Rave: A Modular and Extensible Framework for Program State Re-Randomization

Christopher Blackburn
Virginia Tech
Blacksburg, Virginia, USA
krizboy@vt.edu

Xiaoguang Wang
Virginia Tech
Blacksburg, Virginia, USA
xiaoguang@vt.edu

Binoy Ravindran
Virginia Tech
Blacksburg, Virginia, USA
binoy@vt.edu

ABSTRACT

Dynamic software diversification is an effective way to boost software security. Existing diversification-based approaches often target a single node environment and leverage in-process agents to diversify code and data, resulting in an unnecessary attack surface on a fixed software/hardware stack. This paper presents Rave, a practical system designed to enable *out-of-bound program state shuffling on a moving target environment*, avoiding any sensitive agent code invoked within the running target. Rave relies on a user-space page fault handling mechanism introduced in the latest Linux kernel and seamlessly integrates with CRIU [10], the battle-tested process migration tool for Linux.

Rave consists of two components: `librave`, a library for static binary analysis and instrumentation, and CRIU-Rave, a runtime that dynamically updates program execution states (e.g., internal stack data layout and the machine node the program runs on). We built a prototype of Rave and evaluated it with four real-world server applications and 13 applications from the SPEC CPU 2017 and the SNU C version of NAS Parallel Benchmarks (NPB) benchmark suites. We demonstrated that Rave can continuously re-randomize the program state (e.g., internal stack layout, instruction sequences, and machine node to run on). The evaluation shows that Rave increases the internal program state entropy with an additional ≈*200 ms* time overhead for each re-randomization epoch on average.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software and application security**;

KEYWORDS

Code and Stack Randomization, Moving Target Defense, Software Security

ACM Reference Format:
Christopher Blackburn, Xiaoguang Wang, and Binoy Ravindran. 2022. Rave: A Modular and Extensible Framework for Program State Re-Randomization. In *Proceedings of the 9th ACM Workshop on Moving Target Defense (MTD '22), November 7, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3560828.3564008

MTD '22, November 7, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9878-7/22/11.
https://doi.org/10.1145/3560828.3564008

1 INTRODUCTION

Software diversification techniques have been proposed to break static and predictable program layouts and states while maintaining correct functionality [6, 11, 12, 23, 33, 36]. Among these techniques, dynamic software diversification becomes more promising, as it demonstrates the capability of defeating advanced exploits [13, 31]. Existing approaches dynamically re-randomize a target's application memory layout [6, 23, 24, 36], update various configurations [18] or change the execution environment at runtime [11, 17, 33] to make the target less predictable. For example, there are several code re-randomization approaches that leverage an in-process randomization agent to dynamically update code and data layouts [6, 11, 33, 36]. However, solutions that use a randomization agent introduce an additional attack surface (the agent) which often needs extra protection [36].

Moving target defense (MTD) is another concept of dynamic software diversification [8, 18]. It aims to break static attack surfaces by dynamically changing the software execution environment, such as system configurations, software stacks, and network addresses [18], to name a few. However, existing software-oriented MTD approaches *treat the target program as a whole entity* without diversifying the program's internal state [8]. These approaches cannot prevent attacks that exploit the internal code vulnerability. Especially with the emergence of advanced code-reuse attacks like position-independent return-oriented programming (PIROP) [13] or non-control data attacks like data-oriented programming (DOP) [15, 16]; defenses against these are growing thin. These new variants of code-reuse attacks make existing mitigation mechanisms less effective.

In this work, we present Rave - a modular and extensible framework for re-randomizing a live process's code and memory space in a distributed moving target defense environment. Specifically, Rave can migrate a live process (or a container in our future implementation) to machine nodes with different hardware settings or software stacks, obfuscating the attacker's knowledge of the target's location and runtime software/hardware stack. Meanwhile, Rave can also update the program's internal state, such as the stack layout or code instruction sequences, at a minimal cost. More importantly, Rave allows out-of-bound program state transformation and execution relocation without invoking any program agent within the target.

Rave leverages the user-space page fault handling mechanism (i.e., `userfaultfd` [29] in Linux) to reload the randomized code and data pages and seamlessly integrates with the transformation logic provided by CRIU [10] to migrate processes among different machine nodes. Rave consists of two components: `librave`, a library responsible for static binary analysis and instrumentation, and CRIU-Rave, an extended version of CRIU [10], which is a battle-tested process migration tool available for Linux. We also built a prototype of Rave and evaluated it using four server applications and 13 applications from the SPEC CPU 2017 and the SNU C version of NAS Parallel Benchmarks (NPB) benchmark suites. We demonstrate that Rave can relocate the program execution across the machine boundary and simultaneously re-randomize the internal program state. The evaluation results show that Rave increases the internal program state entropy with an additional ≈*200 ms* time overhead added to the live migration. Overall, we made the following contributions:

- We propose a modular and extensible framework for code and memory randomization in a distributed moving target defense environment.
- We present the design and one type of implementation of Rave for dynamic program stack randomization during the live migration for high program entropy; Rave is transparent and out-of-bound to the target program.
- We report evaluation results showing that Rave increases the entropy of internal program states during the process migration at near-zero cost.

