Protecting Legacy Code against Control Hijacking via Execution Location Equivalence Checking

Tobias F. Pfeffer, Stefan Sydow, Joachim Fellmuth and Paula Herber
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
Email: {tobias.pfeffer,stefan.sydow,joachim.fellmuth,paula.herber}@tu-berlin.de

***Abstract*—Current anomaly detection systems that enforce control flow integrity based on control flow graph information are not able to precisely monitor dynamic aspects of execution. Consequently, they are typically too coarse-grained to comprehensively detect modern code-reuse attacks. Even when enriched with dynamic monitoring information such as shadow stacks, the heuristics used are either too imprecise or produce many false negatives. In this paper, we present a novel approach to establish control flow integrity in multi-variant execution through *execution location equivalence*. The concept of execution location equivalence allows us to precisely detect execution divergence using a diversified control flow model and, consequently, to detect a broad variety of code-reuse attacks. In this way, execution of position-independent executables can be reliably protected against a broad range of control hijacking attacks.**

***Keywords*-Security, control hijacking, control flow integrity, multi-variant execution**

I. Introduction

Memory corruption bugs and the resulting vulnerabilities have continuously threatened the software industry throughout the last decades. Although a variety of countermeasures have been proposed, new attacks keep defeating new solutions [1]. Much progress has been made concerning analysis and testing methods to identify and fix possible flaws during the software production process. Nevertheless, reliable protection of legacy common-of-the-shelf binaries remains an unanswered problem. Existing approaches in this area either fortify the applications beforehand (e.g., through binary rewriting) or monitor its execution to detect anomalies during run-time. The latter approach can again be subdivided into attack-specific strategies, policy-based techniques (e.g., control flow integrity checking), and, more recently, multi-variant execution.

Attack-specific strategies rely on intrinsic characteristics of the exploited vulnerability, and are therefore often very effective against a small selection of attacks. However, they have been proven to be circumventable through mutations of the attack vector. Because of their narrow attack definition, it is possible to efficiently escape the applied heuristic [2]. **Policy-based approaches** are more resilient to adaptations as their detection relies on common execution patterns. Thus, they cover a broader range of anomalies. However, discerning benign and malicious executions in a variety of applications requires precise reconstruction of the intended behavior. One common way to achieve this is static control flow graph construction, as used in control flow integrity systems. However, due to the lack of run-time information during the model creation phase, control flow integrity systems use an overestimation of the control flow graph to keep the false positive rate low. As a result, detection through these systems can be escaped at points of overestimation [3]. Monitoring systems based on **multi-variant execution** do not use a pre-constructed behavior model. Instead, they compare observed execution traces from a multitude of diversified replicas running on the same input. The idea is that diversified variants are sufficiently distinct to diverge when under attack, but equivalent under unimpaired execution. However, arguing the necessity to issue system calls for all applications that aim at exploiting the system, current multi-variant execution engines validate executions only at system call granularity [4]. Although this restriction helps to reduce the performance overhead of multi-variant execution, it also implies that coarse-grained multi-variant execution systems are still vulnerable to control hijacking attacks through memory corruption. When exploited with sophisticated control diversion schemes such as return oriented programming, this can potentially enable attackers to execute arbitrary code. Overall, a comprehensive security solution that inescapably ensures integrity of the control flow with a reliably low rate of false positives is still missing.

In this paper, we propose a combination of control flow integrity and multi-variant execution to detect control hijacking attacks with high precision.

In particular, we require our approach to fulfill the following criteria.

Comprehensiveness: Our approach is comprehensive in the sense that any diversion from the intended control flow is always detected as it happens. Consequently, our approach thwarts a broad range of attacks that hijack the control flow through memory corruption. This implies that no attack-specific characteristics or heuristics can be applied as they thwart only a snippet of malicious behavior, providing probabilistic security. Comprehensiveness can be expressed by a low rate of false negatives.

Precision: Whether a protection system can be widely used is determined mainly by its ability to correctly differentiate between normal and malicious behavior. Hence, to achieve high reliability, false positive errors should occur as little as

possible. This implies that underlying models need to be as precise as possible and encapsulate dynamic effects of the execution.

Performance: We are aware that fine-grained sound protection of control flow comes at the cost of performance. However, the past has shown that compromising security for performance gains often undermines the reliability or even enables evasion from the protection system. We keep the imposed performance penalty within reasonable bounds, but it is not the primary concern at this point. Thus, reducing the overhead is left for future work.

Usability: Our approach is automatically applicable without the need to instrument applications or modify the behavior of either the target or the underlying operating system kernel. As a result, our approach should be transparently applicable to stripped binaries.

Security: To reduce the vulnerability of our approach concerning information leakage attacks, randomized values should not be considered as a secret. Consequently, the system should be secure even to an aware adversary.

Our main idea to fulfill these criteria is to ensure *execution location equivalence* of two or more diversified variants. If the execution locations of two variants diverge, we conclude that the integrity of the control flow has been violated. This allows us to automatically classify observed behavior as intended or malicious on the fine-granular level of execution locations, i.e. on instruction level. We show that at run-time, it is sufficient to check execution location equivalence at basic block granularity. With that, we are still able to detect all divergences on instruction level, but the performance overhead is significantly reduced.

As a proof of concept, we have implemented a prototypical multi-variant execution engine, extended with a fine-grained validation stage that monitors execution traces on basic block level. The experimental results discussed in Section V demonstrate that our approach is more comprehensive than existing multi-variant execution (MVE) solutions, as it can detect divergences at finer granularity, and more precise than existing control flow integrity (CFI) solutions, as it uses an exact model instead of an overapproximation of the control flow graph (CFG). Additionally, our approach is widely applicable, as it only requires position-independence of the binary.

The rest of this paper is structured as follows: in Section II we give some background on attacks and countermeasures. We then introduce our approach in Section III and move on with a description of our implementation in Section IV. We present our evaluation results in Section V and discuss related Work in Section VI. Finally, we conclude our work in Section VII and present an outlook in Section VIII.

II. Background

In this section, we give an overview of the most common memory corruption vulnerabilities, resulting attack vectors and countermeasures. Because of their major importance for our approach, we introduce control hijacking attacks in more detail in Section II-A and present address space layout randomization in Section II-B. Subsequently, we outline control flow integrity (CFI) and multi-variant execution (MVE) systems as possible mitigation techniques in Section II-C and Section II-D. Finally, we describe the detection model of MVE systems as introduced by Cox et al. in their work on N-variant execution in Section II-E.

A. Control Hijacking Attacks

Among software vulnerabilities, memory corruption bugs are within the most common and most dangerous software faults. Through exploitation of these vulnerabilities, reading from and writing to mostly arbitrary locations of a process' memory is possible. In the worst case, an attacker can hijack the control and run arbitrary code inside the vulnerable process. Since unprotected operating systems cannot diagnose the process as compromised, the intrusion remains transparent and will continue with the same privileges as the exploited process.

To hijack the control of the execution, the attacker modifies control data that is stored in writable regions of the memory. Often, return addresses residing on the stack are targeted, as they can easily be located and have to be writable by definition. Alternatively, known function pointers can be adjusted to redirect the execution, e.g. in the global offset table (GOT). In the past, the most commonly exploited vulnerability was unsafe buffer handling. By precisely overflowing the buffer, adjacent control information can be altered [5]. With the ability to write attacker defined contents to arbitrary locations, format string exceptions constitute an additional threat that does not rely on buffer placement in memory [6], [7].

