Moving Target Defense Considerations in Real-Time Safety- and Mission-Critical Systems

Nathan Burow[†], Ryan Burrow[†], Roger Khazan[†], Howard Shrobe[*], Bryan C. Ward[†]
[†] MIT Lincoln Laboratory, [*] MIT CSAIL

ABSTRACT

Moving-target defenses (MTDs) have been widely studied for common general-purpose and enterprise-computing applications. Indeed, such work has produced highly effective, low-overhead defenses that are now commonly deployed in many systems today. One application space that has seen comparatively little focus is that of safety- and mission-critical systems, which are often real-time systems (RTS) with temporal requirements. Furthermore, such systems are increasingly being targeted by attackers, such as in industrial control systems (ICS), including power grids. The strict timing requirements of these systems presents a different design objective than is common in general-purpose applications – systems should be designed around the worst-case performance, rather than the average case. Perhaps in part due to these alternative design considerations, many real-time systems have not benefited from much of the work on software security that common general-purpose and enterprise applications have, despite the ubiquity of real-time systems that actively control so many applications we as a society have come to rely on, from power generation and distribution, to automotive and avionic applications, and many others.

This paper explores the application of moving-target defenses in the context of real-time systems. In particular, the worst-case performance of several address-space randomization defenses are evaluated to study the implications of such designs in real-time applications. These results suggest that current moving-target defenses, while performant in the average case, can exhibit significant tail latencies, which can be problematic in real-time applications, especially if such overheads are not considered in the design and analysis of the system. These results inform future research directions for moving-target defenses in real-time applications.

1 INTRODUCTION

The security, or lack there of, of critical real-time systems (RTS) has recently been highlighted by a series of high-profile attacks. For example, the cyber attacks on the Ukrainian power grid [9], and the Triton malware targeted against safety systems in a Saudi chemical plant [15] demonstrate the evolving threat malicious actors pose to RTS. These attacks targeted the software stack on real-time control systems as opposed to enterprise IT systems. As adversaries become more sophisticated, and target increasingly low-level systems where they can craft more targeted and controlled exploits, novel defenses are required to stop these threats. The consequences for failing to stop them are severe, these attacks pose real threats to life and limb, not merely revenue and reputation.

RTS present novel challenges for cyber defenders. In particular, they have strict temporal requirements that, if violated, can cause catastrophic failures. Consider, for example, the case of the Toyota recalls associated with sudden acceleration and numerous fatalities. Investigations of the Toyota electronic control systems found that the processor utilization of the supported workload was higher than that which could be proven *schedulable*, or guaranteed to meet all deadlines [25]. Cases like these demonstrate the dire need for real-time performance for safety-critical RTS, and in turn, motivate the need for defenses that will not violate the underlying real-time performance requirements of the system.

The key distinguishing factor between RTS and general-purpose systems when designing software-security measures is that RTS must undergo rigorous testing and analysis to ensure both the logical correctness of the application, as well as the temporal correctness of the system overall. Depending upon the criticality of the application, there are varying degrees of such testing and analysis, ranging from strict hard real-time applications, which are common in safety-critical domains such as avionics, to soft real-time and best-effort applications, which can tolerate varying degrees of limited temporal overruns. One commonality, however, among all of these such real-time applications is the desire to design for the worst- or near-worst-case performance to ensure that computations can keep up with the pace of the physical world. As an example, in highly safety-critical software in avionics, *schedulability analysis* is often employed to mathematically guarantee that all temporal requirements will be satisfied at runtime, and these analyses are often based on static worst-case execution time (WCET) analysis to determine the worst-case runtime of an application.[1] Furthermore, many real-time applications are also embedded, and have significant Size, Weight, and Power (SWaP) constraints.

Perhaps because of many of these rigorous worst-case oriented design requirements, the security of RTS has been largely ignored by the research community, with recent investigations of embedded systems [11, 12, 33, 43] being the closest line of work of which we are aware. Existing work on memory safety for general purpose systems [30] is poorly suited to RTS as the overhead imposed – 100%

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

MTD'20, November 9, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8085-0/20/11.
https://doi.org/10.1145/3411496.3421224

[1] WCET analysis in general is undecidable, as it is very similar to the halting problem. Safety-critical code is often restricted to design patterns that are more easily analyzable. For example, dynamic memory allocation and recursion are often disallowed, and all loops must be statically bounded.

or more is typical – makes them ill suited to SWaP-constrained RTS. Fuzzers [26] in combination with sanitizers [40] are a best-practice for finding bugs during development, but do not protect deployed code. While we strongly encourage the use of fuzzers and santizers by developers, the stakes involved with many RTS, *e.g.,* physical harm, require the development of additional runtime defenses. Control-flow integrity (CFI) [1, 7] and Code-pointer integrity (CPI) are the lowest overhead deterministic defenses to emerge in the past decade. They are designed to prevent attackers from modifying function pointers and virtual calls, but these design patterns may be less common in safety-critical systems as they complicate static WCET analysis.

Randomization-based defenses are a promising class of general-purpose defenses. Randomization defenses offer low overhead and can protect against both control-flow hijacking and data-only attacks. The downside to using randomization defenses is that they are bypass-able with information leaks [37, 44]. What randomization lacks in iron-clad security guarantees it makes up for in low overhead. These low overheads are compelling in SWaP-constrained applications, and so in this paper we investigate the real-time impacts of moving-target defenses (MTDs).