The rest of this paper is organized as follows: Section 2 provides background information of dynamic software protection and the threat model. We then describe the design and implementation of Rave in Section 3. The evaluation is presented in Section 4. Afterwards, we discuss the limitations and potential future works in Section 5 and summarize the related works in Section 6. Finally, we conclude the paper in Section 7.

2 BACKGROUND

This section briefly introduces the background on dynamic software diversification and CRIU and then defines the threat model and assumptions.

2.1 Dynamic software diversification

Moving target defense (MTD) is a concept for dynamic software system protection [8]. Many existing software defense methods often employ static and predictable defense strategies, such as static checks against the integrity of control flow data vulnerable to attacks (CFI [1], shadow stack [7]), authentication of the identity of users and their actions [30] or formal verification of program correctness against its original design [22]. Although practical, these approaches give attackers time to analyze the target program and eventually find the vulnerability of the defense method [21, 31]. Dynamic software defense aims to break the attacker's advantage by making their targets less predictable.

There are several ways to make the target program execution unpredictable. One method is to dynamically change the target system's configurations. Previous research efforts leveraged dynamic network configuration or hardware settings to shift the attack surface [8, 17]. Such systems dynamically change the routing table by assigning each virtual IP to different real IPs of the hosts, which hides the actual server configurations [17]. However, even if the hosts' IP addresses are frequently changed, the internal state of the server application does not change. A smart attacker can prepare the payload through advanced techniques such as PIROP even if the program is address-randomized [13].

Load-time address space layout randomization (ASLR) [12, 25] is another form of dynamic software diversification. It generates the position-independent executable (PIE) and loads the executable into a randomized location. The original ASLR does not bring any runtime overhead but it may suffer from heap spraying [14] or just-in-time code reuse attacks [31]. Advanced ASLR techniques, such as runtime code re-randomization and fine-grained memory randomization, aim to either relocate code block locations during process execution [6, 36] or randomize the code at the finer-grained basic block level [20, 34]. They focus on building particular randomization mechanisms and often embed a randomizing agent within the target application. This may create a larger attack surface [36] and introduce difficulties in decoupling the randomization agent from the target program.

Unlike existing works, we aim to decouple the randomization tooling from the target program. The target program is first loaded with ASLR; Rave shuffles the internal stack layout during each cross-node migration.

2.2 Live process migration and CRIU

Process (container) live migration moves application instances to different machines without disconnecting the clients. It has been primarily used for server maintenance (e.g., OS kernel update) and load balancing. There are several projects that implement live process migration [3, 10, 32]. For example, Checkpoint/Restore in Userspace (CRIU) is a recent project that supports userspace process (container) live migration [10]. When users want to migrate a process, they can invoke CRIU to dump the process' state into a set of image files, then, from those files, restore the process.

At the beginning of the *dumping process*, CRIU attaches to a process and all its children using ptrace [35]. To stay as true to the current state of the process, CRIU does not use ptrace to signal the process to stop. Instead, it uses an in-kernel facility to freeze the process before collecting and saving to disk information about the running process(es). Information about the process is mostly gathered from Linux's `/proc` file system. The process image files can be moved to the target machine node for restoration. On *restoring a process*, CRIU will analyze the dumped process image then morph itself into the target to be restored. For every restoree, CRIU will fork itself then continue per-process restoration. Files are reopened, memory is remapped and filled with dumped data, thread's executions are resumed, and the process gets restored. This can happen on the *same or different machine*, but there are some restrictions: the filesystems must match (or else things like open files cannot be restored). Any kernel features that exist in the source node, must also be available on the target node.

CRIU also has additional methods for restoring a process. In some cases, like live migration, it may be undesirable to copy all the dumped process data to another machine before restoring that process (since this data could be very large). CRIU provides a way to *lazily-load memory pages*. Processes are restored like normal for the most part, but instead of reading and copying all dumped memory from the files into the restorees' memory, some pages are marked as lazy loadable and registered with a `userfaultfd` file descriptor [29]. Userfaultfd is a Linux kernel facility that allows users to handle page faults in user space. Basically, this allows us to register regions of memory with a file descriptor, then when a page fault happens (e.g. when some memory has not been loaded into the current address space), we are notified of it and are able to serve the fault. This allows the lazy-pages process to provide the restoree with data only when it needs it (both locally or over the network). In this work, we leverage lazy-pages restoration to perform randomization outside of the target process' context, making the randomization tooling transparent to the target.

