruption can occur at any point during the execution. To manage this case, ScaRR models the signal handler as a separate thread (adding *beginning/end thread checkpoints*) and computes the relative *LoAs*. If no handler is available for the *signal* that interrupted the program, the entire process ends immediately, producing a wrong *LoA*.

Exception Handler. Similar to *signals*, when a thread rises an *exception*, the execution path is stopped and control is transferred to a catch block. Since ScaRR has been implemented for Linux, we model the catch blocks as a separate thread (adding *beginning/end thread checkpoints*), but it is also possible to adapt ScaRR to fulfill different exception handling mechanisms (*e.g.,* in Windows). In case no catch block is suitable for the *exception* that was thrown, the process gets interrupted and the generated *LoA* is wrong.

5 System Design

To apply runtime RA on a complex system, there are two fundamental requirements: (i) handling the representation of a complex CFG or execution path, (ii) having a fast verification process. Previous works have tried to achieve the first requirement through different approaches. A first solution [8, 22, 50] is based on the association of all the valid execution paths of the *Prover* with a single hash value. Intuitively, this is not a scalable approach because it does not allow to handle complex

CFG/execution paths. On the contrary, a second approach [21] relies on the transmission of all the control-flow events to the *Verifier*, which then applies a symbolic execution to validate their correctness. While addressing the first requirement, this solution suffers from a slow verification phase, which leads toward a failure in satisfying the second requirement.

Thanks to its novel control-flow model, ScaRR enables runtime RA for complex systems, since its design specifically considers the above-mentioned requirements with the purpose of addressing both of them. In this section, we provide an overview of the ScaRR schema (Section 5.1) together with the details of its workflow (Section 5.2), explicitly motivating how we address both the requirements needed to apply runtime RA on complex systems.

5.1 Overview

Even if the ScaRR control-flow model is composed of *checkpoints* and *LoAs*, the ScaRR schema relies on a different type of elements, which are the *measurements*. Those are a combination of *checkpoints* and *LoAs* and contain the necessary information to perform runtime RA. Figure 4 shows an overview of ScaRR, which encompasses the following four components: a *Measurements Generator*, for identifying all the program valid *measurements*; a *Measurements DB*, for saving all the program valid *measurements*; a *Prover*, which is the machine running the monitored program; a *Verifier*, which is the machine performing the program runtime verification.

As a whole, the workflow of ScaRR involves two separate phases: an *Offline Program Analysis* and an *Online Program Verification*. During the first phase, the *Measurements Generator* calculates the CFG of the monitored *Application A* (Step 1 in Figure 4) and, after generating all the *Application A* valid *measurements*, it saves them in the *Measurements DB* (Step 2 in Figure 4). During the second phase, the *Verifier* sends a challenge to the *Prover* (Step 3 in Figure 4). Thus, the *Prover* starts executing the *Application A* and sending partial reports to the *Verifier* (Step 4 in Figure 4). The *Verifier* validates the freshness and correctness of the partial reports by comparing the received new *measurements* with the previous ones stored in the *Measurements DB*. Finally, as soon as the *Prover* finishes the processing of the input received from the *Verifier*, it sends back the associated output.

5.2 Details

As shown in Figure 4, the workflow of ScaRR goes through five different steps. Here, we provide details for each of those.

(1) Application CFG. The *Measurements Generator* executes the *Application A()*, or a subset of it (*e.g.,* a function), and extracts the associated CFG *G*.

(2) Offline Measurements. After generating the CFG, the *Measurements Generator* computes all the program *offline measurements* during the *Offline Program Analysis*. Each *offline measurement* is represented as a key-value pair as follows:

$$(\mathrm{cp}_A, \mathrm{cp}_B, H(\mathrm{LoA})) \Rightarrow [(\mathrm{BBL}_{s1}, \mathrm{BBL}_{d1}), \ldots, (\mathrm{BBL}_{sn}, \mathrm{BBL}_{dn})]$$

The key refers to a triplet, which contains two *checkpoints* (*i.e.,* cp_A and cp_B) and the hash of the *LoA* (*i.e., H(LoA)*) associated to the significant BBLs that are traversed when moving from the source *checkpoint* to the destination one. The value refers only to a subset of the BBLs pairs used to generate the hash of the *LoAs* and, in particular, only to procedure calls and procedure returns. Those are the control-flow events required to mount the shadow stack during the verification phase.

(3) Request for a Challenge. The *Verifier* starts a challenge with the *Prover* by sending it an input and a nonce, which prevents replay attacks.

(4) Online Measurements. While the *Application A* processes the input received from the *Verifier*, the *Prover* starts generating the *online measurements* which keep trace of the *Application A* executed paths. Each *online measurement* is represented through the same notation used for the keys in the *offline measurements*, *i.e.,* the triplet $(\mathrm{cp}_A, \mathrm{cp}_B, H(\mathrm{LoA}))$.