A major challenge in adopting randomization defenses for RTS is the variability of performance times across randomizations. To quantify this issue, we evaluate four different code randomization defenses, distinguished by the granularity at which they randomize, and thus the security they provide. In increasing order of randomization granularity they are: (i) ASLR [32], which provides library-level randomization; (ii) Selfrando [13], which provides function-level randomization; (iii) CCR [27], which provides function or basic-block-level randomization; and (iv) the Multicompiler [22], which provides instruction-level code randomization by inserting No-operations (NOPs) between instructions. Our evaluation shows that worst-case execution time spikes by a factor of up to 15x over average-case execution time, while the most performant randomizations are faster than baseline.

Our evaluation shows the promise and the pit-falls of using randomization-based defenses for RTS. Based on our evaluation, we present a research agenda for adapting MTDs to RTS, and RTS security more broadly. Furthermore, we discuss important research issues, including bringing worst case execution time in line with average case execution time without sacrificing entropy, and how to protect mixed-criticality workloads. This discussion sheds light on application domains where current MTDs are well suited, where more research is needed, and where perhaps other defensive approaches may be preferable.

2 BACKGROUND AND RELATED WORK

In this section, we provide background on RTS, before providing relevant background on the moving-target defenses that we considered in our evaluations.

2.1 Real-Time Systems Overview

There are many different classes of RTS, which include varying types of real-time requirements. Indeed, the term "real-time" is often overloaded and takes on different meanings in different contexts. For example, in many safety-critical application domains, "real-time" refers to a provable and hard requirement that certain computations will be performed within a predetermined time period, so as to ensure the physical system is actuated properly (*e.g.,* anti-lock brakes). On the other hand, in the field of computer graphics, "real-time" refers to the general capability of rending frames with little to no noticeable delays. This is sometimes referred to as "real-time" vs. "real-fast" computing. In between real-time and real-fast lies a spectrum of how strict the real-time requirements are, and how much rigor should/must be applied to ensure real-time performance. We describe such requirements here, as they poses different constraints on the defenses that are applicable in such domains.

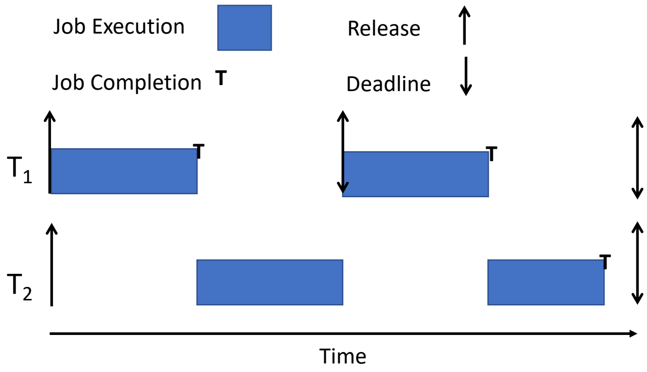


Figure 1: Example real-time schedule.

In *hard real-time systems (HRT)*, computations must complete before associated deadlines. Rigorous *schedulability analysis* is used to mathematically show that all deadlines will be satisfied. This schedulability analysis operates on an abstract *task model.* The most widely studied such real-time task model is the Liu and Layland model [28]. In this model, the workload is defined as a *task system* τ, composed of n *tasks*, $\tau = T_1, \ldots, T_n$. Each task T_i is defined as a (potentially infinite) sequence of *jobs*, or invocations of the task. Each job has an associated *WCET*, denoted e_i, which can be determined through static timing-analysis tools such as aiT [19]. Jobs are *released* or made available for execution periodically, with a *period*, denoted p_i. The period is often also the *deadline* by which each job must complete. This model is depicted in Fig. 1. Many more complex task models have been developed over the years, which allow more flexibility and/or fidelity in modeling more complex workloads, such as multi-threaded applications, and systems with dependencies, but such models are outside the scope of this paper.

Given a task model and scheduling algorithm, schedulability analysis can be derived to determine for a given task system whether or not all jobs will complete before their respective deadlines. For example, consider fixed-priority scheduling, in which tasks are prioritized by their task index, (*i.e.,* T_1 has the highest priority, and T_n has the lowest priority). The *response time* r_i of a job can be determined with the following recurrence:

$$r_i = e_i + \sum_{j=1}^{i-1} \lceil \frac{r_i}{p_i} \rceil e_j \qquad (1)$$

Systems in which some events may be delayed or dropped are often categorized as soft real-time (SRT) systems. There are several models for SRT performance guarantees. One such model is bounded deadline tardiness [14], in which no events are dropped but the maximum extent of a deadline miss can be analytically bounded. Furthermore, there are additional methods to tailor the scheduling policy around application-specific maximum tardiness bounds [17]. Other models for SRT scheduling include (m, k) systems, in which at least m of k successive computational jobs must complete before their deadlines. (This is also sometimes referred to as a weakly-hard real-time system.) Erickson [16] includes discussion of several other related SRT scheduling models. Several of these models have also been considered in the context of fault tolerance, and resilience to benign failures [21], but not malicious adversaries.

