KASLR in the age of MicroVMs

Benjamin Holmes
Vassar College
Poughkeepsie, New York, USA
bholmes@vassar.edu

Jason Waterman
Vassar College
Poughkeepsie, New York, USA
jawaterman@vassar.edu

Dan Williams
Virginia Tech
Blacksburg, Virginia, USA
IBM T.J. Watson Research Center
Yorktown Heights, New York, USA
djwillia@vt.edu

Abstract

Address space layout randomization (ASLR) is a widely used component of computer security aimed at preventing code reuse and/or data-only attacks. Modern kernels utilize kernel ASLR (KASLR) and finer-grained forms, such as functional granular KASLR (FGKASLR), but do so as part of an inefficient bootstrapping process we call *bootstrap self-randomization*. Meanwhile, under increasing pressure to optimize their boot times, *microVM* architectures such as AWS Firecracker have resorted to eliminating bootstrapping steps, particularly decompression and relocation from the guest kernel boot process, leaving them without KASLR. In this paper, we present *in-monitor KASLR*, in which the virtual machine monitor efficiently implements KASLR for the guest kernel by skipping the expensive kernel self-relocation steps. We prototype in-monitor KASLR and FGKASLR in the open-source Firecracker virtual machine monitor demonstrating, on a microVM configured kernel, boot times 22% and 16% faster than bootstrapped KASLR and FGKASLR methods, respectively. We also show the low overhead of in-monitor KASLR, with only 4% (2 ms) increase in boot times on average compared to a kernel without KASLR. We also discuss the implications and future opportunities for in-monitor approaches.

CCS Concepts: • **Security and privacy → Virtualization and security**.

Keywords: Virtual Machines, Operating Systems, KASLR, MicroVM, Security

ACM Reference Format:
Benjamin Holmes, Jason Waterman, and Dan Williams. 2022. KASLR in the age of MicroVMs. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3492321.3519578

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519578

1 Introduction

Never do anything yourself that others can do for you.
—Agatha Christie (1890-1976)

The operating system kernel is highly privileged and often shared, especially in a cloud computing environment. Its security is critical. Even when running as a guest kernel in a virtual machine (VM), security is important, either as a first layer of defense for the host or to isolate multiple mistrusting tenants running containers atop the guest kernel. In any case, kernels have evolved a suite of techniques to mitigate attacks. One technique used by kernels to frustrate attackers, specifically code reuse attacks, such as return-oriented programming [61], is address space layout randomization (ASLR). Kernel ASLR (KASLR) consists primarily of randomizing the base address where the kernel and kernel modules are loaded, or, more recently, randomizing individual function sections (FGKASLR). Randomization makes code reuse fragments (gadgets) more difficult for attackers to locate. Some form of KASLR is used in all major OS kernels. It has been upstream in Linux since 3.14 (2014), and made default as of 4.12 (2017). Traditionally, prior components in the boot sequence (e.g., bootloaders) have been unaware of KASLR, so the OS kernel relocates itself in memory at boot time in a process we refer to as *bootstrap self-randomization*, which is implemented as part of the kernel's bootstrapping routines. Bootstrap self-randomization is depicted in Figure 1(a).

Meanwhile, lightweight virtualization environments, or *microVMs*, are gaining popularity as a secure unit of execution that can meet the increasingly stringent performance demands of emerging cloud programming models like serverless computing [1–3, 5], and modern virtual machine monitors (VMMs) bypass much of the traditional boot sequence, especially bootstrapping logic [18]. For example, on x86_64 processors, lightweight monitors bypass the awkward iteration of CPU modes, descriptor tables and page tables to transition from a 16-bit "real mode" environment to a 64-bit "long mode" environment by executing the kernel directly from its 64-bit entry point. Lightweight monitors also bypass kernel decompression, opting instead to directly boot

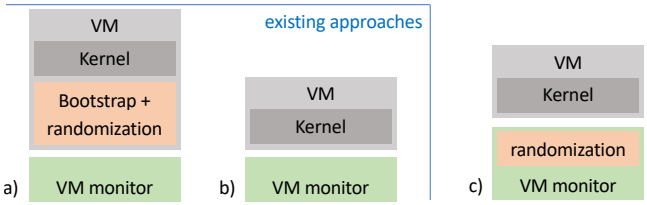


Figure 1. Overview: (a) guests on standard virtualization stacks perform inefficient bootstrapping and randomization early in their boot; (b) fast-booting modern monitors eliminate bootstrapping (and randomization) entirely; (c) in-monitor KASLR efficiently implements randomization in the monitor instead of the guest.

a kernel ELF file. Unfortunately, perhaps carried away in their attempts to avoid the inefficiencies of the bootstrapping process, lightweight monitors bypass randomization, as depicted in Figure 1(b).

In this paper, we make the observation not only that KASLR is bypassed in modern monitors, but that the traditional approach of bootstrap self-randomization is incompatible with the goal of minimized boot time imposed by evolving cloud computing workloads. Instead, we propose *in-monitor KASLR*, in which the task of randomization is moved to the virtual machine monitor. In-monitor KASLR is depicted in Figure 1(c). It exploits the observation that fundamentally, *bootstrapping (including self-randomization) should only be undertaken when there is no controlling principal that can set up an appropriate environment.* Lightweight monitors are such a principal for virtualized environments; indeed, this is the same reason why they can bypass bootstrapping steps for processor modes.