When the number of *online measurements* reaches a preconfigured limit, the *Prover* encloses all of them in a partial report and sends it to the *Verifier*. The partial report is defined as follows:

$$P_i = (R, F_K(R||N||i))$$
$$R = (T, M).$$

In the current notation, P_i is the i-th partial report, R the payload and $F_K(R||N||i)$ the digital fingerprint (*e.g.,* a message authentication code [15]). This is generated by using: (i) the secret key K, shared between *Prover* and *Verifier*, (ii) the nonce N, sent at the beginning of the protocol, and (iii) the index i, which is a counter of the number of partial reports. Finally, the payload R contains the *online measurements* M along with the associated thread T.

The novel communication paradigm between *Prover* and *Verifier*, based on the transmission and consequent verification of several partial reports, satisfies the first requirement for applying runtime RA on complex systems (*i.e.,* handling the representation of a complex CFG/execution path). This is achieved thanks to the ScaRR control-flow model, which allows to fragment the whole CFG/execution path into sub-paths. Consequently, the *Prover* can send intermediate reports even before the *Application A* finishes to process the received input. In addition, the fragmentation of the whole execution path into sub-paths allows to have a more fine-grained analysis of the program runtime behaviour since it is possible to identify the specific edge on which the attack has been performed.

(5) Report Verification. In runtime RA, the *Verifier* has two different purposes: verifying whether the running application is still the original one and whether the execution paths

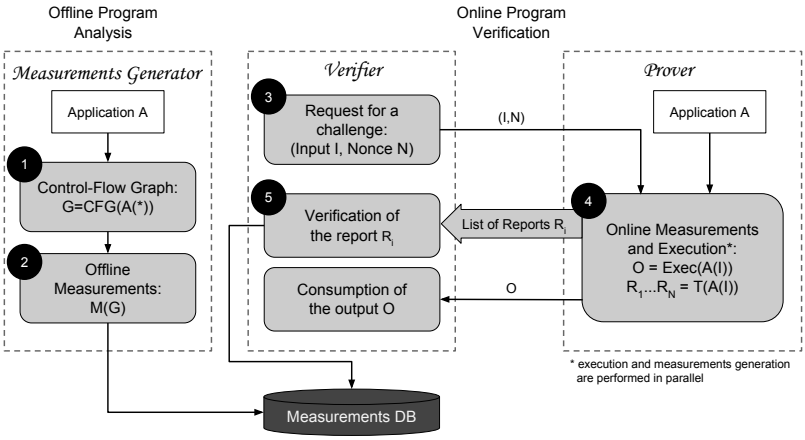


Figure 4: ScaRR system overview.

traversed by it are the expected ones. The first purpose, which we assume to be already implemented in the system [18, 47], can be achieved through a static RA applied on the *Prover* software stack. On the contrary, the second purpose is the main focus in our design of the ScaRR schema.

As soon as the *Verifier* receives a partial report P_i, it first performs a formal integrity check by considering its fingerprint $F_K(R||N||i)$. Then, it considers the *online measurements* sent within the report and performs the following checks: (C1) whether the *online measurements* are the expected ones (*i.e.,* it compares the received *online measurements* with the offline ones stored in the *Measurements DB*), (C2) whether the destination *checkpoint* of each *measurement* is equal to the source *checkpoint* of the following one, and (C3) whether the *LoAs* are coherent with the stack status by mounting a shadow stack. If one of the previous checks fails, the *Verifier* notifies an anomaly and it will reject the output generated by the *Prover*.

All the above-mentioned checks performed by the *Verifier* are lightweight procedures (*i.e.,* a lookup in a hash map data structure and a shadow stack update). The speed of the second verification mechanism depends on the number of procedure calls and procedure returns found for each *measurement*. Thus, also the second requirement for applying runtime RA on complex systems is satisfied (*i.e.,* keeping a fast verification phase). Once again, this is a consequence of the ScaRR control-flow model since the fragmentation of the execution paths allows both *Prover* and *Verifier* to work on a small amount of data. Moreover, since the *Verifier* immediately validates a report as soon as it receives a new one, it can also detect an attack even before the *Application A* has completed the processing of the input.