Mixed-criticality scheduling. Some systems are composed of a mix of applications with different types of timing requirements, and some tasks that are more safety critical than others. Extensive work has been conducted on such *mixed-criticality* systems [6]. Much of this work is motivated by the desire to consolidate applications onto a common computing platform to save on hardware costs. Furthermore, given the conservative nature of RTS design, in which systems are provisioned based on the worst case, rather than the average, in practice many HRT systems largely idle at runtime. By mixing lower-criticality tasks with less stringent real-time requirements onto a common computing platform with high-criticality tasks, the remaining computational horsepower can be harnessed to perform less critical work. This observation is made by Vestal [42], and has guided significant research in mixed-criticality scheduling, on how to ensure high-criticality tasks are guaranteed to complete on time, while still enabling idle time to be usefully re purposed for lower-criticality workloads.

2.2 Moving-Target Defense Overview

MTDs, also known as randomization-based defenses, are a broad category of defenses that share a common goal: preventing an adversary from achieving their objective by hiding / obfuscating key information from them. This class of defenses encompasses everything from randomizing network topologies [20] to instruction sets [3, 23, 31, 34, 41]. See [5] for a survey and taxonomy of such MTD classes. Other considerations include the frequency of randomization, resilience to information leakage attacks [44], and other implementation details, the discussion of which is beyond the scope of this paper. Here we focus on code-layout randomization defenses at different granularities, and introduce four representative defenses that we evaluate in the RTS context.

Code-layout randomization defenses attempt to prevent attackers from learning the address of any given snippet of code, *i.e.,* gadgets for code-reuse attacks [8, 35, 38]. Note that there can be architectural limitations on code motion, *e.g.,* x86 has `jump` instructions that can only do relative jumps of 256 bytes and `call` instructions with a relative range of 64KB. While it is possible to change the compiler or use a binary rewriter to ensure that `jump` and `call` instructions have arbitrary range, most defense chose to randomize within the limitations of the existing, emitted instructions. This is particularly limiting for fine-grained defenses, though they achieve significant entropy in practice.

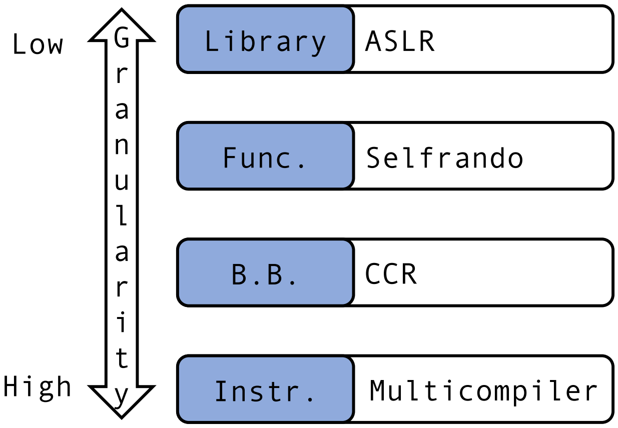


Figure 2: Granularity level of randomization based defenses. Note that libraries contain functions (func) contain basic blocks (BB) contain instructions (instr).

We consider defenses at four different levels of granularity: (i) library, (ii) function, (iii) basic block, and (iv) instruction. Address-Space Layout Randomization (ASLR) [32] is a library level randomization defense that is enabled by default on Linux and multiple other operating systems, and operates by placing shared libraries at random addresses, while maintaining all offsets within the library. Function-level randomization defenses such as Selfrando [13] change the addresses of functions, while maintaining the layout of basic blocks and instructions within them. Compiler-assisted Code Randomization (CCR) [27] does the same thing at the basic-block level. The Multicompiler [22] changes the addresses of instructions by inserting a random number of `no-ops` in between instructions.

Different randomization granularities carry different security vs. performance trade-offs. Finer-granularity randomization provides better security, but comes at a cost of potentially higher performance overhead. Other than the Multicompiler, which does change the sequence of instructions executed, the overhead of these randomization techniques all comes from micro-architectural effects of changing code layout, *e.g.,* cache and TLB misses, page faults, *etc.*. In particular, finer-grained randomizations introduce more changes in caches *etc.*, leading to less deterministic performance. These effects are highly non-deterministic and vary greatly between different randomizations of the same binary. In the next section, we experimentally evaluate these effects.

Here we give a brief overview of each evaluated defense and how it is implemented:

ASLR [32] is the coarsest granularity that we consider, changing the locations of entire shared libraries. It is enabled by default on most systems, and randomizes once at load time.

Selfrando [13] modifies the LLVM compiler and the system loader to enable function-level randomization by the loader. Consequently, all binaries are the same on disk, but have different memory layouts at runtime. Randomization occurs once, at load time.

CCR [27] modifies the LLVM compiler to preserve symbolization information. Binaries are then randomized by a separate shuffler, which uses the symbolization information to fix addresses in, *e.g.,* jump instructions after randomization. The shuffler can be executed at install time, leading to a randomized binary on disk, or load time resulting in a randomized memory image but non-randomized disk image. Additionally, the shuffler can be used multiple times on the same binary.

Multicompiler [22] is also an LLVM extension. In this case, the compiler inserts a random number of NOPs (zero or more) between instructions. As this changes the address of individual instructions, it is the finest grained defense we consider. Randomization occurs at compile time, not load time. Thus, despite it's finer granularity, it does not randomize as frequently as our other defenses.

We note that there are numerous other address-randomization defenses, including some that periodically re-randomize the address space at runtime [2, 4, 10, 45]. However, such defenses are outside of the scope of our evaluation. The above four defenses were chosen as representative defenses that operate at several granularities ranging from very coarse to very fine grained.