Up to the introduction of $W \oplus X$ strategies [8], control was typically redirected to injected instructions to realize complex functionality. However, with operating systems now commonly enforcing the non-existence of memory regions that are both modifiable by the user and executable by the processor, code-reuse attacks are on the rise. These attacks divert control information residing in writable pages to application or library code mapped in executable pages. This approach was first introduced in return-to-libc attacks, which focus on executing critical functionality directly through standard interfaces of commonly linked libraries [9]. In 2007, Shacham has proposed an enhanced method to reuse code for arbitrary execution, later called return oriented programming (ROP) [10], [11]. In ROP, control is redirected to short basic blocks ending with a return instruction. These so called gadgets can be chained to achieve arbitrary functionality. Further developments such as jump oriented programming (JOP) extend the gadget catalog, distinctly aggravating detection of code reuse attacks [12].

B. Address Space Layout Randomization

Control hijacking attacks rely on hard coded locations and addresses to modify control information in the memory. Therefore, modern operating systems enforce address space layout randomization (ASLR), as an approach to diversify the virtual address space. With ASLR, exploits will not behave equally among all executions. The authors of the Unix implementation state that "[ASLR] will make a class of exploit techniques fail

with a quantifiable probability and also allow their detection" [13]. In the current version, randomization is by default applied to the base addresses of dynamic libraries (which are compiled as position-independent code (PIC)) as well as the stack. However, it is not applied to the base address of the application, unless it was compiled as a position-independent executable (PIE). Because of the limited randomness on 32 bit machines, Shacham et al. have demonstrated the possibility of derandomization attacks with reasonable effort [14]. Additionally, Marco-Gisbert and Ripoll demonstrate that ASLR can even be broken on 64 bit machines using the Offset2Lib attack [15].

C. Control Flow Integrity

Conceptually, CFI is closely linked to program shepherding. In program shepherding, a range of policies are applied to prevent the execution of malevolent code [16]. For example, by restricting returns to call-preceded locations, the number of ROP gadgets is greatly reduced. In an effort to enable more accurate and restrictive policies, CFI extends this approach with information from the statically created CFG. Originally, every control transition instruction (CTI) is augmented with source and destination checking code that encodes the information directly in the application [17]. By validating control flow at every CTI, the system can ensure that control flow remains within the allowed model. However, the quality of such CFI systems depends largely on the precision of the underlying CFG. With indirect branches that determine their destination at run-time, over-approximations have to be added to the CFG "because it is difficult to resolve indirect branch targets statically" [18].

Finally, since the ordering of issued calls is not covered by the CFG, Abadi et al. proposed the additional use of a shadow call stack originally introduced in StackGhost [19], [17], [20]. With a backup of control flow information in a secure data region, returning to the calling function can often be assured. However, exceptional control flow complicates this approach.

D. Multi-Variant Execution

Combining the ideas of intrusion monitoring and fault tolerance through diversified redundancy, multi-variant execution (MVE) is an interesting new approach to secure execution. Instead of validating application behavior with a pre-defined behavior model (i.e., the CFG), the execution is compared to slightly modified but equivalent replicas of the target. These replicas function as input-specific run-time models of the execution behavior. During execution, the monitoring agent compares properties of the execution states of all variants to ensure equivalence of their behavior. The comparability is enabled by the introduction of rendez-vous points that act as execution barriers. To prevent side-effects due to the multi-execution, variants are typically synchronized at system call granularity. Using the master call strategy, only one variant executes environment-sensitive functionality and publishes the result to the other variants [21]. Additionally, this implies that all variants issue the same system call sequences under unimpaired execution, which can be checked by the monitoring agent. Consequently, an intrusion is detected through differences in the observed system call sequences, triggered by attacks that concern the artificial diversity introduced in the variants. Variant-specific attacks are hardly probable, because the single input is multiplexed to all executions. As a result, the discernability of benign and malicious execution is mainly impacted by the diversification schemes. Jackson et al. have identified nine variation techniques and two levels of monitoring granularity [4]. We discuss assorted approaches in detail in Section VI-C.

E. Detection Model

In their work on multi-variant execution systems, Cox et al. introduced a formal model for multi-variant execution engines to define the detection mechanism via normal equivalence [22]. While the full trace created during single execution can be described as a sequence of states (i.e. $[\sigma^0, \sigma^1, ...]$), redundant execution can be described as a sequence of state-tuples: $[< \sigma_0^0, \sigma_1^0, ..., \sigma_{N-1}^0 >, < \sigma_0^1, \sigma_1^1, ..., \sigma_{N-1}^1 >, ...]$ Here, a state is additionally annotated with the identifier of each replication. As the replications managed by a multi-variant execution engine are artificially diversified, their states logically differ even at the same step. To establish comparability among the variants, Cox et al. introduced the concept of a canonicalization function per variant, C_v, which removes diversification from a state, effectively mapping the *diversified* state σ_v^t to the *canonical* execution state σ^t at the same step t. Under normal execution, all variant states are equal to a common canonical state:

$$\forall t \geq 0, 0 \leq v < N, 0 \leq w < N : C_v(\sigma_v^t) = C_w(\sigma_w^t) = \sigma^t$$

Inversely, a state that can not be mapped to the canonical state via the canonicalization function is categorized as compromised. Furthermore, variants can enter an alarm state that describes observable anomalous behavior (e.g. crashing or variants out of sync at rendez-vous points). In this paper, we adapt the notion of normal equivalence and the concept of canonicalization to define *execution location equivalence*.

III. Control Hijacking Protection

With our work, we present a novel approach for the safeguarding of control flow integrity. The goal is to detect control flow hijacking attacks during the execution and thereby offer protection from a broad range of threats. Additionally, we aim to overcome the inherent overapproximation problem of CFG-based intrusion detection that often leads to a high rate of false positives. To achieve this, we shift the focus of the behavior model from a complete description of all possible execution paths (as expressed by the CFG) to the observed sequence of execution locations. In our solution, no statically built behavior model, heuristics, or global sanitation policies are assumed. Instead, multiple parallel executions are observed to ensure equivalence in their diversified execution location sequences. Consequently, we combine the most promising approaches, namely control flow integrity and multi-variant

execution, to establish fine-grained control flow protection without impairing normal execution by raising too many false positives. To achieve this, we develop a multi-variant execution engine capable of running two diversified replicas in system call lock-step. We then enrich this engine with rendez-vous points inserted at run-time on control transition instructions. Based on a canonicalization function, we create a validation stage that ensures that the diversified execution remains synchronized on instruction level.

The main advantages of our approach are threefold: firstly, similar to the most recent systems, our solution can be run without elevated privileges or adjustments in the kernel. It relies neither on a pre-learned model of intended behavior nor on static analysis but selectively constructs the model on-the-fly from the replicated execution state. Secondly, our approach can be considered secret-less as awareness of our approach does not grant an adversary reasonable advantages. Thirdly, and most importantly, the fine-grained nature of our approach allows detection of additional control hijacking attacks not traceable with existing coarse-grained solutions. These include (1) denial-of-service attacks based on infinitely executed loops and (2) code-reuse attacks via different paths. Our approach is promising, as it does not only protect system call locations, but also the possibility of reaching them in the scope of the current function and execution state. It is therefore a true enhancement of coarse-grained anomaly detection systems. It is also challenging, as the establishment of CFI has to be highly reliable without impairing normal execution, to be usable in a wide range of scenarios.