5.3 Shadow Stack

To improve the defences provided by ScaRR, we introduce a shadow stack mechanism on the *Verifier* side. To illustrate

```
S    int main(int argc, char ** argv) {
M1     a(10);
M2     /* irrelevant code */
M3     a(6);
M4     return 0;
E    }

     void a(int x) {
A1     /* irrelevant code */
C      printf("%d\n", x);
A2     return;
     }
```

Figure 5: Illustrative example to explain the shadow stack on the ScaRR *Verifier*.

it, we refer to the program shown in Figure 5, which contains only two functions: `main()` and `a()`. Each line of the program is a BBL and, in particular: the first BBL (*i.e., S*) and the last BBL (*i.e., E*) of the `main()` function are a *beginning thread* and *end thread checkpoints*, respectively; the function `a()` contains a function call to `printf()`, which is an *exit-point*. According to the ScaRR control-flow model, the *offline measurements* are the following ones:

$$(S,C,H_1) \Rightarrow [(M_1,A_1)],$$
$$(C,C,H_2) \Rightarrow [(A_2,M_2),(M_3,A_1)],$$
$$(C,E,H_3) \Rightarrow [(A_2,M_4)].$$

The significant BBLs we consider for generating the *LoAs* are: (i) the ones connecting the BBL S to the *checkpoint* C, (ii) the ones connecting two *checkpoints* C, and (iii) the ones to move from the *checkpoint* C to the last BBL E.

In this scenario, an attacker may hijack the return address of the function `a()` in order to jump to the BBL M_3. If this happens, the *Prover* produces the following *online measure-*

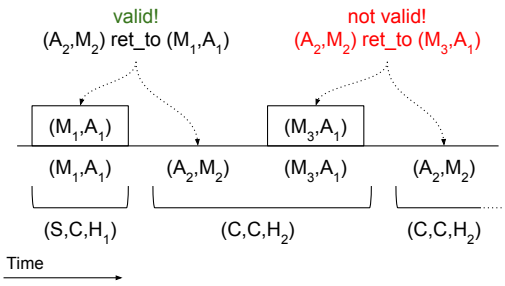


Figure 6: Implementation of the shadow stack on the ScaRR *Verifier*.

ments:

$$(S,C,H_1) \rightarrow (C,C,H_2) \rightarrow (C,C,H_2) \rightarrow \ldots.$$

Although generated after an attack, those measurements are still compliant with the checks $(C1)$ and $(C2)$ of the *Verifier*. Thus, to detect this attack, we introduce a new relation (*i.e.,* `ret_to`) to illustrate the link between two edges. The *Measurements Generator* computes all the `ret_to` relations during the *Offline Program Analysis* and saves them in the *Measurements DB* using the following notation:

$$(A_2,M_2) \texttt{ ret\_to } (M_1,A_1),$$
$$(A_2,M_4) \texttt{ ret\_to } (M_3,A_1).$$

Figure 6 shows how the *Verifier* combines all these information to build a remote shadow stack. At the beginning, the shadow stack is empty (*i.e.,* no function has been invoked yet). Then, according to the *online measurement* (S,C,H_1), the *Prover* has invoked the `main()` function passing through the edge (M_1,A_1), which is pushed on the top of the stack by the *Verifier*. Then, the *online measurement* (C,C,H_2) indicates that the execution path exited from the function *a()* through the edge (A_2,M_2), which is in relation with the edge on the top of the stack and therefore is valid. Moving forward, the *Verifier* pops from the stack and pushes the edge (M_3,A_1), which corresponds to the second invocation of the function `a()`. At this point, the third measurement (C,C,H_2) indicates that the *Prover* exited from the function `a()` through the edge (A_2,M_2), which is not in relation with (M_3,A_1). Thus, the *Verifier* detects the attack and triggers an alarm.

6 Implementation

Here, we provide the technical details of the ScaRR schema and, in particular, of the *Measurements Generator* (Section 6.1) and of the *Prover* (Section 6.2).

6.1 Measurements Generator

The *Measurements Generator* is implemented as a compiler, based on LLVM [31] and on the CRAB framework [24]. Despite this approach, it is also possible to use frameworks to lift the binary code to LLVM intermediate-representation (IR) [5].

The *Measurements Generator* requires the program source code to perform the following operations: (i) generating the *offline measurements*, and (ii) detecting and instrumenting the control-flow events. During the compilation, the *Measurements Generator* analyzes the LLVM IR to identify the control-flow events and generate the *offline measurements*, while it uses the CRAB LLVM framework to generate the CFG, since it provides a heap abstract domain that resolves indirect forward jumps. Again during the compilation, the *Measurements Generator* instruments each control-flow event to invoke a tracing function which is contained in the trusted anchor. To map LLVM IR BBLs to assembly BBLs, we remove the optimization flags and we include dummy code, which is removed after the compilation through a binary-rewriting tool. To provide the above-mentioned functionalities, we add around 3.5K lines of code on top of CRAB and LLVM 5.0.

6.2 Prover

The *Prover* is responsible for running the monitored application, generating the application *online measurements* and sending the partial reports to the *Verifier*. To achieve the second aim, the *Prover* relies on the architecture depicted in Figure 7, which encompasses several components belonging either to the user-space (*i.e., Application Process* and *ScaRR Libraries*) or to the kernel-space (*i.e., ScaRR sys_addaction*, *ScaRR Module*, and *ScaRR sys_measure*).