3 EXPERIMENTAL EVALUATION

We evaluate the performance characteristics of four randomization defenses, each of which randomizes at a different level of granularity, in order to determine their suitability for deployment in RTS. In RTS, the primary evaluation metric for software is not the average-case execution time (ACET), but rather the WCET. To date, all evaluations of randomization defenses report only ACET, which is the appropriate metric for general-purpose software. Unfortunately, testing for WCET is significantly more difficult than testing for ACET; system noise can be hard to differentiate from actual performance overhead imposed by the defense, and there is no definitive way to assert that the WCET for a specific process has actually been measured. Minimizing system noise is both vital to achieve consistent and accurate results and challenging to accomplish in practice. The general principle for reducing system noise is to isolate benchmarks as much as possible from the rest of the system. Additionally, longer running benchmarks help mitigate the issue of system noise, as the noise is proportionally lower the longer the runtime in our experience. As empirically evaluating the WCET is impossible, researchers should instead sample enough randomizations to have statistical confidence in the magnitude of the WCET. Using Extreme Value Theory (EVT) is a promising avenue [39] for gaining such statistical confidence.

To address these benchmarking concerns for RTS, we adopt the following methodology. We measure only the run-time performance of the benchmarks, excluding load-time overheads as RTS are long running and infrequently restarted. Performance is measured using the Linux time command. To control for system noise, we pinned the processes under test to a secondary core of the system. This helps alleviate cache pollution by other processes, interrupts, *etc.* from the OS. To ensure the process runs without preemption, we ran them at the highest priority using the Linux SCHED_FIFO scheduling policy. Our goal is to have the process run continuously on a dedicated core so that only its performance, and not its interaction with the system as a whole is measured. From the data collected, we found that system noise appeared to be present in a small percentage of tests, despite our best efforts.

In order to report an accurate distribution of runtimes for benchmarks, and thus be able to confidently determine the WCET, every binary under test was run a multitude of times; the number of executions was dependent on the MTD under test and attempted to balance total testing time with accurate data sets. We report the 95^{th} percentile execution time when analyzing WCET. This accounts for the difference in performance across different randomizations, while also controlling for outliers and remaining system noise. It is clear to us that reporting the worst observed execution time as representative of the WCET would distort results, whether the 95^{th} is the best proxy, or whether other statistical approximations such as EVT are better suited, is a question for follow on work.

For Selfrando, CCR and the Multicompiler we generated 10,000 unique randomizations of each benchmark. CCR supports multiple different randomization granularities; here we evaluate its basic-block-level randomization. We configured the Multicompiler to insert NOPs randomly with a 50% probability of insertion at each instruction. Additionally, we seeded the pseudo random number generator to allow for repeatable randomizations. For these defenses, each unique randomization was executed 100 times. We report the median across these 100 executions as the runtime for a given randomization when evaluating ACET, and the 95^{th} as the value when evaluating WCET. Consequently, the repeated runs per randomization help us control for system noise in our evaluation, while evaluating different randomizations lets us make general performance observations about a given randomization technique.

Our evaluation treats ASLR differently than the other defenses. ASLR operates at load time, and there is no way to repeatedly test the same randomization. Consequently, we ran each benchmark 1,000 times, and thus measured 1,000 different randomizations. Ideally, each randomization would be tested repeatedly to help control for system noise in measuring. However, as ASLR has negligible overhead, we do not consider this to be a serious issue.

We report results from the TACLeBench suite [18] - a series of benchmarks aimed to emulate processes found in RTS. Benchmarks that had execution times of less than 15 Linux time units were excluded as they were too short to accurately measure overheads and differentiate system noise. We performed all tests on an Intel® Xeon® CPU E5-2650 v2 operating at 2.60GHz with 125GB of RAM running Ubuntu Server 18.04. The baselines were generated for each benchmark by building them with the corresponding compiler, disabling only the defensive flags. The baselines were then executed 1000 times and the median and 95^{th}% percentile values were compared to the corresponding defended binaries. We show figures for one of the benchmarks, g723_enc, due to space constraints. However, our observations generalize across all benchmarks in TACLeBench.

Our evaluation of these runtime defenses highlights three key takeaways. First, despite our best efforts, system noise is still reflected in the measurements. Next, the degree to which randomization effects performance is correlated with the granularity of randomization. Finally, randomization effects WCET significantly more than ACET, highlighting the need for further research to adopt randomization defenses for RTS.

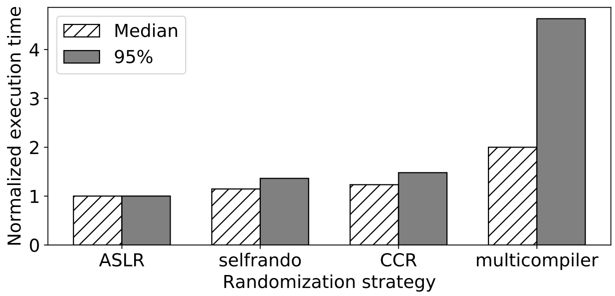


Figure 3: ACETs and WCETs of worst performing binary randomization for various MTDs normalized to the corresponding baseline values