We implement in-monitor KASLR in the open-source AWS Firecracker[1] [18] Linux/KVM-based virtual machine monitor, a modern hypervisor implementation aimed at addressing next-generation cloud workloads such as serverless computing. Our implementation also introduces state-of-the-art, function-granular randomization (FGKASLR) within the monitor by performing steps similar to those taken by its in-development implementation in Linux. By performing randomization within the monitor, we can amortize the kernel load process and the guest no longer needs to perform inefficient relocations that arise from the traditional methods of decompression and self-randomization. Moreover, our approach requires no changes to the guest Linux kernel.

The value proposition of microVM monitors like Firecracker rests on their ability to boot workloads quickly (e.g., less than 150 ms [18]) to support serverless computing. We evaluate in-monitor randomization and find it can meet this threshold even while providing function-granular randomization, demonstrating boot times with coarse/fine-grained randomization as low as 16 ms/28 ms in a minimally configured kernel. We also show our in-monitor implementation can speed up boot times by 42%/38%, on average, over existing self-randomization approaches. Finally, we show low overhead for in-monitor KASLR, showing as little as a 2.3% increase in boot times over a kernel without KASLR.

In summary, we make these primary contributions:

- Thorough study of the root cause of bootstrap-related overhead including decompression and relocation.
- Identification of the omission of KASLR in current fast-booting microVM architectures.
- Design and implementation of coarse and fine-grained *in-monitor KASLR* in AWS Firecracker.
- Evaluation of in-monitor (FG)KASLR.

The rest of this paper is organized as follows: Section 2 and 3 provide background on microVMs, (FG)KASLR, and the unfortunate current realities of their (nonexistent) combination. Section 4 describes the design and implementation of in-monitor (FG)KASLR. Section 5 presents an evaluation of in-monitor (FG)KASLR. Section 6 discusses implications and future directions, Section 7 surveys related work, and Section 8 concludes.

2 microVMs

In this section, we first describe the important trends guiding microVMs and their focus on low boot times. We then provide background on the Linux early boot process, break down the source of bootstrap-related overhead, and identify how microVM monitors currently eliminate that overhead.

2.1 The rise of microVMs

Since the inception of the cloud, there has been a general trend towards even more fine-grained elasticity, in the form of renting computing resources on a pay-as-you-go basis. Whereas workloads on the cloud originally resembled traditional systems (VMs running for days, months, or years), they have been giving way to workloads organized as fine-grained application components that individually scale. The most extreme incarnation of this trend is visible in the function-as-a-service, or *serverless computing* domain. Embodied by systems like Amazon Lambda [2], Microsoft Azure Functions [3], IBM Cloud Functions [5], and Apache OpenWhisk [1], individual units of execution, called *lambdas* or *actions*, only run for seconds or even milliseconds. In order for cloud providers to profit from such a fine-grained elastic cloud model, the time to create a new instance on the cloud must be relatively short compared to its execution time, as cloud users only pay for their execution time. In general, we highlight two fundamental shifts in the cloud landscape: increasingly, the unit of execution in the cloud 1) *has a short lifetime* and 2) *must boot quickly.*

[1] https://github.com/firecracker-microvm/firecracker/

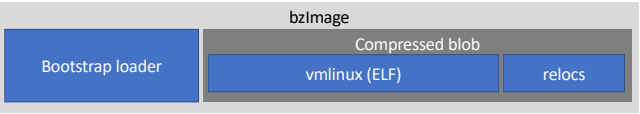


Figure 2. Components of Linux bzImage

Cloud computing as we know it was formed around the use of VMs; Amazon EC2 is the canonical example of providing compute resource in a pay-as-you-go model via VMs. Yet VMs have traditionally been heavyweight, especially in boot time, often taking minutes to boot [52]. Under pressure from lightweight but less well-isolated *containers* [4, 10], lightweight virtual machine monitors emerged from both academia (e.g., LightVM [51]) and industry [6, 18, 64]. AWS coined the term *microVM* in the context of Firecracker [18]. Lightweight monitors have been applied to the container ecosystem through Intel's clear containers project [64] and later the Kata containers project [7], which touts "the speed of containers, the security of VMs".

Along with lightweight monitors, fast boot times critically depend on reducing the size and inefficiency within the guest. Firecracker and Kata Containers both distribute Linux configurations for lightweight guest kernels. Unikernels [9, 11–13, 24, 45, 50, 53, 55, 62] represent perhaps the most extreme case of reducing the size of a guest, by replacing the guest kernel with only those components of a library OS needed to directly run on a virtual hardware interface. More recently, similar approaches have been applied to Linux primarily via guest kernel configuration in Lupine Linux [48].

The combination of lightweight monitors and guests has allowed virtualization to meet the requirements imposed by modern cloud workloads like serverless, while maintaining a VM-like security posture. Next, we will describe one part of achieving such a fast boot time: bypassing the bootstrap loader (and, unfortunately, (FG)KASLR).

2.2 Booting a (micro)VM

For years, the most common way to distribute Linux kernels has been to compress the kernel and link it to a small *bootstrap loader* that would decompress then jump to the decompressed kernel. Here, we detail the overheads of this booting practice and subsequently the techniques microVMs use to bypass bootstrap overhead.