In the following subsections, we first introduce our novel concept of execution location equivalence and describe the requirements for the diversification scheme in Section III-A. Then, in Section III-B, we present our adaptations of the formal detection model based on normal equivalence and canonicalization presented by Cox et al. [22] (cf. Section II-E). Subsequently, we show that dynamic validation of execution location equivalence can be restricted to sequences of basic blocks to lower the imposed performance penalty without reducing comprehensiveness or precision of our approach. Finally, we elaborate on the detection of execution location divergences in Section III-C. The prototypical implementation of our concept is described in depth in Section IV and evaluated in Section V.

A. Instruction Location Diversification

With our novel concept of *execution location equivalence*, we introduce an adapted detection model based on Cox et al.'s work described in Section II-E [22]. Instead of the complete execution path defined as a sequence of states, we consider only the execution location information extracted from every state σ as the sequence of execution locations. To describe the execution locations of multiple variants we define L_v as the *diversified* execution location of variant v. Additionally, we use $C_v(L_v) = L$ as the *canonical* execution location, created through application of the specific canonicalization function C_v. To ensure that the comparison of execution location sequences yields meaningful results, all variants have to be correctly diversified. For this, we require three characteristics of our diversification scheme. Firstly, diversified instruction locations must be different for all variants:

$$v \neq w, L_v = L_w \Rightarrow C_v(L_v) \neq C_w(L_w) \tag{1}$$

If this requirement was not met, we could not guarantee distinction of address based attacks among our variants. Secondly, instructions at the same locations have to be equal after diversification to ensure equal semantics. For this, we define $I(L)$ as the instruction at the canonical location L and $I_v(L_v)$ as the instruction at the diversified location L_v.

$$C_v(L_v) = L \Rightarrow I_v(L_v) = I(L) \tag{2}$$

Thirdly, no instruction may be split across different diversification values, ensuring correct canonicalization of the code. To express this, we use the Δ function that calculates the size of an instruction, which may vary with the architecture [18].

$$C_v(L_v + \Delta(I_v(L_v))) = C_v(L_v) + \Delta(I_v(L_v)) \tag{3}$$

In Section IV-B, we show that these requirements are met by the standard ASLR implementation described in Section II-B. Hence, we can assume that all three requirements can easily be fulfilled, and we exploit them to reduce the performance penalty imposed by our approach. This is achieved by reduction of costly dynamic validation steps to basic block level as described in the following section.

B. Execution Location Equivalence

The concept of execution location equivalence (ELE), which we use to establish CFI in MVE systems, can be defined similarly to the normal equivalence property introduced by Cox et al. [22]. We adapt their definition to our approach and formalize the execution location equivalence property. Additionally, we show that execution location equivalence is invariant under execution of *fall-through instructions*, i.e., instructions that only have one immediate successor instruction. From this we conclude that it is sufficient to dynamically check the result of *control transition instructions*, i.e., of instructions that may transfer the execution to locations other than the immediate successor.

As outlined in Section II-E Cox et al. strive to "guarantee that the [...] system never enters a state-tuple that contains one or more variants in comprised [sic] states without any variants in alarm states" [22]. Their general normal equivalence property is satisfied if all variants at the same execution step are in the same normal state. However, since we are only interested in control hijacking attacks, we validate only the execution locations instead of the whole state at every step to ensure equivalent execution location sequences. Consequently, we define execution location equivalence as satisfied if all variants are executing the same canonical location at the same step. For this, we introduce L_v^i as the ith entry in the execution location sequence of variant v and L^i as the canonical execution

location at step i. Execution location equivalence can then be defined as

$$C_v(L_v^i) = C_w(L_w^i) = L^i \tag{4}$$

To ensure that no variants become compromised with regard to CFI at any point during execution (i.e., all variants follow the same canonical execution location sequence), execution location equivalence has to be true at every step. Fortunately, all fall-through instructions do not affect execution location equivalence. By definition, fall-through instructions (FTIs) have only one immediate successor (i.e. the subsequent instruction). As a result, these instructions cannot lead variants in normal state into a compromised state if all variants were correctly diversified. To show this, we define the successor function Γ for FTIs using the instruction size function Δ

$$\Gamma[\![\text{fti}]\!](L^i) = L^i + \Delta(I(L^i)) = L^{i+1} \tag{5}$$

We then assume that execution location equivalence initially holds and apply the successor function from Equation 5

$$\begin{aligned}
& \Gamma[\![\text{fti}]\!](C_v(L_v^i)) && = \Gamma[\![\text{fti}]\!](L^i) \\
\overset{5}{\Rightarrow}\ & C_v(L_v^i) + \Delta(I(C_v(L_v^i))) && = L^i + \Delta(I(L^i)) \\
\overset{2}{\Rightarrow}\ & C_v(L_v^i) + \Delta(I_v(L_v^i)) && = L^i + \Delta(I(L^i)) \\
\overset{3}{\Rightarrow}\ & C_v(L_v^i + \Delta(I_v(L_v^i))) && = L^i + \Delta(I(L^i)) \\
\Rightarrow\ & C_v(L_v^{i+1}) && = L^{i+1}
\end{aligned}$$

Thus, execution location equivalence is invariant under execution of fall-through instructions if instruction locations are diversified correctly. As a result, execution location equivalence only has to be verified dynamically for control transition instructions. Consequently, it is sufficient to check basic block equivalence at run-time without harming the comprehensiveness of our approach, as execution location equivalence can be inferred. This reduces the dynamic overhead to an upper limit of one instruction per basic block.

C. Execution Location Equivalence Preservation

While execution of fall-through instructions cannot introduce compromised variants, execution of control transition instructions (CTIs) is critical to execution location equivalence. In this section, we outline the conditions under which execution of CTIs preserves execution location equivalence.