2.3 Threat Model

We assume the attacker has remote access to the target process through a standard I/O interface; specifically, a socket connection. The attacker may have access to the target program binaries. However, we assume the distributed MTD machine nodes are physically hidden from external attackers. Similar to most existing dynamic software diversification techniques, Rave does not remove the vulnerabilities but relies on the uncertain program states and execution environment to prevent attackers from successfully (or reliably) launching the payload. We assume the implementation of CRIU and the binary disassembler are correct and sound; we also assume a strong trusted computing base (TCB), including the OS kernel and the ELF loader. Side-channel attacks and kernel vulnerability exploits and mitigation are out of the scope of this paper.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

Rave is designed as a modular and extensible framework for code and memory randomization in an MTD environment (Figure 1). It timely re-randomizes the program's internal state as well as the execution environment. Unlike existing code re-randomization or MTD approaches [6, 11, 33, 36], Rave is fully isolated from the target program's address space; thus an attacker cannot compromise Rave's defense logic. Rave also allows the target process to be migrated in a distributed environment to dynamically change hardware and software stack settings.

Rave is split up into two main components: a library for defining randomization policies (`librave`), and an executing engine to update the randomized code at the runtime (CRIU-Rave). In Rave design, we extended CRIU as the runtime to trigger the randomization during the process migration with near-zero cost. Both Rave's components are written entirely in user space. To defend against memory corruption-based attack payloads, Rave rewrites code and live stack spaces to confuse attackers who are trying to take advantage of stack predictability. Meanwhile, Rave leverages the rich environment of CRIU's process migration, which allows it to take advantage of several existing features while bolstering features like live migration through the additional security techniques it provides. Thus, even if an attacker accidentally locates the program's execution environment, the internal state shuffling can prevent the already prepared payload from succeeding. In the remainder of this section, we will describe Rave in detail.

3.2 `librave`

One draw back of previous runtime code-randomization works is that the driver code (re-randomizer) is often either a part of the target binary [6, 36] or is tightly coupled to the randomization code [23, 24, 33]. That is, it would be a non-trivial engineering task to disassociate randomization techniques in previous works from the applications and adopt those techniques to a new execution environment.

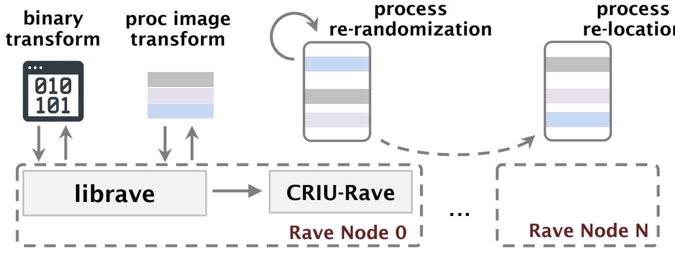

Figure 1: Overview of Rave on a multi-node MTD environment. The target program's internal state is (self) re-randomized or shuffled during the live process migration. CRIU-Rave and `librave` are isolated from the target program's address space.

In order to avoid this type of behavior, `librave` was built as a pluggable library so that no one program was tied to its capabilities. This library aims to provide the basic capabilities that code transformations can rely on. This includes abstractions for reading and navigating binary files (in this case ELF files), abstractions for reading and using binary metadata like DWARF [9] debug information, binary rewriting (disassembly and reassembly), and methods for maintaining records of code transformation (so that live processes can be adjusted to match re-written code). `librave` can be logically broken into two execution phases: an analysis phase, and a transformation phase.

The *analysis phase* of `librave` consists of any setup required to transform code. This includes loading ELF binaries and parsing metadata, creating internal representations of transformable functions, and setting up pages for serving transformed code. The first step in Rave analysis is to prepare the ELF binary. When given an executable ELF, `librave` parses the program headers and section headers to find the `.text` section (the section containing the user's compiled code) as well as the segment indicating where the code is loaded. This segment is artificially loaded into `librave` 's address space for further analysis and transformation. `librave` maintains this memory region for code transformation so that it can be readily served to the target process via page faults or written back to a randomized executable. Either way, Rave separates the code transformation logic from the target program itself.

Next, `librave` parses any metadata available to it through *a metadata abstraction class.* In our prototype, we leverage the DWARF debug information as the backing structure for this metadata class. This class exposes an interface through which we can interact with information about the code we just mapped. For example, the DWARF debug information can provide function offset and the location of their local variables to `librave` [9]. After parsing this information, `librave` iterates through and processes each function defined by the included DWARF metadata. Using the metadata and mapped code region, Rave disassembles each function to discover and determine which functions are randomizable. Our current prototype showcases callee-preservation code permutation (*i.e.,* stack slots containing the contents of function-owned registers). Functions with more than one permutable stack slot are called *randomizable functions*. By permuting the locations of these stack slots, we can make it harder for attackers to guess where certain target slots are located.