Each component works as follows:

- *Application Process* - the process running the monitored application, which is equipped with the required instrumentation for detecting control-flow events at runtime.
- *ScaRR Libraries* - the libraries added to the original application to trace control-flow events and *checkpoints*.
- *ScaRR sys_addaction* - a custom kernel syscall used to trace control-flow events.
- *ScaRR Module* - a module that keeps trace of the *online measurements* and of the partial reports. It also extracts the BBL labels from their runtime addresses, since the ASLR protection changes the BBLs location at each run.
- *ScaRR sys_measure* - a custom kernel syscall used to generate the *online measurements*.

When the *Prover* receives a challenge, it starts the execution of the application and creates a new *online measurement*. During the execution, the application can encounter *checkpoints* or control-flow events, both hooked by the instrumentation. Every time the application crosses a control-flow event, the *ScaRR Libraries* invoke the *ScaRR sys_addaction* syscall to

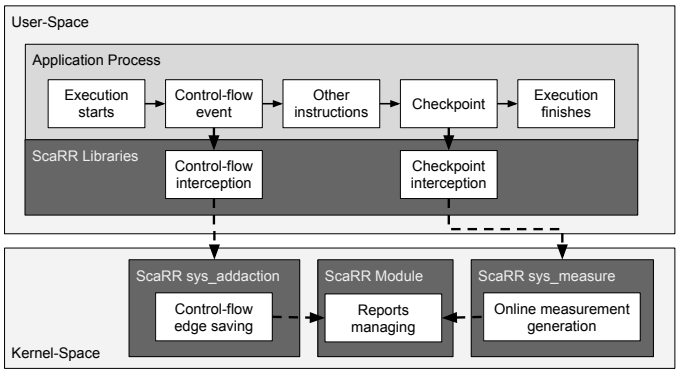


Figure 7: Internal architecture of the *Prover*.

save the new edge in a buffer inside the kernel-space. While, every time the application crosses a *checkpoint*, the *ScaRR Libraries* invoke the *ScaRR sys_measure* syscall to save the *checkpoint* in the current *online measurement*, calculate the hash of the edges saved so far, and, finally, store the *online measurement* in a buffer located in the kernel-space. When the predefined number of *online measurements* is reached, the *Prover* sends a partial report to the *Verifier* and starts collecting new *online measurements*. The *Prover* sends the partial report by using a dedicated kernel thread. The whole procedure is repeated until the application finishes processing the input of the *Verifier*.

The whole architecture of the *Prover* relies on the kernel as a trusted anchor, since we find it more efficient in comparison to other commercial trusted platforms, such as SGX and TrustZone, but other approaches can be also considered (Section 8). To develop the kernel side of the architecture, we add around 200 lines of code to a Kernel version v4.17-rc3. We also include the Blake2 source [4, 11], which is faster and provides high cryptographic security guarantees for calculating the hash of the *LoAs*.

7 Evaluation

We evaluate ScaRR from two perspectives. First, we measure its performance focusing on: attestation speed (Section 7.1), verification speed (Section 7.2) and network impact (Section 7.3). Then, we discuss ScaRR security guarantees (Section 7.4).

We obtained the results described in this section by running the bench-marking suite SPEC CPU 2017 over a Linux machine equipped with an Intel i7 processor and 16GB of memory [1]. We instrumented each tool to detect all the necessary control-flow events, we then extracted the *offline measurements* and we ran each experiment to analyze a specific performance metrics.

[1] We did not manage to map assembly BBL addresses to LLVM IR for 519.lbm_r and 520.omnetpp_r.

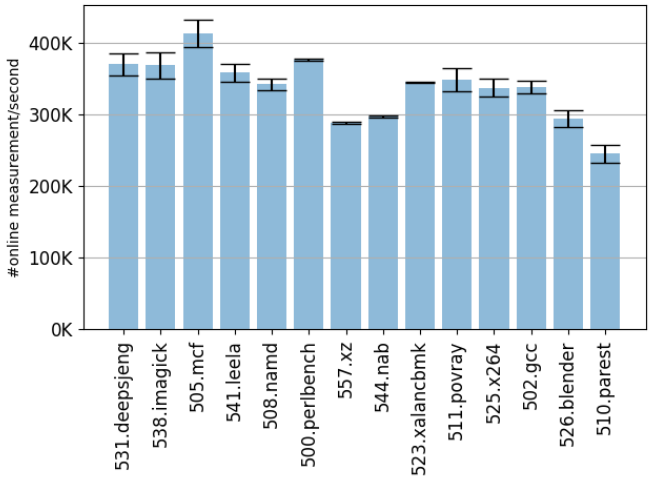


Figure 8: Average attestation speed measured as number of online measurements per second.