Generally speaking, two categories of CTIs exist: (1) conditional control transition instructions (CCTIs) and (2) unconditional control transition instructions (UCTIs). Additionally, in x86 assembly, most CTIs have two addressing modes. The operand defined target is either interpreted as a relative offset to the program counter or as an absolute address in memory. To demonstrate this, we first define a successor function for unconditional CTI, using t as the target operand and m as the addressing mode. Conditional CTI execution is covered below.

$$\Gamma[\![\text{ucti } m, t]\!](L^i) = \begin{cases} t & \text{, if } m = absolute \\ L^i + t & \text{, if } m = relative \end{cases} \tag{6}$$

Using $\Gamma[\![\text{ucti } relative, t]\!]$, we can now show that execution location equivalence is preserved for relative unconditional CTIs if $t_v = t$ and

$$C_v(L_v) + t_v = C_v(L_v + t_v) \tag{7}$$

holds. This implies that unconditional CTIs with relative addressing preserve execution location equivalence if their operands are identical and, just as fall-through, do not target differently diversified locations. We assume execution location equivalence and apply the successor function

$$\begin{aligned}
& \Gamma[\![\text{ucti } relative, t_v]\!](C_v(L_v^i)) = \Gamma[\![\text{ucti } relative, t]\!](L^i) \\
\overset{6}{\Rightarrow}\ & C_v(L_v^i) + t_v && = L^i + t \\
\overset{7}{\Rightarrow}\ & C_v(L_v^i + t_v) && = L^i + t \\
\Rightarrow\ & C_v(L_v^{i+1}) && = L^{i+1}
\end{aligned}$$

Furthermore, UCTIs with absolute addressing can lead to execution location equivalent states independent of execution location equivalence at the current step. To show this, we assume absolute CTI execution and that t_v has a value such that $C_v(t_v) = t$ holds. This implies that UCTIs with absolute addressing establish execution location equivalence if their operands are identical under canonicalization. We now apply the successor function without assuming execution location equivalence

$$\begin{aligned}
& \Gamma[\![\text{ucti } absolute, C_v(t_v)]\!](L_v^i) = \Gamma[\![\text{ucti } absolute, t]\!](L^i) \\
\overset{6}{\Rightarrow}\ & C_v(t_v) && = t \\
\Rightarrow\ & C_v(L_v^{i+1}) && = L^{i+1}
\end{aligned}$$

Fortunately, CCTIs can be handled as a combination of an unconditional CTI and a fall through instruction. The two execution cases can be distinguished by the value of the condition argument b.

$$\Gamma[\![\text{ccti } b, m, t]\!](L^i) = \begin{cases} \Gamma[\![\text{ucti } m, t]\!](L^i) & \text{, if } b = true \\ \Gamma[\![\text{fti}]\!](L^i) & \text{, if } b = false \end{cases}$$

Hence, execution location equivalence holds at step $i+1$ if the condition is equally true among all variants and the application of the successor function for unconditional CTIs is correct, or if the condition is equally false among all variants and the application of the successor function for FTIs is correct. Under all other conditions not explicitly mentioned here, execution of conditional or unconditional control transition instructions will likely lead to compromised variants. This needs to be detected immediately to satisfy the detection property.

In this section, we have shown that shadow execution can be used to enforce CFI with high precision via checking execution location equivalence if the applied diversification scheme is correct according to our requirements outlined in Section III-A. To reduce the performance penalty imposed by validating execution location equivalence for the complete sequence of execution locations, we infer execution location equivalence from equivalent basic block sequences. To achieve

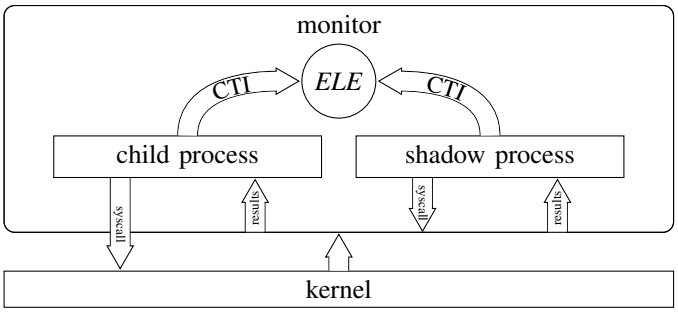

Fig. 1. System Overview

this, we have shown that execution location equivalence is invariant under execution of FTIs. Finally, we have demonstrated conditions under which execution of CTIs preserves execution location equivalence. In the following Section IV, we elaborate on our prototype implementation, which we use to evaluate our approach in Section V.

IV. Enhanced Multi-Variant Execution

To evaluate the feasibility of our approach, we have implemented our own prototypical multi-variant execution engine with enhanced control flow tracing capabilities as described in the last section, as well as a validation stage for execution location equivalence checking on basic block granularity. To keep the implementation effort within reasonable bounds, we restrict our proof of concept engine to a subset of system calls and take advantage of Linux' standard ASLR feature for memory address diversification. The engine itself is based on the `ptrace` API as it supports process tracing both at system call and breakpoint granularity. It is also used in most existing MVE engines. For the implementation and detection of instruction level rendez-vous points, we employ standard x86 software breakpoints along with Capstone as our disassembly engine which is based on the LLVM project and is backed by Intel [23]. In the following subsections, we first outline our MVE engine in Section IV-A and show that ASLR satisfies the diversification requirements for our approach in Section IV-B. Then, we show how compromised states are detected with our detection model based on four attack classes in Section IV-C.

A. Multi-Variant Execution Engine

To implement a proof of concept MVE engine, we have first implemented a coarse-grained multi-variant execution engine using two variants with address diversification. We have then added fine-grained monitoring capabilities to establish execution location equivalence. As depicted in Figure 1, the monitoring agent spawns an additional shadow execution copy along with the target application and traces them using the `ptrace` API. Behavior synchronization of original and shadow execution is achieved by interception of all system calls issued during execution. According to Volckaert et al., proper handling of I/O-related system calls is "sufficient to replicate simple single-threaded programs" [21]. These system calls are handled using the master call strategy to ensure that no side effects from execution duplication are visible to the operating system or user. Input duplication is achieved by copying all results from the original execution to the shadow execution. This includes the return value and modifications to output parameters passed as pointers via argument registers. Some non-I/O-related system calls have to be executed by both spawns as their execution relies on their side effects. For example memory allocation and deallocation (e.g. via `brk()`, `mmap()` or `munmap()`) should not be replicated as their results differ due to memory diversification. Furthermore, some non-I/O system calls need additional care to prevent race conditions among the variants as they can only be executed once (e.g. file opening with `O_CREAT` flag). These system calls are patched and sequentially executed. Rendezvous points at system call granularity are also used to detect behavioral differences. This guarantees that both executions issue the same system call sequence similar to the intrusion monitoring approach proposed by Wagner and Dean [24].

To establish *excution location equivalence*, additional synchronization points are inserted at run-time. We add synchronization points at the end of the currently executed basic block, skipping over all FTIs. We disassemble all subsequent instructions starting from the current execution location in the child process using the Capstone disassembly framework [23]. Once a CTI is reached, we assume the exit of a basic block and set a breakpoint in both the child and shadow process. The breakpoint location for the shadow process is calculated from the diversified child location using the canonicalization function described in Section IV-B.

On every triggered breakpoint, execution location equivalence is tested to detect any divergences in the basic block sequences of our child and shadow execution. If a compromised state is observed in one variant, an alarm is raised and execution halted. Should, at any point during tracing, an unresolved breakpoint persist until a system call is issued, or should more than one breakpoint per process exit, we trigger a (possibly false positive) alarm as protection of the observed execution path cannot be guaranteed. Furthermore, crashing of variants is also observed and used to detect compromised states. In Section IV-C, we describe the detection of control divergence, divide the range of possible control hijacking attacks into four attack classes, and assess their hazardousness.

B. Diversification with ASLR

As elaborated in Section III, distinction of compromised states in our variants relies on correct diversification. Strictly speaking, we require that all equations in Section III-A hold. For the sake of our proof of concept implementation, we decided to rest our diversification on the ASLR feature provided by the operating system. With this simple diversification scheme, canonicalizing known instruction pointers can be done quite efficiently using the randomization offsets. The translation offset is calculated from the randomization bases used in the child and shadow execution: $O_{ASLR} = Base_c - Base_s$.