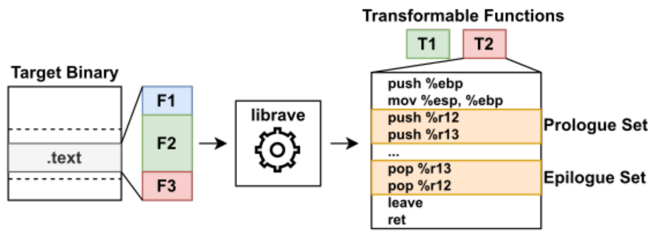

Figure 2: RAVE analysis phase visualized. The binary is loaded into `librave` 's address space and analyzed. `librave` searches for randomizable functions and records metadata about those functions (e.g. locations of prologues and epilogues).

`librave` also exposes a way to artificially relocate the randomized code. For position-independent executables (PIEs), the executable can be loaded into arbitrary addresses. CRIU-RAVE intercepts the base address and adjusts the global offset table (GOT) in the reserved memory region. Likewise, the executable segment may not be located at the address given in the ELF program header. Thus, for live programs, RAVE is able to consume a new base address for these sections. This is mirrored in stack rewriting, since the base address and offsets of the stack space could vary.

Once `librave` has finished analysis, RAVE triggers the *code transformation*. This transformation is applied to each function captured by the analysis phase, then re-encoded back into a local buffer (*i.e.,* code cache). Currently, `librave` only shuffles stack slots (*i.e.,* the stack variable/object) in a function's prologue and epilogue which includes the corresponding instructions, such as `push`/`pop` instructions. Other instructions referencing the stack frame pointer (*i.e.,* `%rbp`), outside of those in the prologue and epilogue, could also be shuffled by further analyzing the stack slot types: if a pointer stored on the stack points to another stack object, we would need to update the pointer value after relocating the stack object. Also consider cases where pointers on the heap point to stack slots that could be relocated. In these instances, those pointer values stored on the heap would likewise need adjustment.

Once a new order of stack slots has been determined, `librave` re-encodes preservation instructions for each function and delivers the code cache to the RAVE driver. The driver code is responsible for taking the modified code and serving it. This could mean saving it to a new, randomized binary, or dynamically serving code pages through page faults for code re-randomization (Section 3.3). Although our current prototype only supports shuffling a subset of stack-accessing instructions, we anticipate that other randomization policies (*e.g.,* fine-grained basic block shuffling [4, 34]) could also be supported with minimal changes.

`librave` also supports data-space memory transformation. In the case of stack slot shuffling, `librave` also transforms the stack according to the new code layout. Specifically, RAVE can process the process snapshot, locate the stack memory pages and unwind a live application stack (using the frame pointer, we can traverse through each stack frame). Each stack frame is matched to its respective function (previously recorded in the analysis phase) by either the instruction pointer in the case of the outermost frame or the return address saved on the stack. After that, the driver program is responsible for providing `librave` with the stack space and relevant information.

3.3 CRIU-RAVE

`librave` is a library, and thus can be driven by a third party. CRIU is one such party which enables process migration in Linux. CRIU-RAVE is a fork of CRIU built to link with and drive `librave`. Upon the process restoration, CRIU-RAVE invokes `librave` to re-randomize a process by rewriting its code and stack. CRIU-RAVE serves the re-randomized pages (through `userfaultfd`) for page faults during the process restoration. Figure 3 illustrates this process. The target process is restored separately from where `librave` is invoked to randomize its layout. The stack and code pages are served from the lazy-pages co-process on demand through page faults. By the time the restoree is ready to resume execution, the randomized code pages and stack are ready to be delivered once accessed.

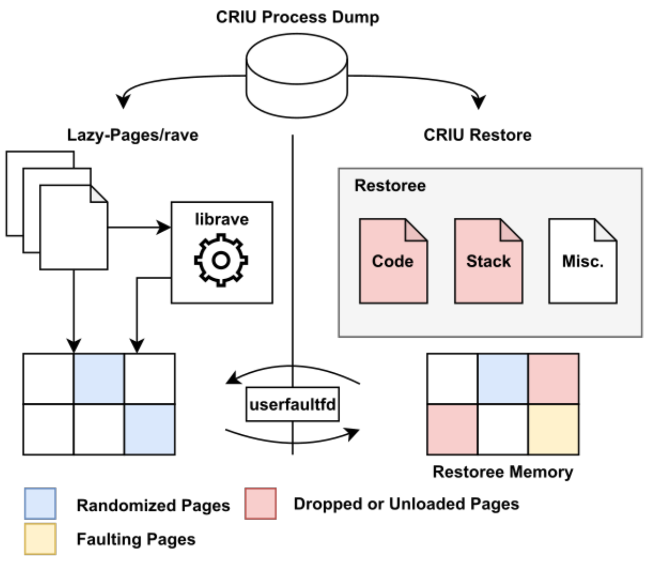

Figure 3: Overview of CRIU-rave Runtime. CRIU-rave runs the restore process and the lazy-pages process in parallel. Relevant pages are dropped or left unloaded by the restoree. `librave` intercepts code and stack pages in the lazy-pages process to serve them out via `userfaultfd` whenever the restoree triggers a page fault by touching an unmapped page.