Booting a bzImage. The standard format for the kernel—to be loaded on physical or virtual machines— is commonly named `bzImage` or `vmlinuz`. As shown in Figure 2, bzImage is actually a concatenation of two components: a small program we refer to as the *bootstrap loader*, and a compressed blob, which when decompressed, is the executable kernel image (`vmlinux`) and a list of *relocations*, the addresses in the kernel text segment that must be adjusted in case the kernel is relocated.

During boot, the bootstrap loader begins executing with a boot stack and heap using a direct, one-to-one memory map. At this point, the bootloader has loaded the bzImage (containing both the bootstrap loader and compressed kernel proper), initrd, and command line arguments to known locations in memory. The first task of the bootstrap loader is to decompress the kernel to an available location in memory. Finally, the bootstrap loader parses the ELF file, loads segments into memory, and jumps to the uncompressed kernel (`startup_64`).

The astute reader will notice that, in this description, we did not describe the use of the relocation information within the bzImage that directly follows the uncompressed kernel image. On x86_64, no relocation is needed unless KASLR is in use (as described in Section 3.2), so for now we ignore the relocations.

Direct kernel boot. Modern VM monitors (such as Firecracker, Cloud Hypervisor, and QEMU) support eliminating bootstrap overhead by directly booting into an *uncompressed* kernel image instead of the bzImage, without loading or running the bootstrap loader. Currently, there are two protocols that VMMs can use to boot into an uncompressed (x86_64) Linux kernel: the Linux boot protocol [15] and the Xen [21] PVH protocol [14]. Both load the raw `vmlinux` ELF file, which is usually compressed in the bzImage, and boot directly into it, avoiding decompression costs. The main differences in boot protocols lie in how boot-time system information is conveyed to the nascent kernel.

bzImage and uncompressed kernel boot times. To understand the benefits of direct kernel boot, we compare the boot times in the Firecracker VMM for compressed (bzImage) and uncompressed (vmlinux) kernels. To capture the effects of increasingly lightweight guest kernels, we further experiment with three configurations of the Linux 5.11.0-rc3 kernel. The first, *Ubuntu*, uses the configuration from Ubuntu 18.05.4, representing a relatively large, standard distribution kernel. The second, *AWS*, uses the reference kernel configuration from AWS Firecracker, representing a state-of-the-art, medium, general-purpose microVM kernel. The third configuration, *Lupine*, uses a configuration from Lupine Linux [48], representing a small single-purpose kernel configuration, approaching a slightly more futuristic unikernel-based environment. A description of all kernels used in our experiments is in Table 1 and our full experimental setup is described in Section 5.1.

Firecracker does not natively support compressed bzImages, so to obtain bzImage boot times we used a lightly-modified version of Firecracker (described in Section 5.1) which adds bzImage support. To determine which kernel compression algorithm boots the fastest, we measured six

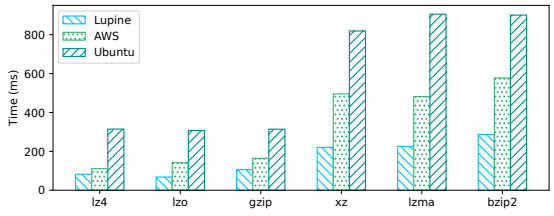


Figure 3. Overhead of boot times under various compression schemes.

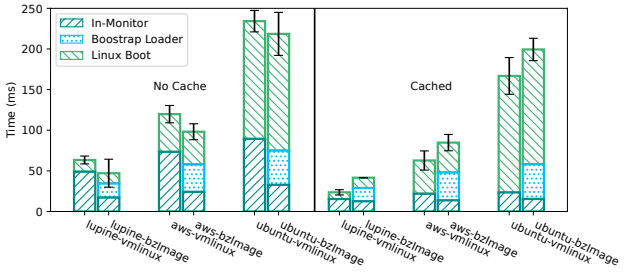


Figure 4. Boot times for compressed (with LZ4) and uncompressed kernels

different compression schemes as shown in Figure 3. To optimize boot performance of the bzImages in our experiments, we configure the guest kernels to use LZ4, rather than the deafult, `gzip`, as it is the fastest booting compression algorithm on our machine.

Figure 4 shows the results of booting the various guest kernels. Overall boot time is measured from the call to execute Firecracker to just after the guest kernel's `init` process is run. We break down the boot time as *In-Monitor*, the time spent in Firecracker before jumping to the kernel and entering the guest context, *Bootstrap Loader*, the time between executing the bootstrap loader and jumping to the uncompressed kernel, and *Linux Boot*, the time between the jump to the uncompressed kernel to just after the `init` process is run. Since (the lack of) disk caching can greatly impact boot time, we also looked at the effects of disk caching. For the *No Cache* experiments, we drop the caches (pagecache, dentries, and inodes) right before each kernel boot. In the *Cached* experiments, we warmed the cache by booting the kernel 5 times before starting to measure boot times. All times shown are the average over 100 kernel boots.