As a result, the canonicalization function for the child process is a simple subtraction

$$C_c(L_c) = L_c - O_{ASLR} = L_s = C_s(L_s) \tag{8}$$

This implies, that the shadow location L_s is used as the canonicalized location while the child location L_c is considered the diversified location. As long as the bases are different, Equation 1 is fulfilled

$$O_{ASLR} \neq 0 \Rightarrow C_c(L) = L - O_{ASLR} \neq C_s(L) = L$$

Additionally, since ASLR only shifts addresses but does not randomize locations on instruction level, Equation 2 is also true

$$C_c(L_c) = L_c - O_{ASLR} = L_s \Rightarrow I_c(L_c) = I_s(L_s)$$

As under Linux the kernel is always mapped to the highest addresses and not affected by ASLR, Equation 3 is also true as no overflow is possible

$$\begin{aligned} C_c(L_c + \Delta(I_c(L_c))) &= L_c + \Delta(I_c(L_c)) - O_{ASLR} \\ &= C_c(L_c) + \Delta(I_c(L_c)) \end{aligned}$$

and $C_s(L_s + \Delta(I_s(L_s))) = C_s(L_s) + \Delta(I_s(L_s))$, respectively.

Consequently, diversification is correct for all locations within the dynamic library regions of position-independent code if the randomized bases are different.

C. Divergence Detection

With our approach to combine CFI and MVE, we establish fine-grained control flow protection by checking execution location equivalence on basic block granularity. With that, we achieve a high comprehensiveness without impairing normal execution by raising too many false positives. For this, we define two attack dimensions and derive four different attack vectors. The attack dimensions are the addressing mode and the invasivness of the attack.

Combined, four attack classes on MVE with CFI are conceivable: (1) absolute full overwrite, (2) absolute partial overwrite, (3) relative full overwrite and (4) relative partial overwrite. In the first and second case, correctly diversified addresses cannot be changed without detection. If we assume that $C_c(L_c) = L_s$ holds before corruption and L_c is to be replaced with another value L_x such that $C_c(L_x) = L_s$, we conclude that $L_c = L_x$. Hence, whether partially or fully overwritten, the value at L_c cannot be changed if the value at L_s is not adapted as well. In the third case, we assume that both L_c and L_s are replaced with another value L_x and conclude that

$$C_c(L_x) = L_x - O_{ASLR} = L_x = C_s(L_x) \Rightarrow O_{ASLR} = 0$$

which is not possible under correct diversification as described in Section IV-B. Unfortunately, relative partial overwrite attacks, case four, do pose a threat to our approach. If we assume that Equation 8 holds before corruption and L_c and L_s are modified to L_x and L_y, such that

$$L_c - L_x = L_s - L_y \tag{9}$$

holds, our detection potentially fails

$$\begin{aligned} & C_c(L_x) = L_x - O_{ASLR} \\ \overset{9}{\Rightarrow} \quad & C_c(L_x) = L_y - L_s + L_c - O_{ASLR} \\ \overset{8}{\Rightarrow} \quad & C_c(L_x) = L_y = C_s(L_y) \end{aligned}$$

As a result, if memory locations relative to correctly diversified addresses are partially manipulated, control flow hijacking is still possible. However, the restriction that L_c and L_s are modified to values with the same distance has severe implications for the attack. We discuss the feasibility and impact of relative partial overwrite attacks on our system and show possible refinements to our approach to thwart these attacks in Section V.

V. Evaluation

For the evaluation of our approach and implementation, we apply it to a set of case studies and assess the fulfillment of the criteria formulated in Section I. We review the detection comprehensiveness by simulation of the four attack vectors described in Section IV-C in a simple test application in Section V-A. Furthermore, we evaluate the precision of our implementation with a current `coreutils` version for Linux in Section V-B. Finally, we have measured the performance of our implementation and discuss its scalability and identify possible sources of slowdown in Section V-C. In each Section, we also shortly discuss possible steps to further increase the quality of our approach or implementation.

A. Comprehensiveness

Measuring the comprehensiveness of an anomaly detection system tends to be a complicated task. Since perfect comprehensiveness is achieved when every possible malicious action is correctly classified as such, assessing the comprehensiveness would require knowledge of all possible malicious actions. However, attackers keep inventing new methods to compromise applications and hijack the control flow. As a result, instead of evaluating a range of known attacks, we examine the detection rate of the four general attack classes on multi-variant execution system described in Section IV-C. Perfect comprehensiveness in the context of these attack classes would require the immediate detection of compromised states in either of our variants.

To simulate the four attack classes, we have created a test application. It is able to hijack its own control flow either via absolute or via relative addressing and uses partial or full overwrite. Addresses are leaked to and read from a temporary file (temporarily disabling buffer content comparison). As file writing is handled using the master call strategy (cf. Section IV-A), only the child execution leaks information. Upon reading from the file, the previously leaked address is duplicated to both variants, thereby simulating an address specific attack. The target application features one sensitive function capable of spawning a shell, one benign function and a function pointer that decides which functionality is executed. An attack is considered successful if the sensitive function is

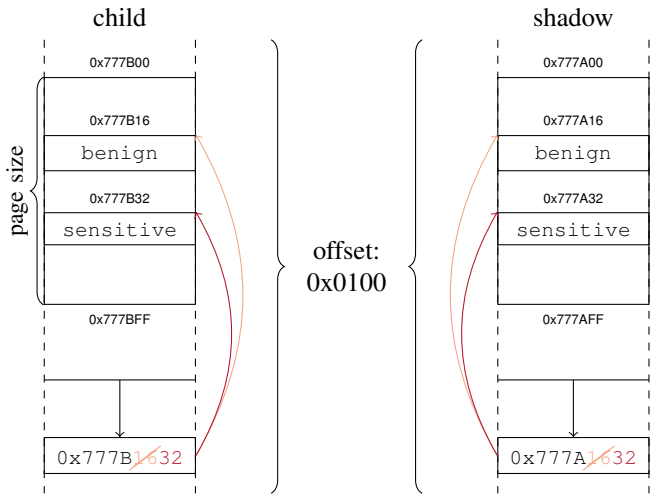

Fig. 2. Illustration of a relative partial overwrite attack

executed and a shell is created. In the following, we show that this is only possible using relative partial overwrites, demonstrating heightened comprehensiveness of our approach when compared to traditional MVE approaches, which detect malicious behavior on system call granularity. Additionally, we present a potential fix for this last vulnerability proposed by Bruschi et al. [25].

Absolute addressing: To simulate absolute addressed attacks like format string exploits (classes (1) and (2) in Section IV-C), we leak the address of the function pointer from the child execution to the shadow execution. Both variants then use this address to manipulate the memory in their respective contexts. Since the child address is typically not mapped in the shadow execution, writing to it will raise an error. Internal error checking routines consequently lead the shadow to a different execution path, which is immediately detected by our monitoring agent. The execution is halted and the attack successfully prevented (no shell is spawned).