To serve code and stack pages from the lazy-pages daemon, CRIU uses Linux's `userfaultfd` facility. This interface allows us to handle page faults from user space. During CRIU's restore process, we can register memory regions in the restoree's address space with a `userfaultfd` file descriptor so that any page fault within the registered memory region will raise an event to that file descriptor. Note that this interface only works on anonymous memory mappings. Trying to register a file backed mapping, like the executable region of the target binary, will fail. To work around this, we replace the file-backed mapping of the executable segment with an anonymous mapping (matching all original permissions of course). This is the only artifact in the target application that might suggest our tooling is active. Once this region is registered with `userfaultfd`, we make a `madvise()` system call to with the `DONT_NEED` flag, effectively telling the kernel that these pages can be dropped. Thus, the next time these pages are accessed, it will trigger a page fault.

The stack region needs no special treatment as it is already marked for lazy loading. Once both the code and the stack regions are prepared in the restoree (and any other lazy-loadable pages), CRIU-Rave sends the `userfaultfd` file descriptor to the lazy-pages process. Normally, under this facility, we could only serve pages from within the same process. However, by using a Unix socket to transfer the file descriptor to a listening process, we can continue to serve page faults in user space from outside the target process.

CRIU-Rave lazy-pages will initialize itself in preparation to receive the `userfaultfd`. It sets up a list of lazy-process structures which carry any relevant information necessary to serve page faults, including structures that are prepared to read memory from the dumped process images. During this initialization, we can concurrently prowl the dumped images (on a per-process basis) to find the on-disk location of the executable file via `/proc/<pid>/exe`, as well as the virtual memory address of the executable segment. At this time, we can also read the saved stack memory and CPU register snapshot, which will be used to rewrite the stack.

The binary file is not saved in the dumped memory (it is reopened on CRIU restore). So, we end up passing the location of this file to `librave`, triggering the analysis and transformation of the code. Once the code is transformed, we can send it the stack space we read from the dumped memory for rewriting. Once these components are available, the co-process must wait for a page fault. In unmodified CRIU, when a page fault occurs, it captures that event and serves memory directly from the process images. However, in CRIU-Rave, we intercept this process and check whether the page fault happened in a registered code or stack region. If this was the case, `librave` exposes the modified code or stack to CRIU-Rave so that it can serve the page fault, thus injecting the randomized memory into the target application.

4 EVALUATION

In this section, we evaluate both `librave` and CRIU-Rave in terms of security and performance. Specifically, we will quantify the level of randomness introduced into the target application runtime. We also evaluate the time it takes to perform code analysis and transformations to understand what types of overheads are induced through CRIU-Rave's code modifications.

Experimental setup: Rave was evaluated on an x86-64 machine with an Intel i7-6500U CPU clocked at 2.5 GHz. The CPU has two physical cores, two threads per core (4 threads in total). This machine has 16 GB of DDR4 RAM. For the OS, it is running Ubuntu 20.04 LTS (kernel version 5.8). To compile and link benchmarks and other test programs, we used GCC version 10.3.0 and binutils version 2.34.

Rave was tested on several programs including: SPEC CPU 2017, SNU C version of NAS Parallel Benchmarks (NPB), NGINX, Redis, Lighttpd, and MySQL server. All programs were compiled with flags `-fno-omit-frame-pointer` and `-mno-red-zone`. They are also dynamically linked. Note that Rave only randomizes the target application code itself, the external libraries, such as `glibc`, are not touched.

Figure 4 shows the total number of functions and the number of randomizable functions in the benchmark binaries mentioned above. Here the randomizable functions are those functions with stack local variables inside (`push`/`pop` instructions of the function prologue and epilogue in our case). The result was obtained from `librave`'s analysis. Naturally, smaller applications have fewer (randomizable) functions. Two extreme cases are NPB `EP` with zero and `IS` with one randomizable function (because of the small codebase). Therefore, they cannot gain security benefits through Rave. In contrast, larger applications like MySQL have a more extensive codebase than others; thus, they have more (randomizable) functions (*i.e.,* more than 10x functions in MySQL than other applications). This is also reflected in the performance of function analysis reported in Section 4.2.