When the kernel is not cached and has to be read directly from disk, compressed kernels all had faster boot times, due to decreased I/O operations compared to an uncompressed kernel, showing that the cost of reading the uncompressed image from disk is greater than the cost of bzImage decompression on our system. In this case, a direct boot is slower than a bzImage by 26% with *Lupine*, 18% with *AWS*, and 7% with *Ubuntu*. This helps explain why kernels are traditionally compressed; when booting on bare metal, the reduced I/O cost allows compressed kernels to boot faster where a warm cache is impossible.

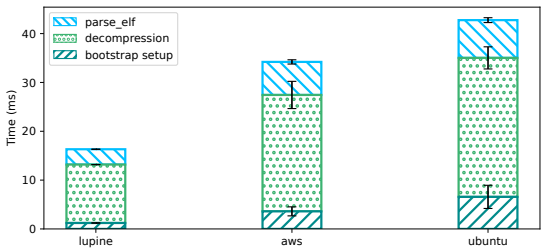


Figure 5. Breakdown in cost of in the Linux bootstrapping process

However, the situation reverses when the kernel image is cached and does not have to be read from disk. In addition to cached boot times being (unsurprisingly) faster than corresponding uncached kernel boot times, Figure 4 also shows that a direct boot of an uncompressed kernel is the better strategy in the cached case. Now, a direct boot is faster by 36% with *Lupine*, 33% with *AWS*, and 20% with *Ubuntu*. These results help to explain the lack of enthusiasm for supporting bzImage images in most modern hypervisors; cloud providers booting VMs for general-purpose workloads will likely reuse the same kernel for each instance, and cached, uncompressed kernels have the lowest overhead, so we assume that kernels will be cached for the rest of the paper. See Section 6.1 for further discussion about when this assumption may not hold.

Digging deeper, we also examine the costs of each step in the bootstrap loader, which is bypassed in the direct boot of an uncompressed kernel. The results of these microbenchmarks are shown in Figure 5. We can see that most of the time in the bootstrap loader, up to 73%, is spent on decompression.

To test the validity of these experiments across multiple modern hypervisors, we repeated the above experiments using QEMU v6.0.94, and we can draw the same conclusions. Due to differences in the implementations of Firecracker and QEMU, the time spent in the hypervisor varies. With cached kernels in QEMU, a direct boot is faster than a bzImage by 2% with *Lupine*, 33% with *AWS*, and 17% with *Ubuntu*. The takeaway is that in both VMMs, an uncompressed and cached kernel is the fastest way to boot Linux.

To summarize, by eliminating the bootstrap loader and directly booting an uncompressed kernel, hypervisors like Firecracker are saving up to 36% in boot times over a traditional, compressed kernel boot. Unfortunately, as we will see next, this approach is incompatible with KASLR in its current form.

3 KASLR

As W^X and SMAP/SMEP have all but eliminated attacks consisting of user code injection into the kernel, attackers have largely resorted to various forms of code reuse attacks. Code reuse attacks typically require knowledge of the locations of various structures within an address space: for example, the location of the stack or of various code snippets that could form gadgets for ROP attacks [61]. Two complementary defense strategies have emerged to make code reuse fragments (gadgets) both *(a) hard for an attacker to use*, by attempting to enforce control flow integrity (CFI) [16], and *(b) hard for an attacker to find*, through techniques such as address space layout randomization (ASLR). Due to its simplicity and low cost, ASLR has been rapidly and widely adopted, for both userspace processes and the kernel (KASLR). Operating system kernels, including the Linux kernel since version 3.14, have adopted KASLR as a configurable defense option; it has been a default option since version 4.12.

KASLR was introduced to the Linux kernel in a coarse grained fashion by randomizing the base at which the kernel was loaded at boot time [30, 33]. Attackers would need to guess the offset to find code reuse gadgets. More recently, fine-grained kernel randomization has been proposed in the form of FGKASLR [17], which randomizes kernel addresses at a function-level granularity. In the rest of this section, we describe why KASLR remains relevant and important, show how KASLR is implemented in the boot process, and highlight its absence for microVMs. We then revisit compression to argue that it is not the sole cause of slowdown in the boot process, as there are historical artifacts, and fundamental design properties that contribute to the inefficiency of a bzImage boot.

3.1 The Enduring Relevance of KASLR

Since its inception, KASLR has faced criticism in terms of its effectiveness [63], yet it remains an important component of system security. We focus on two criticisms. First, unlike userspace applications, the kernel will only be randomized once over its (presumably long) lifetime. Furthermore, the monolithic nature of Linux makes it unsuitable for re-randomization techniques proposed for microkernels [36]. Second, the prevalence of information leaks in the kernel [31, 46] are particularly devastating due to the coarse granularity of KASLR (e.g., CONFIG_RANDOMIZE_BASE results in the entire text of the kernel sharing the same offset).

However, these criticisms are eroding over time. First, the previously mentioned trends in the cloud computing domain towards microVMs suggest a use case in which (guest) kernel instances are short lived. For example, AWS Firecracker [18] is used for serverless computing workloads for AWS Lambda in which the lifetime of the VM could be as short as a function invocation. With KASLR, each instance would utilize a different randomization. Second, concerted efforts to avoid leaking kernel pointers have accompanied KASLR, such as `kptr_restrict`, which controls a format specifier that converts kernel address pointers to zeros when printed [60]. More recently, inspired by application self-randomization [29], fine-grained, FGKASLR [17] has been implemented and is being discussed on the Linux Kernel Mailing List for inclusion upstream to improve entropy and add a layer of defense used in tandem with coarse-grained KASLR. With the addition of FGKASLR, the virtual space enjoys randomization at the function level, on top of the KASLR base offset. This greatly decreases the value of an information leak since once can only gain knowledge of a single function's location. This means that attackers will not be able to explot the entire kernel with a single information leak, minimizing the damage done by potentially buggy code.