7.1 Attestation Speed

We measure the attestation speed as the number of *online measurements* per second generated by the *Prover*. Figure 8 shows the average attestation speed and the standard deviation for each experiment of the SPEC CPU 2017. More specifically, we run each experiment 10 times, calculate the number of *online measurements* generated per second in each run, and we compute the final average and standard deviation. Our results show that ScaRR has a range of attestation speed which goes from 250K (510.parest) to over 400K (505.mcf) of *online measurements* per second. This variability in performance depends on the complexity of the single experiment and on other issues, such as the file loading. Previous works prove to have an attestation speed around $20K$/ $30K$ of control-flow events per second [8, 9]. Since each *online measurement* contains at least a control-flow event, we can claim that ScaRR has an attestation speed at least 10 times faster than the one offered by the existing solutions.

7.2 Verification Speed

During the validation of the partial reports, the *Verifier* performs a lookup against the *Measurements DB* and an update of the shadow stack. To evaluate the overall performance of the *Verifier*, we consider the verification speed as the maximum number of *online measurements* verified per second. To measure this metrics, we perform the following experiment for each SPEC tool: first, we use the *Prover* to generate and save the *online measurements* of a SPEC tool; then, the *Verifier* verifies all of them without involving any element that might introduce delay (*e.g.,* network). In addition, we also introduce a digital fingerprint based on AES [39] to simulate an ideal scenario in which the *Prover* is fast. We perform

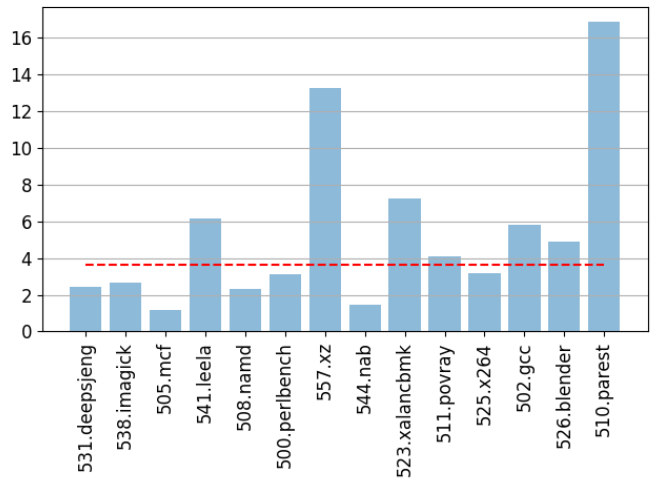


Figure 9: Average number of procedure calls and procedure returns found during the *Online Program Analysis* of the SPEC CPU 2017 tools.

the verification by loading the *offline measurements* in an in-memory hash map and performing the shadow stack. Finally, we compute the average verification speed of all tools.

According to our experiments, the average verification speed is 2M of *online measurements* per second, with a range that goes from 1.4M to 2.7M of *online measurements* per second. This result outperforms previous works in which the authors reported a verification speed that goes from 110 [21] to 30K [9] of control-flow events per second. As for the attestation speed, we recall that each *online measurement* contains at least one control-flow event.

The performance of the shadow stack depends on the number of procedure calls and procedure returns found during the generation of *online measurements* in the *Online Program Analysis* phase. To estimate the impact on the shadow stack, we run each experiment of the SPEC CPU 2017 tool and count the number of procedure calls and procedure returns. Figure 9 shows the average number of the above-mentioned variables found for each experiment. For some experiments (*i.e.,* 505.mcf and 544.nab), the average number is almost one since they include some recursive algorithms that correspond to small *LoAs*. If the average length of the *LoA*s tends to one, ScaRR behaves similarly to other remote RA solutions that are based on cumulative hashes [8,9]. Overall, Figure 9 shows that a median of push/pop operations is less than 4, which implies a fast update. Combining an in-memory hash map and a shadow stack allows ScaRR to perform a fast verification phase.

7.3 Network Impact and Mitigation

A high sending rate of partial reports from the *Prover* might generate a network congestion and therefore affect the verification phase. To reduce network congestion and improve verification speed, we perform an empirical measurement of the amount of data (*i.e.,* MB) sent on a local network with respect to the verification speed by applying different settings. The experiment setup is similar to Section 7.2, but the *Prover* and the *Verifier* are connected through an Ethernet network with a bandwidth of 10Mbit/s. At first, we record 1M of *online measurements* for each SPEC CPU 2017 tool. Then, we send the partial reports to the *Verifier* over a TCP connection, each time adopting a different approach among the following ones: *Single*, *Batch*, *Zip* [7], *Lzma* [32], *Bz2* [1] and *ZStandard* [6]. The results of this experiment are shown in Figure 10. In the first two modes (*i.e., Single* and *Batch*), we send a single *online measurement* and 50K *online measurements* in each partial report, respectively. As shown in the graph, both approaches generate a high amount of network traffic (around 80MB), introducing a network delay which slows down the verification speed. For the other four approaches, each partial report still contains 50K *online measurements*, but it is generated through different compression algorithms. All the four algorithms provide a high compression rate (on average over 95%) with a consequent reduction in the network overload. However, the algorithms have also different compression and decompression delays, which affect the verification speed. The *Zip* and *ZStandard* show the best performances with 1.2M of *online measurements*/s and 1.6M of *online measurements*/s, respectively, while *Bz2* (30K of online measurements/s) and *Lzma* (0.4M of online measurements/s) are the worst ones. The number of *online measurements* per partial report might introduce a delay in detecting attacks and its value depends on the monitored application. We opted for 50K because the SPEC CPU tools generate a high number of *online measurements* overall. However, this parameter strictly depends on the monitored application. This experiment shows that we can use compression algorithms to mitigate the network congestion and keep a high verification speed.