Relative addressing, full overwrite: For the third attack class, we simulate relative addressing (e.g. achieved via buffer overflow exploits) by using the function pointer's address directly in both variants. Hence, its location in memory is individually manipulated in the correctly diversified form in both executions. However, since the attacker cannot specify the input selectively for each variant, both are overwritten with the address of the sensitive function leaked from the child execution. As a result, calling the function pointer is immediately detected by our engine as the call targets are equal among both variants. This observation implies that either the application is under attack or ASLR is not working correctly in which case we cannot enforce CFI any longer. Again the execution is halted and the attack unsuccessful.

Relative addressing, partial overwrite: The previous attack class can be refined by overwriting only the parts necessary to shift the function pointer from the benign function to the subsequent sensitive function. Instead of the complete function pointer, only the n least significant bits are overwritten with values from the child's leaked sensitive function address. Logically, if n is chosen large enough, this attack is identical to a full overwrite and detected as such. In the case of our 64 bit Linux, the most significant three bytes in the function pointer are always equal. Thus, any attack with $n >= 5$ is essentially the same as a full overwrite attack. For $1 < n < 5$, a corruption is detected as the function pointer cannot be canonicalized correctly. Finally, if $n = 1$, only the least significant byte is overwritten. Unfortunately, this byte is not protected by ASLR as the lowest 12 bits in the ASLR-offset have to be zero to align pages in the memory. As a result, partial overwrites restricted to page boundaries cannot be detected by our prototype. Figure 2 illustrates a relative partial overwrite attack on a system with a page size of 256 bytes. Since the benign and sensitive functions are located on the same page, Equation 9 holds

$$0x777B16 - 0x777B32 = -0x16 = 0x777A16 - 0x777A32$$

Thus, the partial corruption of the code pointer relative to a correctly diversified base cannot be detected by our current implementation.

Although this example demonstrates that it is still possible to hijack the control flow under CFI protection in our current MVE engine due to the limitations of ASLR, there exist enhanced address diversification schemes that also consider partial overwrite attacks. To ensure that the virtual address ranges of all variants do not overlap, Cox et al. introduced address space partitioning (ASP) [22]. ASP follows a stricter form of our diversification correctness requirement described in III-A (Equation 1) were L_v can never be equal to L_w. This strategy has the benefit of certainly crashing variants if absolute addresses are overwritten as no address can be valid in all variants at the same time. It can be applied to our approach by using an offset value greater than the highest address used in the child variant excluding the kernel stack (e.g., by flipping the most significant address bit). Independently of Cox et al., Bruschi et al. defined their own version of ASP which they called non-overlapping address spaces [25]. On account of the potential partial overwrite vulnerability originating from unaffected bits in the diversification, their scheme does not only guarantee completely disjoint address spaces but additionally shifts all mappings by k bytes. They note that their strategy is "defeating or at least strongly thwarting partial address overwriting attacks." It remains to show that this additional protection can be added to our approach without harming the precision.

B. Precision

To measure the false positive rate of our implementation, we used sample runs of the most current version of the `coreutils` binaries for Linux (version 8.24). All binaries were compiled as PIE to ensure full ASLR protection. This being the only additional effort, we can ensure wide usability as all further steps work fully automated. The `coreutils` package includes 103 binaries. With our implementation, we are able to run 89 different binaries with a total of over 7.3 million

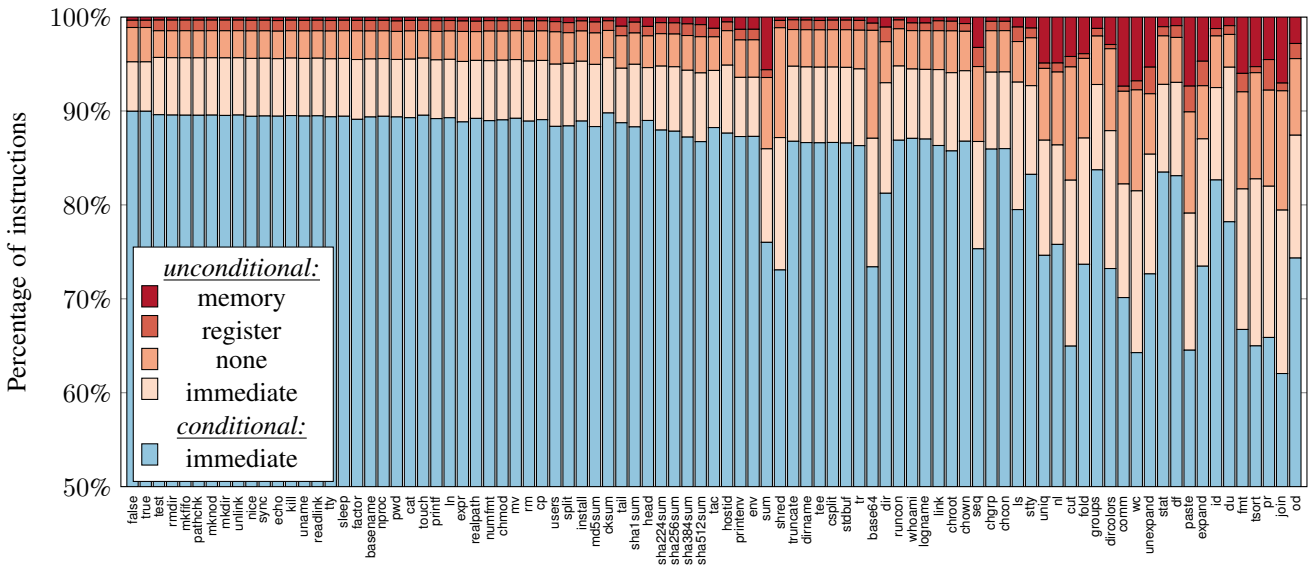

Fig. 3. Breakdown of observed CTI types

checked CTIs without any false positive alarm, demonstrating high precision. Most false positive alarms in the excluded 14 binaries can be attributed to usage of the virtual dynamic shared object (VDSO) which is randomized with a different base than PIC in standard ASLR. Additionally, `timeout` uses `timer_create()` which we have not implemented yet. We note that our tests cannot reflect every possible execution path in the applications. Instead, these results show that for every binary at least one execution passes without raising false positive alarms. For example, if files were requested as arguments, we used our `.bashrc` file. The amount of checked CTIs per run ranges from roughly 20.000 to over 400.000, further promoting high precision. Figure 3 provides a breakdown of observed CTI types checked per binary execution. With conditional CTIs using an immediate operand, two-destination instructions are by far the most frequently used kind of control flow transition throughout all executions. This reflects good synchronization of our variants as in all cases both variants followed the same path, implying equal execution conditions at every conditional CTI. Among the unconditional CTIs, single-destination instructions using an immediate operand are the most common, closely followed by operand-less many-destination instructions. Finally, we compared the amount of call instructions to the amount of return instructions. In all runs, at least four call instructions were not matched by returns with a maximum of eight unmatched calls in the execution of `csplit`. While the minimum of four can potentially be attributed to different call stack depths at attaching and detaching of the monitor, the remaining unresolved calls would potentially harm shadow stack protection. However, they are correctly validated by our implementation, not raising false positive alarms. As described in Section IV we restricted our proof of concept engine to a subset of system calls, which currently prevents the evaluation of more complex examples.