Other concerns over the efficacy of KASLR have emerged in the context of microarchitectural side channels. As KASLR forms a first barrier for an attacker to overcome, breaking KASLR has recently become a proving ground for emerging side-channel attacks, including on virtual machines in the cloud [22]. To date, KASLR has been broken using memory management side channels [41], prefetch side channels [38], transient executed loads [28], out-of-order execution [27], the branch target buffer [34], and hardware instructions to support transactions [42]. However, while side channels remain a concern, mitigations such as KAISER [37] or LAZARUS [35] continue to be adapted to close side channels [28] indicating that KASLR will likely remain a useful and popular component to improve kernel security.

3.2 KASLR in the bootstrap loader

As described in Section 2.2, the Linux kernel is typically distributed in a compressed format. Early in the boot sequence, the bootstrap loader is responsible for decompressing the kernel, parsing and loading the resulting decompressed ELF image, and if (FG)KASLR is enabled, performing the necessary relocations.

Initially KASLR was implemented as an offset in the physical dimension; by maintaining a fixed offset between physical and virtual addresses, this additionally resulted in randomization in the virtual address space [30, 33]. A few years later, it was noticed randomization in the virtual address space could be decoupled from randomization in the physical address space, allowing more entropy in the virtual address space [40]. Virtual addresses, not physical addresses, are necessary for code reuse. For example, return addresses, jump targets, and data are all addressed with virtual addresses. Moreover, attacks that do require knowledge of physical addresses, such as DRAMA [58] or rowhammer [44] attacks, often rely on low order address bits, which do not change regardless of virtual/physical randomization, especially when large pages (e.g., 2 MB or 1 GB) are in use. Hence, we focus our discussion on randomization in virtual address space.

Once the bootstrap loader selects a randomized and aligned virtual address offset, it can begin to fix up the addresses present in the loaded kernel. Linux divides the relocations into three types: 1) 64-bit addresses that need an offset added to them, 2) 32-bit virtual addresses that need an offset added to them, and 3) 32-bit virtual addresses that need an offset subtracted from them (inverse relocations). The bootstrap loader iterates through each relocation entry, provided by the `relocs` section of the bzImage (Figure 2). Each entry is a pointer to the physical address that will hold the virtual address to be modified, and the bootstrap loader adds or subtracts the offset. In contrast to the large overhead from decompression, these operations comprise at most 8.8% of the time spent in the bootstrap loader, making KASLR a good candidate for relocation to the VMM. The overhead of KASLR is further discussed in Section 5.2.

When using FGKASLR, every function in the kernel's text section has its own offset from its original location. To do this, the compiler places each function into its own section (`.text.<function name>`) in the kernel ELF when compiled with the GCC flag `-ffunction-sections`. During the bootstrap process, additional parsing of the kernel ELF must be done to retrieve the section headers for each section to be randomized, the string table, and the symbol table, to extract symbols used to handle relocations. The sections are then shuffled and re-aligned contiguously, giving each function a unique random offset from its original location in the kernel. This means that more work must now be done when handling relocations, and at each entry the function section headers that were affected by randomization are searched using a binary search to determine if they contain an address that points to a function that was moved from its original location. If so, the offset between the function's original location and its new location is added to the address, as well as the KASLR virtual offset. The same process must be done for `/proc/kallsyms`, the exception table, and the ORC stack unwinder table[2], because addresses to symbols affected by FGKASLR must be updated to reflect their new location. The increase of complexity over KASLR comes at a performance cost, but significantly decreases the value of a single data leak, as an attacker can only gain knowledge of a single function's location.

3.3 Combining KASLR with microVMs

Currently, both KASLR and FGKASLR are tightly coupled with the bootstrap loader. But as we saw in Section 2.2, microVMs seek to improve boot times by directly booting uncompressed kernels, eliminating the bootstrap overhead (and the ability to do (FG)KASLR). Figure 5 shows that most of the time in the bootstrap loader stems from decompression, so in this section we explore modifications to bootstrap loader to boot uncompressed kernels. First, we implemented our own compression scheme, *compression-none*, that simply leaves the kernel uncompressed when linked into a bzImage, and during decompression it is copied to the location that it expects to run. Figure 6 shows that without further modification to the bootstrap loader, *compression-none* leaves us with more overhead than an LZ4 compressed kernel. This is because the overhead comes not only from the cost of decompression, but also from the cost of relocating the kernel, and uncompressed kernels can be up to ten times larger. During a bzImage boot, the following occurs:

1. Monitor reads bzImage into guest memory and jumps to bootstrap loader entry point
2. Bootstrap loader copies compressed kernel out of the way for in-place decompression
3. Kernel is decompressed to its location in physical memory where it is configured to run
4. Bootstrap loader jumps to decompressed kernel