7.4 Attack Detection

Here, we describe the security guarantees introduced by ScaRR.

Code Injection. In this scenario, an attacker loads malicious code, *e.g., Shellcode*, into memory and executes it by exploiting a memory corruption error [38]. A typical approach is to inject code into a buffer which is under the attacker control. The adversary can, then, exploit vulnerabilities (*e.g.,* buffer overflows) to hijack the program control-flow towards the shellcode (*e.g.,* by corrupting a function return address).

When a W⊕X protection is in place, this attempt will generate a memory protection error, since the injected code is placed in a writable memory area and it is not executable. In case there is no W⊕X enabled, the attack will generate a wrong *LoA* detected by the *Verifier*.

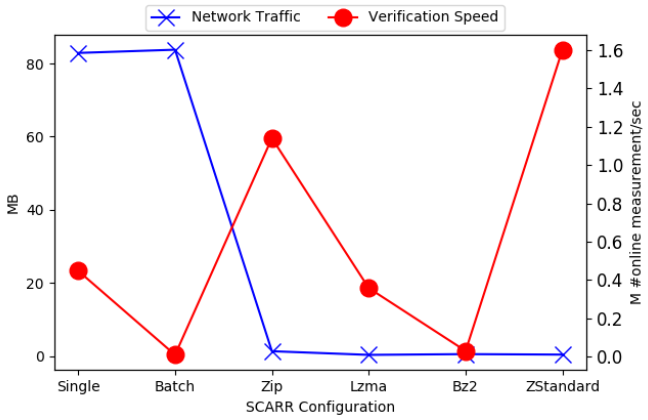


Figure 10: Comparison of different approaches for generating partial repors in terms of network traffic and verification speed.

Another strategy might be to overwrite a node (*i.e.,* a BBL) already present in memory. Even though this attempt is mitigated by W⊕X, as executable memory regions are not writable, it is still possible to perform the attack by changing the memory protection attributes through the operating system interface (*e.g.,* the `mprotect` system call in Linux), which makes the memory area writable. The final result would be an override of the application code. Thus, the static RA of ScaRR can spot the attack.

Return-oriented Programming. Compared to previous attacks, the code-reuse ones are more challenging since they do not inject new nodes, but they simply reorder legitimate BBLs. Among those, the most popular attack [37] is ROP [17], which exploits small sequences of code (gadgets) that end with a `ret` instruction. Those gadgets already exist in the programs or libraries code, therefore, no code is injected. The ROP attacks are Turing-complete in nontrivial programs [17], and common defence mechanisms are still not strong enough to definitely stop this threat.

To perform a ROP attack, an adversary has to link together a set of gadgets through the so-called ROP chain, which is a list of gadget addresses. A ROP chain is typically injected through a stack overflow vulnerability, by writing the chain so that the first gadget address overlaps a function return address. Once the function returns, the ROP chain will be triggered and will execute the gadget in sequence. Through more advanced techniques such as stack pivoting [19], ROP can also be applied to other classes of vulnerabilities, *e.g.,* heap corruption. Intuitively, a ROP attack produces a lot of new edges to concatenate all the gadgets, which means invalid *online measurements* that will be detected by ScaRR at the first *checkpoint*.

Jump-oriented Programming. An alternative to ROP attacks are the JOP ones [16, 48], which exploit special gadgets based on indirect `jump` and `call` instructions. ScaRR can detect those attacks since they deviate from the original control-flow.

Function Reuse Attacks. Those attacks rely on a sequence of subroutines, that are called in an unexpected order, *e.g.,* through virtual functions calls in C++ objects. ScaRR can detect these attacks, since the ScaRR control-flow model considers both the calling and the target addresses for each procedure call. Thus, an unexpected invocation will result in a wrong *LoA*. For instance, in Counterfeit Object-Oriented Programming (COOP) attacks [36], an attacker uses a loop to invoke a set of functions by overwriting a *vtable* and invoking functions from different calling addresses generates unexpected *LoAs*.

8 Discussion

In this section we discuss limitations and possible solutions for ScaRR.

Control-flow graph. Extracting a complete and correct CFG through static analysis is challenging. While using CRAB as abstract domain framework, we experienced some problems to infer the correct forward destinations in case of virtual functions. Thus, we will investigate new techniques to mitigate this limitation.