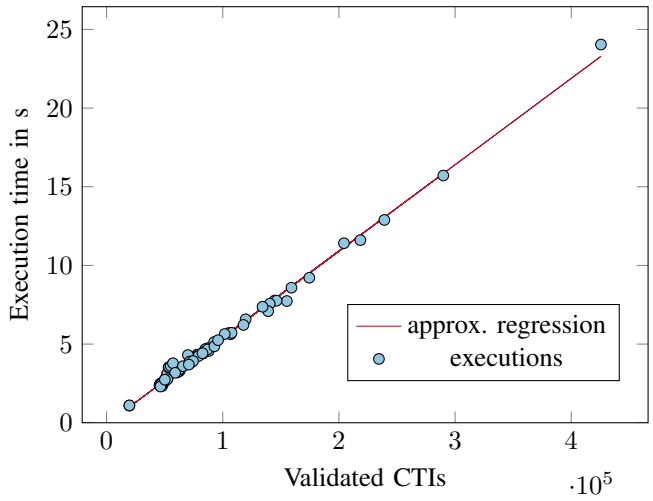

Fig. 4. Execution time over validated CTIs

C. Performance

Using the `coreutils` binaries, we measured the execution time of each sample run and compared our values to the amount of validated CTIs. We note that our multi-execution engine relies on full duplication of both the memory contents as well as the execution. The overhead therefore depends on the availability of system resources such as idle cores and memory. As stated in Section I, the performance was not the main concern at the current stage, currently leading to an average overhead of 1 to 1000. Nevertheless, from Figure 4 we infer linear scaling of the execution time with the number of validated CTIs. This demonstrates that execution location equivalence checking should be possible with reasonable performance once our proof of concept prototype has matured to a stable release.

In an effort to break down the performance penalty imposed by our prototype, we used `perf` to measure CPU cycles spent in different areas of the code. The results show that around 80% of all CPU cycles are spent within our framework. These can be divided into around 60% spent in the `ptrace` function, and 15% spent in Capstone, including all system calls issued from these functions. This reinforces the claim that less frequent synchronization is better for performance. Furthermore, Volckaert et al. remark that "for comparing and replacing system call argument, the `ptrace` API is extremely slow" [21]. They propose using the faster `process_vm` API capable of inter-process copying since Linux 3.2. Alternatively, our approach could be merged to the kernel to reduce the number of costly context-switches introduced by `ptrace`. In our sample runs, synchronization through our framework lead to 500.000 context-switches on average.

Overall, we have shown that our approach is both comprehensive, as it reliably thwarts three of four attack vectors and severely limits the power of the remaining fourth, and precise, as correctly validated over 7.3 million CTIs. Additionally, we have identified possible improvements for each criteria, which for the most part are implementation issues not yet handled by our proof of concept engine.

VI. Related Work

In this Section, we provide an overview of work related to the detection or prevention of code-reuse attacks such as ROP. In Section VI-A, we examine heuristic approaches that are designed specifically to counter ROP attacks. The Section is followed by an overview of modern CFI enforcing systems in Section VI-B, and finally by a review of current MVE engines in Section VI-C.

A. Code-Reuse Detection

To mitigate the growing threat of code-reuse based attacks, accelerated by the creation of fully automated tools for the discovery and chaining of gadgets, various code-reuse detection systems have been proposed. Since early code-reuse attacks based on ROP relied solely on return statements to maintain control over the hijacked executable, ROPdefender uses return-address protection to detect and prevent ROP attacks [26]. Modifications of return addresses are detected using a shadow stack (as proposed by [19]). It does, however, not verify indirect jumps used in more recent JOP gadgets. Chen et al. proposed additional rules for their gadget detection system, including call integrity and jump integrity checks [27]. With these rules, they are able to detect gadget chains that transfer execution somewhere other than a typical function prologue and indirect jump instructions leading outside of the originating function's scope. Following their work, Ivan Fratric proposed ROPGuard, a system for the prevention of ROP attacks at run-time [28]. Based on the observation that any malicious code has to call critical library functions to interact with the operating system, ROPGuard reduced the validation frequency to entry points of sensitive functions. Furthermore, future execution of return instructions is approximated through a partial simulation of the execution flow to detect gadget chaining as early as possible.

To achieve a low-overhead gadget detection system, Pappas et al. propose a technique relying on hardware features of modern processor architectures [29]. Upon entering a critical function, kBouncer detects chaining of gadgets by leveraging the history of indirect branches from last branch recording (LBR). To extend the approach, ROPecker additionally inspects the future execution flow [30] similar to ROPGuard. Furthermore, a sliding window technique is applied that sets all inactive pages to non-executable, assuming that ROP attacks typically do not follow expected temporal and spacial locality of applications.

In 2014, Schuster et al. have evaluated the effectiveness of the last three systems mentioned above [2]. They show that the limited amount of entries in the LBR registers renders kBouncer ineffective if benign LBR-flushing gadgets are added to the ROP chain. The authors can provide sample exploits that evade all three examined anti-ROP defense systems, concluding that they "can be bypassed in generic ways with little effort by aware adversaries."

Finally, we consider ROPStop, an anti-ROP system using conformant program execution based on CFG recovery through static binary analysis [31]. Although the authors note that ROPStop produced no false-positives on modern applications and incurred only very little overhead, the underlying CFG is overestimated whenever indirect control flow instructions are encountered, making the tool potentially vulnerable to JOP-based attacks.

B. Control Flow Integrity Systems

In 2013, Zhang et al. have proposed a CFI system that collects indirect control-flow targets using a dedicated and secured "springboard section" directly in the binary. This technique is called compact control flow integrity and randomization (CCFIR) and can be applied to all binaries compiled with sufficient information in relocation tables [32]. Control flow integrity is enforced through redirection of every indirect control transition through the springboard section. This implies that traditional ROP attacks cannot return to arbitrary gadget locations, as they rely on return instructions which have to dereference their targets from the springboard section. The springboard section is additionally randomized on start-up to harden location guessing attacks. However, as any protection guarded by secret randomization, it is vulnerable to information leakage attacks. Recent research has shown that CFI can also be applied to common of-the-shelf (COTS) binaries without additional information such as relocation information by using costum disassembly techniques [33]. However, the indirect control flow targets are overapproximated to account for compiler optimization effects. This relaxation makes the approach less comprehensive than the original work by Abadi et al. [17], [20]. In 2011, Bletsch at al. have shown that CFI can be used to mitigate code-reuse attacks in a cost-effective way [34]. Their technique, called control flow locking (CFL), is based on lock values in memory, which secure ongoing control

flow transfers through the means of transactions. This ensures that control flow is never redirected to targets whose locks were not set beforehand or through branching instructions whose locks have not been cleared before. Although a very promising approach, it requires access to the source code.

In [3], Göktaş et al. remark that reliable CFG creation is hard in the absence of source code. Hence, they demonstrate a way to detect and chain gadgets of sufficient variety to stage an attack under CFG-based CFI protection. The authors suggest the use of run-time mechanisms for checking that functions return to their caller to complement CFI towards comprehensive protection. They state that "there is no question that gathering such information comes at a cost in performance, but neglecting to do so comes at the cost of security." Davi et al. support the view of Göktaş et al. in [35]. In their work, they categorize existing coarse-grained approaches referencing kBouncer, ROPecker, ROPGuard and CFI for COTS binaries. The authors devise a combined approach, enforcing the most restrictive policies and show that evasion is still possible, albeit aggravated. Just as Göktaş et al., they state that "we should not sacrifice security for small performance gains."