But in the case of an uncompressed kernel, steps 2 and 3 are not necessary and can be eliminated, giving us a fully optimized self-bootstrapped approach for uncompressed kernels. Step 2 can be eliminated by simply skipping it, since we are avoiding decompression so there is no risk of the kernel being overwritten. To remove step 3, the bootstrap loader and the bzImage linking process must be changed. The kernel expects to be loaded and executed at a minimum address of CONFIG_PHYSICAL_START, which cannot be set lower than 0x1000000. The load address must also be aligned to MIN_KERNEL_ALIGN. To accomplish this without copying the kernel, we first link the kernel to the bootstrap loader code aligned to MIN_KERNEL_ALIGN. Then, instead of loading the bzImage into guest memory at the default 0x100000, we increase the load address to 0x1000000. Because we have now met the constraints to execute the kernel, we can simply jump to the entry address of the kernel (*startup_64*).

By identifying and eliminating redundancies in the bootstrap loader that are no longer necessary with respect to a modern VMM use case, we have devised the most time efficient self-bootstrapping method. But as Figure 6 shows,

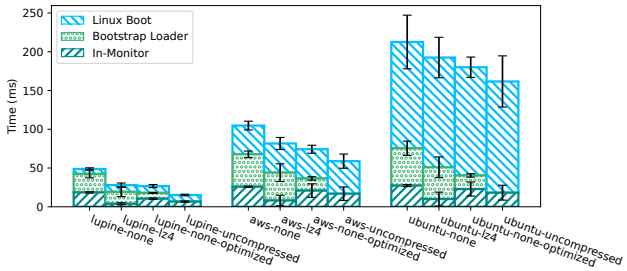


Figure 6. Bootstrap timings comparing LZ4 to compression "None".

[2]The ORC stack unwinder table only needs to be updated if a kernel is configured with CONFIG_UNWINDER_ORC, otherwise it is not present in the ELF.

compression-none-optimized remains slower than a direct boot, which suggests that more overhead lies in the bootstrap loader. This makes a compelling argument for a new design: *in-monitor KASLR*

4 In-monitor KASLR

In this section, we first specify the threat model for KASLR in microVMs, before describing the design and implementation of in-monitor KASLR and FGKASLR.

4.1 Threat model

We distinguish the microVM threat model from that of VMs in traditional infrastructure-as-a-service (IaaS) clouds. In particular, we assume that the attacker is not in control of the guest kernel. Instead, we assume the guest kernel is maintained by the host. Guest workloads are specified in containers that run atop the guest kernel, some of which may have more or less privilege to access other cloud based services. We assume an attacker may compromise a process running in a container, but that such a compromise does not necessarily imply full attacker control of the VM. We discuss unikernel models, in which there is no separation between kernel and application in Section 6.

The guest kernel therefore acts as a first level of a layered defense to maintain isolation between the components that make up a cloud user's application and the host infrastructure. We assume that the guest kernel employs state of the art defenses, including `seccomp`, AppArmor and/or SELinux. We assume the kernel may contain vulnerabilities that allow an attacker to redirect control flow, but new code cannot be injected and run by the attacker (W^X). We similarly assume that SMEP is in use to prevent the attacker from running user code with kernel privileges.

Adding (FG)KASLR to the guest kernel minimizes the capability of the guest workload and adds a layer of defense to protect the host infrastructure from VM escape attacks. While microVMs likely contain less control software (e.g., monitoring, lifecycle, etc.) than IaaS cloud VMs, we suspect that some amount of control software, potentially belonging to the trust domain of the host, may remain inside the guest VM. In this case, (FG)KASLR helps prevent horizontal attacks. We assume that FGKASLR effectively mitigates information leaks that will disclose the location of the kernel offset in the virtual address space. We further assume that all microarchitectural side channels or those that exploit the transient execution domain have been closed.

4.2 Design

Figure 7 provides an overview of the steps used by in-monitor (FG)KASLR and those steps eliminated from the standard (FG)KASLR process in the bootstrap loader (using bzImage format, as described in Section 2.2). Four of the steps appear in both cases: choosing a physical offset, parsing the kernel

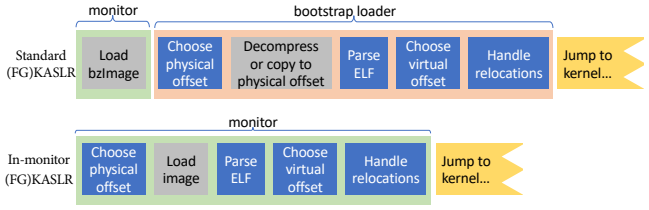


Figure 7. In-monitor KASLR eliminates expensive/redundant bootstrapping.

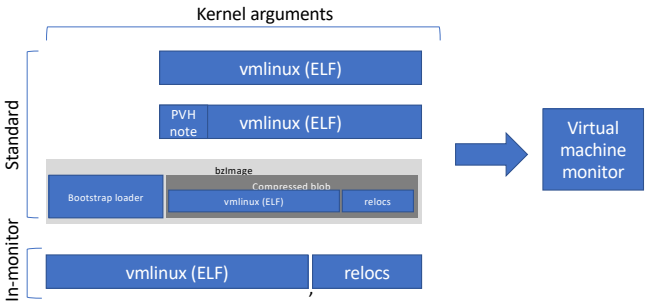


Figure 8. Possible kernel input to microVMs. In-monitor KASLR takes an uncompressed kernel with relocation information.