Reducing context-switch overhead. ScaRR relies on a continuous context-switch between user-space and kernel-space. As a first attempt, we evaluated SGX as a trusted platform, but we found out that the overhead was even higher due to SGX clearing the Translation-Lookaside Buffer (TLB) [40] at each enclave exit. This caused frequent page walks after each enclave call. A similar problem was related to the Page-Table Isolation (PTI) [46] mechanism in the Linux kernel, which protects against the Meltdown vulnerability. With PTI enabled, TLB is partially flushed at every context switch, significantly increasing the overhead of syscalls. New trusted platforms have been designed to overcome this problem, but, since they mainly address embedded software, they are not suitable for our purpose. We also investigated technologies such as Intel PT [25] to trace control-flow events at hardware level, but this would have bound ScaRR to a specific proprietary technology and we also found that previous works [25, 27] experienced information loss.

Physical attacks. Physical attacks are aimed at diverting normal control-flow such that the program is compromised, but the computed measurements are still valid. Trusted computing and RA usually provide protection against physical attacks. In our work, we mainly focus on runtime exploitation, considering that ScaRR is designed for a deployment on virtual machines. Therefore, we assume to have an adversary performing an attack from a remote location or from the user-space and the hosts not being able to be physically compromised. As a future work, we will investigate new solutions to prevent physical attacks.

Data-flow attestation. ScaRR is designed to perform runtime RA over a program CFG. Pure data-oriented attacks might force the program to execute valid, but undesired paths without injecting new edges. To improve our solution, we will investigate possible strategies to mitigate this type of attacks, considering the availability of recent tools able to automatically run this kind of exploit [28].

Toward a full semantic RA. We will investigate new approaches to validate series of *online measurements* by using runtime abstract interpretation [25, 27, 33].

9 Related Work

Runtime RA shares properties with classic CFI techniques. Thus, we discuss current state-of-the-art of both research areas.

Remote Attestation. Existing RA schemes are based on a cryptographic signature of a piece of software (*e.g.,* software modules, BIOS, operating system). Commercial solutions that implement such mechanisms are already available: TPM [42], SGX [18], and AMD TrustZone [47]. Academic approaches, which focus on cloud systems, are proposed by Liangmin et al. [45] and Haihe et al. [12]. More specifically, their solutions involve a static attestation schema for infrastructures as a service and JVM cloud computing, respectively. Even though these technologies can provide high-security guarantees, they focus on static properties (*i.e.,* signatures of components) and cannot offer any defence against runtime attacks.

To overcome design limitations of static RA, researchers propose runtime RA. Kil et al. [30] analyze base pointers of software components, such as stack and heap, and compare them with the measurements acquired offline. Bailey et al. [13] propose a coarse-grained level that attests the order in which applications modules are executed. Davi et al. [20] propose a runtime attestation based on policies, such as the number of instructions executed between two consecutive returns. Previous works suggest first to acquire a runtime measurement of software properties, but do not provide a fine-grained control-flow analysis.

A modern fine-grained control-flow RA is represented by C-FLAT, which is proposed by Abera et al. [8]. This schema measures the valid execution paths undertaken by embedded systems and generates a hash, which length depends on the number of control-flow events encountered at runtime. Then, the hash is compared with a list of offline measurements. The main differences between ScaRR and C-FLAT are the following ones: (i) C-FLAT control-flow representation grows along with software complexity, while ScaRR manages complex control-flow paths by using partial reports, and (ii) ScaRR is designed to use features of modern computer architectures (*e.g.,* multi-threading, bigger buffers). Dessouky et al. propose LO-FAT [22], which is a C-FLAT hardware implementation aimed at improving runtime performances for embedded systems. However, LO-FAT inherits all of C-FLAT design limitations in terms of control-flow representation. Zeitouni et al. designed ATRIUM [50], that strengthens runtime RA schemes against physical attacks for embedded devices. Even though the authors address different use cases, this solution might be combined with ScaRR.

Dessouky et al. propose LiteHax [21], that deals with data-only attacks. Their approach shares some similarities with ScaRR: they send detailed control-flow events information to a *Verifier*. However, they target data-oriented attacks (instead of control-flow). Moreover, LiteHax uses symbolic execution to validate the reports, which slows down the verification phase. Abera et al. discuss DIAT [9], which is a scalable RA for collaborative autonomous system. They model a runtime control-flow as a multi-set. This allows DIAT to represent complex control-flow graphs by using a relatively short hash. However, its model loses information about the execution order of the branches. This makes their approach prone to attacks like COOP [36]. ScaRR, instead, combines a strong static analysis and a shadow execution at the *Verifier* side that provides a sound approach by design. Overall, our experiments show that ScaRR can handle a higher number of branches per second compared to all the state-of-the-art runtime RA schemes.