Significant non-determinism exists despite noise reduction. Figs. 4 to 6 plot the various randomizations of the g723_enc benchmark against the baseline for the three defenses. To generate these plots, the randomized binaries were sorted by mean execution time, plotted in blue alongside the median and 95th percentile values, shown in red and green, respectively. Comparing the ACET values plotted on the left figures, and the WCET shown in the right figures, it can be seen that each randomization of the binary has non-negligible fluctuations across the 100 executions. To concretely measure the variability across executions of a given randomization, we computed the standard deviation, σ. For Selfrando, the lowest σ for the g723_enc benchmark was 4.39 unix `clock` ticks, with a maximum value of 20.15 ticks and an average value of 6.88 ticks. CCR had greater variability with a minimum σ of 0.50 ticks, a maximum of 31.90 ticks, and an average of 4.87 ticks. The Multicompiler had the greatest variability with min, max, and average σ values of 0.48, 78.00, and 12.70 ticks, respectively. With the majority of system noise accounted for, we suggest that, while some of these differences may be attributed to remaining noise, there are other system effects responsible. One such system effect may be virtual to physical mappings of code pages, managed by the OS memory-allocation system, which we discuss in the next section.

Non-determinism varies with randomization granularity. Firstly, randomization effects from ASLR have little to no impact to either the ACET or the WCET. We hypothesize that due to the coarse granularity of ASLR, randomizations have no impact on cache performance. Since all randomization still maps memory into the same offsets in a page, and therefor the same cache-lines, the hardware is affected in a deterministic manner. In this case, the buddy allocator used to map virtual to physical pages, and which returns a non-deterministic mapping per execution, has a greater impact on the cache variability than the randomization. On the other hand, the finer randomization defenses we evaluated showed clear trends indicating their lack of suitability for RTS applications. Figs. 4a, 5a and 6a highlight the ACET for a given randomization and reflect that certain binary randomizations have significantly worse performance than others, and confirm that the mean value is not being driven higher by a few extreme values, but that the overall average is rising as well. Figs. 4b, 5b and 6b plot the WCET values, reflecting the 95th percentile execution times for the associated binary randomization. Fig. 3 plots the ACET and WCET for each of the MTDs averaged across all tested benchmarks. For the defenses which had multiple randomizations, we selected the ACET and WCET with the greatest overheads. In addition, these values are normalized to the corresponding baseline value, *i.e.,* the WCET for Selfrando is normalized to the WCET for the Selfrando baseline. This plot reflects an important takeaway; the WCET for certain binaries can be significantly slower than the ACET for the baseline, even if the ACET for that randomization only has minor overheads.

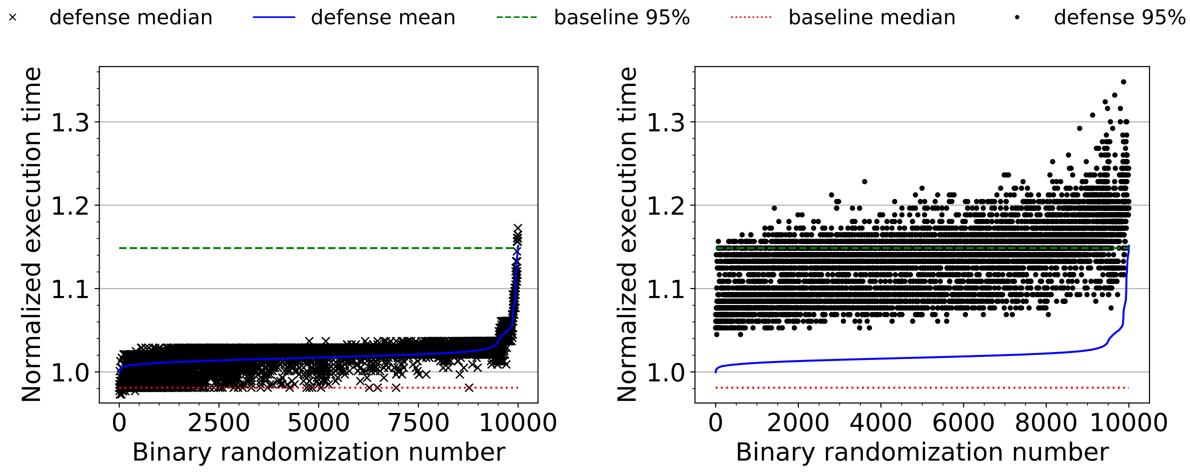
Randomization has greater effects on WCET than ACET. Fig. 3 not only shows how the different granularities of randomization affect execution time, but also it highlights how WCETs grow at a much greater rate than ACET. Since this figure also plots the ACET for the worst performing binary randomization when referencing Selfrando, CCR, and the Multicompiler, it can also be seen that the ACET across all binary randomizations would be lower. This is further reflected in Figs. 4 and 5, where the majority of binaries have ACETs at least 10% lower than the ACET of the worst performing randomization. Not only does randomization affect the WCET, it also affects the best-case execution time, sometimes in a positive manner. For each MTD with granularity finer than ASLR, there are binary variants that have ACET and WCET better than the baseline values.

4 DISCUSSION

Our evaluation of MTD for RTS applications reveals several interesting research challenges and opportunities. For example, while our evaluation focuses on the WCET, we observe that the best case can be faster than the baseline, pointing towards a promising direction for future work on performance optimization. More importantly for security, new MTD techniques are needed for RTS, and we highlight both a straw man possibility and give insight into how principled new defenses can be built. Finally, we end this section with thoughts on how to secure mixed criticality execution environments.