ELF (and shuffling functions for FGKASLR), choosing a virtual offset, and handling relocations in the virtual address space. In-monitor (FG)KASLR avoids the bootstrap loader costs of relocating the kernel prior to decompression, the decompression itself, and the relocation of the (uncompressed) kernel as done in standard (FG)KASLR.

One of the first tasks of the virtual machine monitor is to read the kernel image from a file on the filesystem into the memory that it has allocated for the guest VM. Once the monitor loads the kernel into guest memory, the process proceeds similarly to the standard bzImage (FG)KASLR case. Instead of the bootstrap loader, the monitor parses the (uncompressed) kernel ELF file, shuffles functions in the case of FGKASLR, chooses a random virtual offset, handles relocations, and updates important tables if FGKASLR is used. Finally, the monitor enters guest context and the guest executes the (uncompressed) kernel image.

In order to perform relocations, the monitor requires some information about where the relevant addresses are located in the kernel image. As described in Section 3, the kernel build process appends this relocation information to the kernel ELF prior to compression. Existing uncompressed guest kernel boot protocols only require the kernel ELF file, not the relocations. After all, they do not perform KASLR or relocation in the virtual address space, so they have no use for such information. Thus, our design for in-monitor KASLR requires an additional parameter to include relocation entries. Figure 8 depicts the additional argument to the monitor.

4.3 Implementation

We implemented in-monitor KASLR and FGKASLR in Firecracker v0.26. Firecracker was selected due to its position as the state of art in terms of reported boot times in production by AWS. Firecracker is open source and implemented in Rust, and our changes are not invasive to the fundamental system. In-monitor KASLR does not exceed 200 lines of code in the guest kernel loading logic, and in-monitor FGKASLR spans around 1000 lines, but the majority of the required logic is contained in its own module. To perform KASLR in the virtual address space, we adapted the implementation of handling relocations directly from the C implementation in the Linux bootstrap loader. Because the computational steps for in-monitor (FG)KASLR are the same as those in the Linux bootstrap loader, the entropy provided by in-monitor randomization is equivalent to that of Linux.

To perform FGKASLR, each major function was also adapted directly from the C implementation in Linux, but some functionality was omitted to help keep boot times down. While investigating FGKASLR in its C implementation, we noticed that updating and sorting `/proc/kallsyms` is unnecessary, because the kernel does not depend on it being correct for a successful boot. From our benchmarks, fixing up `/proc/kallsyms` amounts to 22% of overall boot times on average, so we propose that this fixup can be delayed until `/proc/kallsyms` is first examined. Because microVMs have short lifespans running workloads consisting of a single function, and unprivileged userspace applications should not have knowledge of kernel symbol locations, it is reasonable to assume that `/proc/kallsyms` may never be examined and the cost of updating it can be avoided completely.

Similarly, we decided to omit functionality for updating the ORC stack unwinder table, because the ORC stack unwinder is typically used for debugging and is disabled by default in each of our kernel configurations. Implementing this functionality is feasible if necessary, and in Section 5 we compare our in-monitor implementation to a bootstrap loader with the ORC stack unwinder and kallsyms fixup removed to have a fair apples-to-apples comparison. In Section 6, we discuss the possibility of code reuse between monitor and guest kernels.

As described in Figure 8, we modified the monitor to accept an additional argument to specify relocation entries. As a part of the standard Linux compilation process (and input to bzImage creation), the kernel binary (`vmlinux.bin`) and, when (FG)KASLR is enabled, its relocations (`vmlinux.relocs`) are created. While the monitor is already capable of reading the first, we modified it to accept the second as well. Alternatively, the `relocs` tool in the Linux source tree can take a `vmlinux.bin` as input and generate its respective `vmlinux.relocs` file. With either method, obtaining relocations is straightforward and introducing them to the monitor took the form of an extra configuration option at runtime.

Rather than relying on a complex mix of entropy pools and hardware instructions like `rdrand` to find random numbers, as is done in the standard KASLR, we directly leverage the underlying system's random number supply via a Rust crate, allowing the monitor to pull from an entropy pool on the longer-running host system. Moving randomization to the monitor allows the use of well maintained libraries with higher level languages, providing choice and simplicity in implementation.

To locate a random virtual offset, we mimic the same algorithm as Linux by randomly selecting an appropriately aligned offset between the default kernel load address of 16 MB and the maximum offset permitted for the kernel (1 GB, to avoid the fixmap).

Several numbers are specified by the guest code—and thereby available to the bootstrap loader—but unavailable to the monitor at the time of guest loading. For example, `CONFIG_PHYSICAL_START` and `CONFIG_PHYSICAL_ALIGN` (e.g., the expected load address and alignment in the physical address space that the kernel was compiled for) are both available in the kernel configuration. On the other hand, `__START_KERNEL_MAP` and `KERNEL_IMAGE_SIZE` (e.g., the expected virtual address starting point and the maximum kernel-devoted memory in the virtual address space) are hardcoded in a Linux kernel header file. Our current implementation simply hardcodes these values. In the future, the monitor could possibly take as input the kernel configuration, or these values could be prepended to the kernel binary as an ELF note, making them easy to retrieve.