An additional problem arises with exceptional control flow. This is also mentioned by both Göktaş et al. and Davi et al., who agree that relying on statically gained information is not sufficient to guarantee control flow integrity in all cases. Göktaş et al. state that even in its ideal form, CFI based on a finite, static CFG "cannot guarantee that a function call returns to the call site responsible for the most recent invocation to the function" [3]. Hence, they recommend the maintenance of a shadow stack, to strengthen the system. However, in their paper, Zhang and Sekar give a list of exceptional corner cases, which complicate the creation and maintenance of a shadow stack and indirect control flow targets [33], including cases where returns are used as jumps (e.g. for thread switching or signal handling), and generation of indirect control flow targets at run-time through the use of dynamic linking. If not handled correctly, these corner cases can introduce false positives or even false negatives.

C. Multi-Variant Execution Engines

Various diversification schemes have been tested in previous MVE engines. Salamat et al. have proposed a reverse stack growth variant to counter stack based buffer overflow and code injection attacks [36], [37]. Berger et al. have introduced randomized allocation of heap memory to counter heap overflow attacks with probabilistic certainty [38]. In the presentation of their N-Variant System, Cox et al. outline an approach using, among others, randomized diversity for the full address space layout [22]. They argument that, unlike single-execution ASLR, a low range of randomization values does not weaken their protection from a range of attacks as it does not rely on confidentiality. Similar to their approach, Bruschi et al. introduce a multi-variant execution framework based on address space diversification, which enriches the approach with protection against partial overwrite attacks [25]. Finally, the most recent iteration, GHUMVEE, follows a more economical approach with a disjoint code layout diversification scheme that does not guarantee randomization for data sections [21]. With disjoint code layout (DCL), any page with execution permissions is guaranteed to be mapped to its own base address, rendering all addresses in the mapping range valid in only one variant. To the best of our knowledge, all existing systems monitor execution at system call granularity. Thus, they cannot guarantee execution location equivalence among variants.

VII. Conclusion

In this paper, we have presented a novel approach to protect legacy code against control hijacking attacks. To achieve this, we employ a shadow execution model that is dynamically built at run-time using multi-variant execution. Unlike other multi-variant execution approaches, we validate the variants at instruction level to establish *execution location equivalence*. The main advantage of our approach is the use of a very precise model that enables strict control protection with a very low probability of false positive alarms. As a further optimization, we have shown that equivalence of execution location sequences can be inferred from equivalence of basic block sequences which reduces the number of necessary checks dramatically. In order to evaluate the fulfillment of our comprehensiveness, precision, and performance criteria, we have implemented a proof of concept multi-variant execution engine and enriched it with control flow tracing and divergence detection routines. Additionally, we have divided the range of possible control hijacking attacks into four classes. With the help of a small test application capable of simulating all four attack classes, we assessed the comprehensiveness of our approach as very high. We have shown that our prototype reliably thwarts three of the four attack vectors and severely limits the power of the remaining relative partial overwrite attack. Furthermore, we have evaluated the precision of our approach by monitoring sample executions of binaries from the `coreutils` package. Since our prototype applies standard ASLR for the diversification of instruction locations, all applications had to be recompiled as position-independent executables, marginally restricting the applicability of our implementation. In 89 out of 103 runs, no false positive alarms were raised. This corresponds to 7.3 million correctly audited control transition instructions. Our findings support the statement by Volckaert et al. that "even though there are many issues that arise when implementing an MVEE, there are very few fundamental limitations to the programs that can be run inside an MVEE" [21] – even when synchronized on instruction level to prevent control hijacking. Overall, our approach can be used to reliably and comprehensively protect a broad range of applications, including legacy code, against control hijacking attacks.

VIII. Future Work

As demonstrated in Section V, our prototypical implementation already achieves high comprehensiveness and high precision. However, we have identified a range of improvements

to the implementation that are left for future work. With regard to the comprehensiveness of our approach, probably the most effectual enhancement is the addition of a byte shift key to ASLR as implemented by Bruschi et al. [25] in their version of address space partitioning as described in Section V-A. If the necessity of equal page alignment in memory layouts was removed, the less significant bits could be protected by our approach as well. Although we already achieve a good precision, covering virtual dynamic shared objects (VDSO) in the divergence detection mechanism would further lower the rate of false positives. Finally, the overall implementation design needs to be improved, e.g., to become suitable for usage in production system environments. Here, switching from `ptrace` to the `process_vm` should improve the performance significantly. Alternatively, our approach could be ported to the extensive GHUMVEE framework once it is released as open-source. This would, however, require investigating the compatibility of our approach and disjoint code layout as instruction location diversification scheme.

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[2] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 88–108.

[3] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[4] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz, "On the effectiveness of multi-variant program execution for vulnerability detection and prevention," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*. ACM, 2010, p. 7.

[5] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 14, 1996.

[6] T. Newsham, "Format string attacks," 2000.

[7] Gera and Riq, "Advances in format string exploitation," *Phrack magazine*, vol. 11, no. 59, p. 7, 2002.

[8] P. Team, "Non-executable pages design & implementation," *http://pax.grsecurity.net/docs/noexec.txt*, 2003.

[9] S. Designer, "Getting around non-executable stack (and fix)," 1997.

[10] H. 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*. ACM, 2007, pp. 552–561.

[11] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[13] P. Team, "Address space layout randomization (aslr)," *http://pax.grsecurity.net/docs/aslr.txt*, 2003.

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[15] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux," 2014.

[16] V. Kiriansky, D. Bruening, S. P. Amarasinghe *et al.*, "Secure execution via program shepherding." in *USENIX Security Symposium*, vol. 92, 2002.

[17] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[18] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," *University of California, Davis, Tech. Rep*, 2009.

[19] M. Frantzen and M. Shuey, "Stackghost: Hardware facilitated stack protection." in *USENIX Security Symposium*, vol. 112, 2001.

[20] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[21] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete rop attack immunity with multi-variant execution," 2015.

[22] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Usenix Security*, vol. 6, 2006, pp. 105–120.

[23] N. A. Quynh, "Capstone: Next-gen disassembly framework." [Online]. Available: http://capstone-engine.org/BHUSA2014-capstone.pdf

[24] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.

[25] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicã¿ for defeating memory error exploits," in *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*. IEEE, 2007, pp. 434–441.

[26] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 40–51. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966920

[27] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in *Information Systems Security*, ser. Lecture Notes in Computer Science, S. Jha and A. Mathuria, Eds. Springer Berlin Heidelberg, 2010, vol. 6503, pp. 140–155.

[28] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," 2012.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 447–462. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas

[30] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[31] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller, "Detecting code reuse attacks with a model of conformant program execution," in *Engineering Secure Software and Systems*. Springer, 2014, pp. 1–18.

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[33] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Usenix Security*, 2013, pp. 337–352.

[34] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 353–362.

[35] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium*, 2014.

[36] B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," in *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008, pp. 1–7.

[37] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 33–46.

[38] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *ACM SIGPLAN Notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.