Measurement Noise. Despite our best efforts, there is still non-determinism in timing across different runs. One culprit of this is the OS memory allocator, which assigns different pages to an application every time it is executed. Consequently, the memory-to-cache mapping may change every execution, which can affect execution times. We note, however, that such noise is not an artifact of our measurement methodology, but rather a byproduct of the underlying operating system not being designed for predictable real-time performance. Other sources of measurement noise include interrupts, which in RTS are often independently measured and accounted for in schedulability analysis.

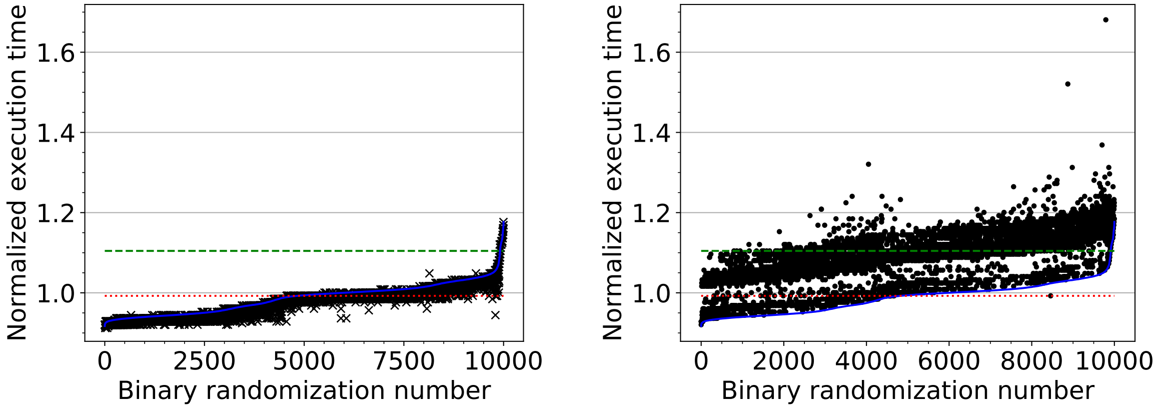
Randomization as an optimization. One under-explored aspect of randomization is that some randomizations actually *improve* application performance. Under CCR, as seen in Fig. 5a, approximately half the randomizations have ACETs that are faster than the ACET of the baseline. This is in contrast to Selfrando, as seen in Fig. 4a, where only approximately 200 randomizations are faster than the ACET of the baseline. CCR randomizes basic blocks, whereas Selfrando randomizes functions. The Multicompiler results, see Fig. 6a,



(a) ACET values

(b) WCET values

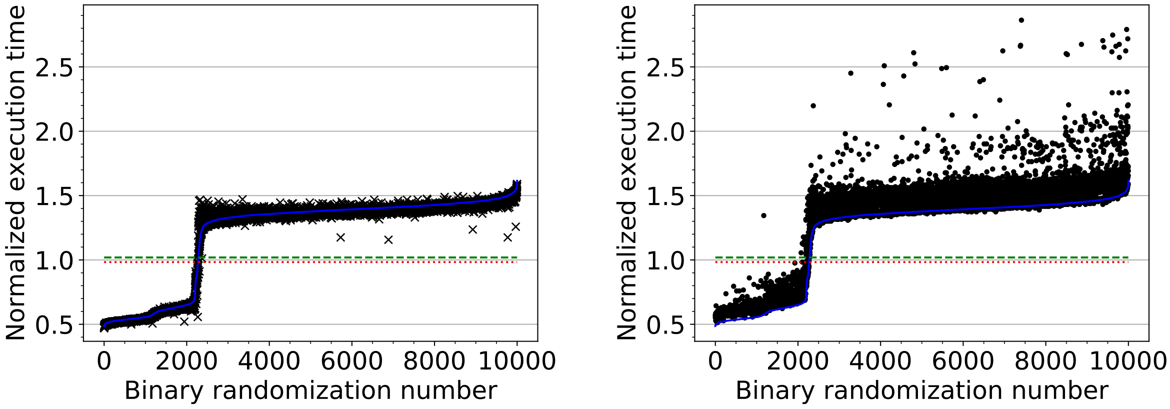
Figure 4: Execution times for Selfrando, sorted by mean value, plotted against the baseline median and 95$^{\text{th}}$%



(a) ACET values

(b) WCET values

Figure 5: Execution times for CCR, sorted by mean value, plotted against the baseline median and 95$^{\text{th}}$%



(a) ACET values

(b) WCET values

Figure 6: Execution times for the Multicompiler, sorted by mean value, plotted against the baseline median and 95$^{\text{th}}$%

show that approximately 20% of the randomizations have a faster ACET than the baseline. As the Multicompiler only inserts no-ops this strongly indicates that changing the alignment of individual instructions can have significant performance impacts, which is also supported by basic-block randomizations having a greater positive performance impact than function-level randomizations. The extent to which these results are input dependent, or whether changing code layout can be a general-purpose optimization remain open questions. Nonetheless, profile guided changes to code layout is a promising direction for future research.