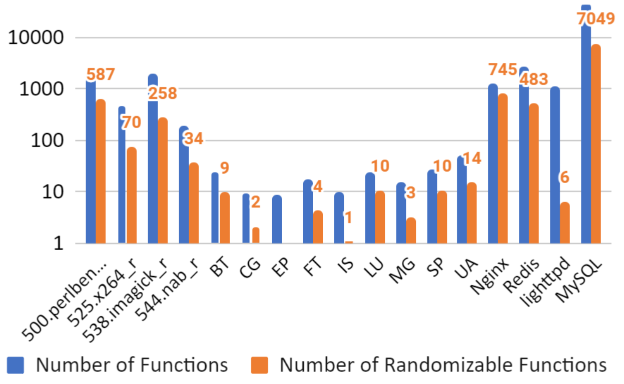

Figure 4: The number of (randomizable) functions for tested binaries. EP had zero randomizable functions.

4.1 Security Analysis

Similar to existing re-randomization or diversification-based works [6, 8, 11, 33, 36], Rave cannot guarantee any attacks will not succeed, but it lowers the chance for an attacker to guess the location of a shuffled stack slot or find the software/hardware stack details. Rave also shares the security benefits of moving target defense systems [8, 17, 18], and the security benefits depend on the physically distributed environment of deployment. Therefore, we only analyze the entropy incurred in each re-randomization epoch for stack slot shuffling and report the additional time cost for code analysis and transformation in this paper.

We quantify the quality of randomization by measuring the average entropy of the program stack states. We define the entropy of the program stack as the number of stack slot locations a stack variable can fill in. Therefore, randomizable functions will always have an entropy of two or higher. In our current implementation, we only permute the order of stack objects. Therefore, a function's entropy is equal to the number of permutable stack slots. For example, if there are three stack slots, there are three possible locations a particular slot could be in, thus that function has an entropy of three. However, we anticipate that future implementation of librave could further utilize the allocated but unused stack spaces to increase the overall entropy.

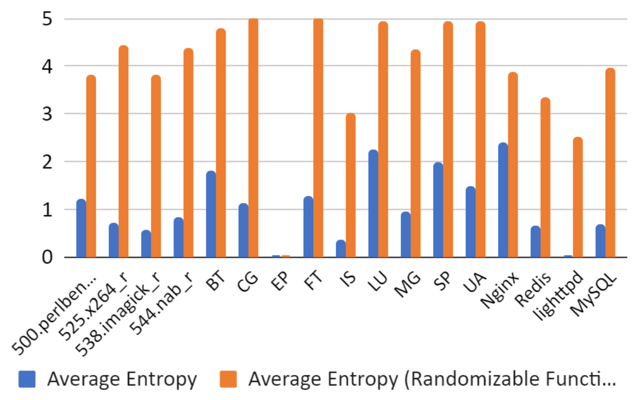

Figure 5: Quality of randomization for various applications.

Figure 5 shows the average entropy for an application assuming all functions are called with uniform probability. For particular workflows or attacks, the true entropy may vary and can be calculated given function call frequency. However, we assume any code in the application is equally vulnerable to, for example, code reuse attacks. For most applications, the total average entropy is less than two, which implies that an attacker can generally guess where stack slots will be located regardless of randomization. The only application here that has a high enough entropy to qualify in disrupting memory corruption attacks is NGINX. This is because more than half of the functions in NGINX are randomizable. With an average entropy of 2.39, an attacker will have an average probability of $\frac{1}{2^{2.39}} \approx 19.1\%$ in guessing the location of a stack slot. Do note that in some cases attackers will generally have to chain together multiple stack slots to execute an attack. In the case where three stack slots are required, there is an average probability of $19.2\%^3 \approx 0.7\%$ that the attacker will find all three slots. We also find the average entropy of randomizable functions is much higher, reaching about 4 bits. This gives librave a more significant chance for relocating the program state. In the future, we plan to break the stack slot swapping constraint in librave and fully utilize the spare spaces in each allocated stack frame to have even higher entropy.

4.2 Performance Evaluation

We evaluated the performance overhead by measuring the time librave takes to analyze and transform binaries. The primary focus for performance overhead lies in the time it takes to analyze and transform binaries. For our current prototype, this overhead is *added before each process restoration phase.* The runtime overhead for CRIU-Rave to migrate a live process is theoretically identical to that of the vanilla CRIU. The only difference is that CRIU-Rave loads randomized pages from the librave code cache, whereas the vanilla CRIU load them from the dumped process images. Therefore, the overall overhead for the current prototype of the Rave framework is incurred only during process restoration (even that is partially absorbed by network latency and file I/O). There is also plenty of room for optimization in librave, namely, opportunities in improving concurrent function analysis and transformation. Note that this overhead does not affect the application's runtime because transformations happen out-of-band in the CRIU lazy-pages process. Nevertheless, we measured the additional time cost incurred by each re-randomization phase (analysis and transformation).