Haldar et al. [26] propose a semantic RA, which leverages a virtual machine to validate semantic properties (*e.g.,* subclass inherited). However, the authors focus on run-time languages, while ScaRR works at a binary level.

Control-Flow Integrity. In the last few years, some authors have proposed architectures that share some similarities with RA [23, 27, 33]. These works are composed by two concurrent processes: a target process (that might be under attack), and a monitor process (that validate some target property). However, ScaRR considers a different attacker model since we consider a fully compromised user-space, *i.e.,* an attacker may tamper with the target software code or attack the monitor process itself. Moreover, unlike ScaRR, these solutions are not designed to provide any report about the execution path of the target process.

10 Conclusion

In this work, we propose ScaRR, the first schema that enables runtime RA for complex systems to detect control-flow attacks generated in user-space. ScaRR relies on a novel control-flow model that allows to: (i) apply runtime RA on any software regardless of its complexity, (ii) have intermediate verification of the monitored program, and (iii) obtain a more fine-grained report of an incoming attack.

We developed ScaRR and evaluated its performance against the set of tools of the SPEC CPU 2017 suite. As a result, ScaRR outperforms existing solutions for runtime RA on complex systems in terms of attestation and verification speed, while guaranteeing a limited network traffic.

Future works include: investigating techniques to extract more precise CFG, facing compromised operating systems, and studying new verification methods for partial reports.

Acknowledgments

This work was partly supported by the SUTD start-up research grant SRG-ISTD-2017-124 and by the European Commission under the Horizon 2020 Programme (H2020), as part of the LOCARD project (Grant Agreement no. 832735).

References

[1] Bzip2, 2002. Last access March 2019.

[2] Amazon web services (aws), 2006. Last access March 2019.

[3] Microsoft azure, 2010. Last access March 2019.

[4] Blake2, 2013. Last access March 2019.

[5] Mcsema, 2014. Last access Feb 2019.

[6] Zstandard, 2016. Last access March 2019.

[7] Zlib, 2017. Last access March 2019.

[8] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[9] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems.

[10] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[11] Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *BLAKE2*, pages 165–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] Haihe Ba, Huaizhe Zhou, Shuai Bai, Jiangchun Ren, Zhiying Wang, and Linlin Ci. jmonatt: Integrity monitoring and attestation of jvm-based applications in cloud computing. In *Information Science and Control Engineering (ICISCE), 2017 4th International Conference on*, pages 419–423. IEEE, 2017.

[13] Katelin A Bailey and Sean W Smith. Trusted virtual containers on demand. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, pages 63–72. ACM, 2010.

[14] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

[15] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.

[16] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[17] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, pages 385–399, 2014.

[18] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] Dino Dai Zovi. Practical return-oriented programming. In *SOURCE Boston*, 2010.

[20] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.

[21] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: Lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pages 106:1–106:8, New York, NY, USA, 2018. ACM.

[22] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[23] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 131–148, Berkeley, CA, USA, 2017. USENIX Association.

[24] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. An abstract domain of uninterpreted functions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 85–103. Springer, 2016.

[25] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 585–598, New York, NY, USA, 2017. ACM.

[26] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.

[27] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1470–1486, New York, NY, USA, 2018. ACM.

[28] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[29] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[30] Chongkyung Kil, Emre C Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 115–124. IEEE, 2009.

[31] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[32] E Jebamalar Leavline and DAAG Singh. Hardware implementation of lzma data compression algorithm. *International Journal of Applied Information Systems (IJAIS)*, 5(4):51–56, 2013.

[33] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1635–1648, New York, NY, USA, 2018. ACM.

[34] Gian Filippo Pinzari. Introduction to nx technology, 2003.

[35] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys Tutorials*, 19(2):1145–1172, Secondquarter 2017.

[36] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[37] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[38] Nathan P Smith. Stack smashing vulnerabilities in the unix operating system, 1997.

[39] William Stallings. The advanced encryption standard. *Cryptologia*, 26(3):165–188, July 2002.

[40] Paulus Stravers and Jan-Willem van de Waerdt. Translation lookaside buffer, December 10 2013. US Patent 8,607,026.

[41] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM systems Journal*, 39(1):175–193, 2000.

[42] Allan Tomlinson. Introduction to the tpm. In *Smart Cards, Tokens, Security and Applications*, pages 173–191. Springer, 2017.

[43] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, pages 86–106. Springer, 2012.

[44] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function hooks without code. In *NDSS*, 2014.

[45] Liangming Wang and Fagui Liu. A trusted measurement model based on dynamic policy and privacy protection in iaas security domain. *EURASIP Journal on Information Security*, 2018(1):1, 2018.

[46] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks. Technical report, University of Cambridge, Computer Laboratory, 2018.

[47] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[48] Fan Yao, Jie Chen, and Guru Venkataramani. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 467–470. IEEE, 2013.

[49] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 66–85. Springer, 2015.

[50] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.