Randomization deficiencies for RTS. Real-time applications are often subject to schedulability analysis, which must make worst-case assumptions about performance. Consequently, code with consistent execution times are preferred, and allows for tighter execution bounds. Based on the results of our experiments it is clear that current randomization-based defenses introduce significant non-determinism, which would be costly in hard real-time applications. Analyzing WCET across all randomizations yields performance bounds that are much higher than they would be without the defense. Improving WCETs should be pursued via two avenues. Firstly, reduction of non-determinism within a given binary randomization without introducing new security vulnerabilities improves the accuracy of modeling. Perhaps more important to improving the usability of MTDs for RTS, however, is reducing the variability of execution times across randomizations. This is imperative if MTDs are to be applied to RTS, as currently, not only does the WCET need to be identified for a specific binary, but the worst performing binary must also be identified. It is also relevant to note that the worst-performing binary, when analyzed by average performance, may not present the overall WCET. This is exemplified in Fig. 6b, where the worst overall execution time is around the 75th percentile of randomized binaries when sorted by mean execution time.

Solution 1: Limit randomization space. One option to address the higher WCET of randomization based defenses is to profile randomizations, and discard those that have unacceptable WCET. Doing so, however, reduces the entropy of the system; such a trade-off would need to be studied to see if the remaining entropy introduces satisfactory amounts of security to be considered useful. Some defenses, like the Multicompiler, include adjustable parameters to modify the randomization outputs. Reducing the allowable modifications to a binary could also reduce the divergence of performance, but will also reduce entropy of the system. In general, we consider such an approach to be drastic, likely to be unsound, and at best a temporary measure until more principled solutions to make randomization for RTS acceptable can be deployed.

Solution 2: Adapt randomization for RTS. Novel randomization defenses for RTS that reduce the spikes in WCET may be possible. To see why, observe that virtual addresses are what matter to attackers. To use gadgets in code-reuse attacks such as Return Oriented Programming (ROP) [38], attackers must know their virtual addresses. MTD are premised on obscuring/hiding those addresses. Attackers have in turn responded with information-leakage attacks [36, 44] that target the disclosure of virtual addresses, and allow them to construct malicious payloads. Note, however, that this security war is entirely over *virtual* addresses.

In contrast to the security concerns over *virtual* addresses, the performance issues of randomization-based defenses stem from cache behavior, which is dictated by *physical* addresses. The physical layout of the executable code within the binary is the largest contributor the cache-layout, and therefore the performance, as shown by the divergence in execution times between different randomizations, regardless of MTD used.

Consequently, *virtual* addresses are critical for security, and *physical* addresses are key to deterministic performance. There is no inherent reason randomizing virtual addresses *must* change physical addresses. In modern operating systems, the mapping between virtual and physical addresses is handled by the operating system. The net effect of this is that in two unique runs of the same application, the same virtual address will likely be mapped to two different physical addresses, while two unique virtual addresses could map to the same physical address between runs. We believe developing technologies that enable RTS to influence virtual to physical page mapping, as opposed to legacy and general-purpose OS designs where the mapping is entirely dictated by the OS, will enable MTD defenses to be deployed in the RTS context.

Securing soft real-time systems. Many RTS control or monitor cyber-physical systems, which have physical inertia. In some such systems, missing a deadline or dropping a job may be acceptable if it is a sufficiently rare event, and the system may continue to function properly given its inertia [29]. Analysis-driven approaches to system design and provisioning, even in SRT systems, offer higher mission assurance and improved performance [24]. MTDs are promising defensive options for such systems, especially if the distribution of execution times under different diversifications are well understood. Such analytical approaches enable the system designer to incorporate stronger, higher-overhead defenses while maintaining sufficiently high assurance SRT performance.

There have also been several MTDs based on *re-randomization*, which are potentially more relevant in the SRT application space. Examples of such techniques include timely address space randomization (TASR) [4], remix [10], and shuffler [45]. More recently, the YOLO project [2] has applied re-randomization to frequently reset and diversify control applications, such as in an automotive Engine Control Unit (ECU) and quadcopter flight controller.

Securing mixed-criticality applications. The design objective of mixed-criticality systems is to ensure that even in a temporal overrun that the most highly critical workloads will still continue to execute in a timely fashion. To support this, lower-criticality workloads are temporarily dropped to maintain safe execution. Thus, these lower-criticality workloads also may have more relaxed timing requirements. Such lower-criticality tasks are especially well suited to MTDs approaches, as on average these defenses exhibit very low average-case runtime overhead, and in the rare case that higher overheads are experienced, the mixed-criticality design will ensure that the high-criticality workloads still execute properly.

Software certification Some real-time application domains, such as avionics, are subject to *certification* requirements. Safety-critical software in commercial aircraft, for example, has to be certified by the FAA, and this certification process is often more costly than the software development costs themselves. Binary-diversification approaches may be problematic in this domain, as the binary executing on each system is different. Thus, one would either have to certify each individual binary independently (*i.e.,* certify each individual aircraft, instead of the entire product line), which is prohibitively costly, or develop certification-worthy analysis that guarantees that each diversified binary will have the same, or similar and bounded performance. This is an open area of research.

5 CONCLUSION

MTDs have been developed to provide security benefits, often with very minimal or lightweight overhead costs. At first glance, such low-overhead defenses may seem well suited to SWaP-constrained RTS. However, we presented experimental evaluations of the worst-case performance of four different address-space randomization defenses of varying granularities, and found that the more secure, finer-granularity defenses exhibit significant worst-case performance overheads, which are comparatively much higher than the average-case performance overheads that are relevant to traditional computing systems.