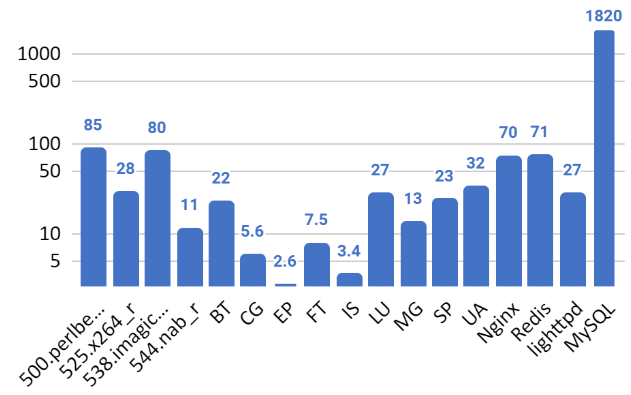

Figure 6: Average time taken to analysis and transform various binaries (ms).

Figure 6 depicts the time cost associated with randomization per program. Even for large applications, like MySQL (which had 42470 functions to analyze, 7049 to transform), it takes less than two seconds from Rave initialization of the unmodified binary to the complete transformation of the target. This is an acceptable performance hit since Rave runs in a migration context where there is already much variability in the checkpoint/restore process. For smaller applications, such as NGINX, Redis and Lighttpd, the introduced overheads are acceptable (less than 0.1 seconds). This is likely due to the smaller number of (randomizable) functions in these applications (Figure 4). The geometric standard deviation for performance was about 7.74%. Upon closer inspection, we see that this standard deviation number (a bit high) is skewed by the performance times of smaller applications (including ones like NPB's EP, which has no randomizable functions). For these smaller applications, standard deviation is very high because the analysis and transformation runtimes are clouded by OS support (e.g. I/O, memory allocation). For larger applications (like MySQL), the standard deviation was only 1.05%, which equates to about 1.8 ± 0.02 seconds.

5 DISCUSSION AND FUTURE WORKS

We have demonstrated a use case of Rave for diversifying the program state and execution locations in a transparent and non-intrusive way. Rave can be used in a distributed environment or locally on the same node. In either case, CRIU maintains the program state, such as TCP connection and opened files, to be alive after the process restoration [10]. We anticipate that Rave can be further developed as a general framework for live program transformation to solve other system and security problems, such as live code updates [26], software feature customization (debloating) [19, 27], etc. For example, we can extend `librave` to add a policy for updating vulnerable code pages (a.k.a., live software patching). It only requires the user to checkpoint the process and restore (reload) the process with updated code pages if they were to use CRIU-Rave.

Rave also provides the capability to unwind the stack and check if the vulnerable code is being used, which is critical for deciding whether the code patching is safe. Similarly, we can also write a `librave` policy to dynamically wipe out undesired code to reduce the attack surface. Such undesired code can be initialization-related code that is never used once the program is completed. We anticipate that this could further improve the state-of-the-art of existing software debloating works [19, 27].

Besides extending Rave for different security purposes, there is some design space to optimize Rave's performance. As shown in Section 4.2, Rave brings observable overhead for analyzing large applications' randomizable functions. We anticipate that Rave can utilize a separate thread parallel to the target program execution to save this extra time. We leave this optimization as future works.

6 RELATED WORK

Parallel to moving target defense, runtime code and data space re-randomization is another way for dynamic software diversification. The basic idea is to play hide-and-seek with the attacker. By shuffling locations of targets [18, 33], or by reducing the predictability of vulnerable components in a program [6, 36], we can significantly increase the difficulty of attack. Some of these works attempt to disrupt code-reuse attacks specifically, while others fight memory corruption directly. Shuffler [36] is a type of fine-grained runtime re-randomization works where functions in the binary are continuously relocated. By continuously reorganizing code locations, attackers will have a more difficult time trying to reuse gadget strings. Shuffler dynamically generates new code (and unwinds the stack) with a thread within the target address space and thus suffers from potential attacks [36]. TASR [6] similarly re-randomizes the code with components that reside in both kernel-space and user-space. Unlike existing works, Rave dynamically re-randomizes the code and data using user-space page faults handling, thus the re-randomization agent is separated from the target.

Rave was also inspired by recent works on data-space randomization [2, 5, 24, 28]. Smokestack [2] is an interesting work where stack frames are randomized as a part of the binary's runtime. Using a modified version of LLVM, Aga et al. instrument binaries such that several permutations of functions' stack allocations are available at runtime. The randomization instrumentation randomly chooses among these permutations when a function is called, thus introducing a different stack layout each time a function is called. The authors have shown that this method is effective in defending against DOP attacks, but it incurs a non-trivial runtime overhead in some cases [2]. CoDaRR [28] protects the data space objects by encrypting the data objects with `xor` and continuously re-randomizing the masks used in loads and stores to prevent sensitive data from being leaked. On the contrary, Rave is designed as a framework for dynamic program state re-randomization. Thus we believe Rave can enhance the security of these systems with an external randomization agent. Chameleon [24] is the most related work to Rave, it creates extra entropy with a customized compiler and controls the target's memory through a `ptrace`-based monitor. However, it requires the application to be statically linked, with self-compiled versions of libraries like *libc*. Thus applications, like Redis or MySQL, do not run under Chameleon's instrumentation (partially because they requires access to the dynamic linker to call `dlopen()`). Moreover, most existing works mentioned above focus on a single node environment, whereas Rave is naturally integrated with CRIU for seamlessly program instance re-location.

7 CONCLUSION

We have presented the design and implementation of Rave, a system that dynamically updates the program's state and location. Rave's main innovation is a novel out-of-bound re-randomization agent serving randomized code and data pages upon page faults. Rave can be seamlessly integrated into CRIU's lazy migration (post-copy memory migration) to enable dynamic process re-location. We have built a prototype of Rave and evaluated the prototype using four server applications and 13 applications from the SPEC CPU 2017 and the SNU C version of NAS Parallel Benchmarks (NPB) benchmark suites. The evaluation results show that Rave increases the internal program state entropy with an additional $\approx$200 *ms* migration time overhead on average.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by US NAVSEA/NSWC Dahlgren under grant N00174-20-1-0009, US Office of Naval Research (ONR) under grants N00014-18-1-2022 and N00014-19-1-2493, and US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 4.

[2] Misiker Tadesse Aga and Todd M. Austin. 2019. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 26–36. https://doi.org/10.1109/CGO.2019.8661202

[3] Kapil Arya, Gene Cooperman, Rohan Garg, Jiajun Cao, and Artem Polyakov. 2022. DMTCP: Distributed MultiThreaded CheckPointing. https://dmtcp.sourceforge.io/.

[4] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. *Proc. 23rd Usenix Security Sym* (2014), 433–447.

[5] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5137)*, Diego Zamboni (Ed.). Springer, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[7] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 985–999. https://doi.org/10.1109/SP.2019.00076

[8] Jin-Hee Cho, Dilli P. Sharma, Hooman Alavizadeh, Seunghyun Yoon, Noam Ben-Asher, Terrence J. Moore, Dong Seong Kim, Hyuk Lim, and Frederica Free-Nelson. 2020. Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense. *IEEE Commun. Surv. Tutorials* 22, 1 (2020), 709–745. https://doi.org/10.1109/COMST.2019.2963791

[9] Eager Consulting. 2021. The DWARF Debugging Standard. https://dwarfstd.org/.

[10] CRIU. 2022. Checkpoint Restore in Userspace. https://criu.org/Main_Page.

[11] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (just-in-time) Return-oriented Programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)* (2015).

[12] Jake Edge. 2013. Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/.

[13] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 227–242.

[14] heapspray Accessed: 2019-02-14. Heap spraying. https://en.wikipedia.org/wiki/Heap_spraying.

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

[16] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1868–1882. https://doi.org/10.1145/3243734.3243739

[17] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. 2012. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 127–132.

[18] Jajodia, Sushil and Ghosh, Anup K and Swarup, Vipin and Wang, Cliff and Wang, X Sean. 2011. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Vol. 54. Springer Science & Business Media.

[19] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, Radu F. Babiceanu, Hélène Waeselynck, Raymond A. Paul, Bojan Cukic, and Jie Xu (Eds.). IEEE Computer Society, 122–131. https://doi.org/10.1109/HASE.2016.27

[20] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-grained Randomization of Commodity Software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 339–348.

[21] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 147–161. https://doi.org/10.1145/3341301.3359662

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA).

[23] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*.

[24] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. 2020. Dynamic and Secure Memory Transformation in Userspace. In *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12308)*. Springer, 237–256. https://doi.org/10.1007/978-3-030-58951-6_12

[25] Alex Ionescu Mark Russinovich, David Solomon. 2012. *Windows Internals, 6th Edition*. Microsoft Press.

[26] Iulian Neamtiu, Michael W. Hicks, Gareth Paul Stoyle, and Manuel Oriol. 2006. Practical dynamic software updating for C. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 72–83. https://doi.org/10.1145/1133981.1133991

[27] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[28] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese (Eds.). ACM, 494–505. https://doi.org/10.1145/3320269.3384757

[29] Mike Rapoport. 2021. userfaultfd(2) — Linux manual page . https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[30] Ravi S Sandhu and Pierangela Samarati. 1994. Access control: principle and practice. *IEEE communications magazine* 32, 9 (1994), 40–48.

[31] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 574–588.

[32] Virtuozzo. 2022. Open source container-based virtualization for Linux. https://openvz.org/.

[33] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, Manuel Egele and Leyla Bilge (Eds.). USENIX Association, 427–442. https://www.usenix.org/conference/raid2020/presentation/wang-xiaoguang

[34] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*.

[35] Wikipedia. 2021. Ptrace. http://en.wikipedia.org/wiki/Ptrace.

[36] 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 *OSDI*. 367–382.