Based on this observations, we have discussed several considerations and future research directions to make MTDs more amenable to RTS. While current MTD may be acceptable for soft real-time and/or mixed-criticality applications where some temporal overruns may be acceptable, more research is necessary to develop such defenses for pure hard real-time applications. We have discussed promising avenues to address this challenge, in particular ways to decouple the non-deterministic performance overheads from the non-determinism inherent in the defense to prevent attacks.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Miguel A. Arroyo, M. Tarek Ibn Ziad, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. 2019. YOLO: Frequently resetting cyber-physical systems for security. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, Michael C. Dudzik and Jennifer C. Ricklin (Eds.), Vol. 11009. International Society for Optics and Photonics, SPIE, 166 – 183. https://doi.org/10.1117/12.2518909

[3] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 281–289.

[4] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *ACM Conference on Computer and Communications Security (CCS)*.

[5] Richard W. Skowyra D. Bigelow Jason N. Martin James W. Landry Bryan C. Ward, Steven R. Gomez and Hamed Okhravi. 2018. *Survey of Cyber Moving Targets Second Edition.* Technical Report 1228. MIT Lincoln Laboratory.

[6] A. Burns and R. Davis. 2018. Mixed Criticality Systems – A Review. (2018). https://www-users.cs.york.ac.uk/burns/review.pdf

[7] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1 (April 2017).

[8] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Sec)*.

[9] Defense Use Case. 2016. Analysis of the cyber attack on the Ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)* 388 (2016).

[10] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 50–61.

[11] A. Clements, N. Almakhdhub, S. Bagchi, and M. Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *Proceedings of USENIX Security*.

[12] A. Clements, N. Almakhdhub, K. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. 2017. Protecting Bare-Metal embedded systems with privilege overlays. In *S&P '17*.

[13] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (2016), 454–469.

[14] U. Devi and J. Anderson. 2008. Tardiness Bounds under Global EDF scheduling on a multiprocessor. *Real-Time Systems* (2008).

[15] Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. 2018. TRITON: The first ICS cyber attack on safety instrument systems. In *Proc. Black Hat USA*. 1–26.

[16] J. Erickson. 2014. *Managing Tardiness Bounds and Overload in Soft Real-Time Systems.* Ph.D. Dissertation. University of North Carolina at Chapel Hill.

[17] J. Erickson, J. Anderson, and B. Ward. 2014. Fair Lateness Scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems, special issue on selected papers from the 24th Euromicro Conference on Real-Time Systems* (2014).

[18] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*.

[19] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.

[20] Steven R Gomez, Samuel Jero, Richard Skowyra, Jason Martin, Patrick Sullivan, David Bigelow, Zachary Ellenbogen, Bryan C Ward, Hamed Okhravi, and James W Landry. 2019. Controller-Oblivious Dynamic Access Control in Software-Defined Networks. In *Proc. of DSN*.

[21] A. Gujarati, M. Nasri, R. Mujumdar, and B. Brandenburg. 2019. From Iteration to System Failure: Characterizing the FITness of Periodic Weakly-Hard Systems. In *ECRTS '19*.

[22] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-generated software diversity. In *Moving Target Defense*. Springer, 77–98.

[23] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS '03)*. ACM, New York, NY, USA, 272–280.

[24] Christopher J Kenna, Jonathan L Herman, Björn B Brandenburg, Alex F Mills, and James H Anderson. 2011. Soft real-time on multiprocessors: Are analysis-based schedulers really worth it?. In *Proceedings of the 32nd Real-Time Systems Symposium*. IEEE, 93–103.

[25] Michael Kirsch. 2011. National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation. (2011). https://www.nhtsa.gov/staticfiles/nvs/pdf/NASA-UA_report.pdf

[26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[27] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 461–477.

[28] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61.

[29] J. Sukarno Mertoguno, Ryan M. Craven, Matthew S. Mickelson, and David P. Koller. 2019. A physics-based strategy for cyber resilience of CPS. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, Michael C. Dudzik and Jennifer C. Ricklin (Eds.), Vol. 11009. International Society for Optics and Photonics, SPIE, 79 – 90. https://doi.org/10.1117/12.2517604

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* (2009).

[31] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 981–992.

[32] PaX. 2003. PaX Address Space Layout Randomization. (2003). ttp://pax.grsecurity.net/docs/aslr.txt

[33] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 271–282.

[34] Georgios Portokalidis and Angelos D. Keromytis. 2010. Fast and Practical Instruction-set Randomization for Commodity Systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. ACM, New York, NY, USA, 41–48.

[35] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (S&P)*.

[36] Jeff Seibert, Hamed Okhravi, and Eric Soderstrom. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*.

[37] Fermin J. Serna. 2012. cve-2012-0769, the case of the perfect info leak. (2012). http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf

[38] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*.

[39] Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva de Oliveira. 2017. On using GEV or gumbel models when applying EVT for probabilistic WCET estimation. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 220–230.

[40] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.

[41] Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? the effectiveness of instruction set randomization. In *14th USENIX Security Symposium*, Vol. 6.

[42] Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE international real-time systems symposium*. 239–243.

[43] R. Walls, N. Brown, T. Le Baron, C. Shue, H. Okhravi, and B. Ward. 2019. Control-flow integrity for real-time embedded systems. In *ECRTS '19*.

[44] Bryan C Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. 2019. The Leakage-Resilience Dilemma. In *European Symposium on Research in Computer Security*. Springer, 87–106.

[45] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. 367–382.