5 Evaluation

We designed in-monitor KASLR and FGKASLR (which we collectively refer to as *in-monitor randomization*) to allow the guest kernel to benefit from the security they provide, while providing microVM-comparable boot times. Thus, the most important metric to evaluate for in-monitor randomization is boot time. In this section, we answer the following questions:

- Does in-monitor (FG)KASLR enable a state-of-the-art microVM to achieve its boot targets while providing randomization?
- How does the boot time overhead of in-monitor randomization compare with existing self-bootstrapped approaches that can provide randomization?
- How does the amount of memory allocated to the guest impact boot time?
- Does in-monitor randomization cause any overhead beyond boot time?

5.1 Experimental Setup

This section describes the hardware used in our experiments, the Linux version and configurations used, our modifications to the Firecracker VMM, and our testing methodology.

kernel	vmlinux size	bzImage size (Compression None, LZ4)	relocs size	config
lupine-nokaslr	20M	22M, 4.1M	N/A	Lupine
lupine-kaslr	20M	22M, 4.2M	95K	Lupine + KASLR
lupine-fgkaslr	22M	24M, 4.7M	304K	Lupine + FGKASLR
aws-nokaslr	39M	41M, 7.9M	N/A	Firecracker
aws-kaslr	39M	41M, 8.3M	348K	Firecracker + KASLR
aws-fgkaslr	42M	45M, 9.8M	1.1M	Firecracker + FGKASLR
ubuntu-nokaslr	45M	47M, 13M	N/A	Ubuntu 18.04.5
ubuntu-kaslr	45M	48M, 14M	1M	Ubuntu + KASLR
ubuntu-fgkaslr	50M	54M, 17M	2.3M	Ubuntu + FGKASLR

Table 1. Kernels used in Firecracker boot time experiments

Hardware Setup. All the experiments were run on single a machine with an Intel® Core ™ i7-4790 CPU @ 3.60GHz, 8GB of DDR3 memory @ 1600MHz, and an SSD with up to 560mb/s reads and 510mb/s writes. This machine was running Ubuntu 18.04 with the Linux 4.15.0-101-generic kernel.

Kernel configuration. Table 1 summarizes the kernels we used, their image sizes, how much relocation information they contain, and notes on their configuration. KASLR increases the size of the compressed kernel because relocation information is appended to the end of the image before compression, and FGKASLR increases the size of both the compressed and uncompressed, as placing functions into their own sections increases the size of the kernel ELF file, and requires more relocation information than KASLR. All kernels are based on the Linux 5.11.0-rc3 source tree. We chose this version because it was used in the initial FGKASLR implementation[3]. To configure the kernels, the reference configuration (e.g., the `.config` file from the Ubuntu 18.05.4 kernel) was applied to clean Linux 5.11.0-rc3 tree using `make olddefconfig` to generate the new kernel configuration.

As mentioned in Section 2.2, we used three kernel configurations representing a range of kernel sizes. The largest kernel, *Ubuntu*, represents a relatively large, standard distribution kernel used in Ubuntu 18.05.4. The second configuration, *AWS*, uses the reference kernel configuration from AWS Firecracker, representing a state-of-the-art, medium sized, general-purpose microVM kernel. The third configuration, *Lupine*, uses a configuration from Lupine Linux [48], representing a small single-purpose kernel. We only use the base Linux configuration from Lupine, not any kernel patches it provides.

Three variants for each kernel were built. The `nokaslr` version does not have KASLR or FGKASLR enabled, `kaslr` only has KASLR enabled, and the `fgkaslr` version has both KASLR and FGKASLR enabled. The `nokaslr` and `kaslr` kernel variants were built from a source tree without the FGKASLR patches, and `fgkaslr` variants were built from the source tree with the patches. This is due to the fact that even when disabling FGKASLR from the kernel boot command line, additional parsing of the kernel ELF still occurs, increasing boot times.

Firecracker versions. We use four versions of Firecracker in our experiments. The first version, *firecracker-baseline*, is Firecracker version 0.26 built directly from the github source repo without any modifications. Firecracker does not natively support compressed bzImages, so the second version of Firecracker, *firecracker-bzimage*, is a lightly-modified version of v0.26 based on an unmerged patch from a pull request that added bzImage support [4]. The third and forth versions of Firecracker, *firecracker-kaslr* and *firecracker-fgkaslr* respectively, contain our in-monitor implementations of KASLR and FGKASLR.

Testing methodology. To measure the boot time, we used *perf* (Linux profiling with performance counters) which allowed us to retrieve performance information from inside both the monitor and the guest kernel. To timestamp important events, we placed port IO writes inside the guest to be detected by *perf*, and we begin our benchmarking at the call to execute Firecracker, as *perf* will trace syscalls.

We break down the cost of an overall boot into four categories: *In-Monitor*, the time spent in Firecracker, *Bootstrap Setup*, the time spent in the bootstrap loader prior to decompression, *Decompression*, the time to decompress the kernel (only present with LZ4), and *Linux Boot*, the time from the jump to the uncompressed guest kernel's entry point and just after its *init* process is run. The *Linux Boot* portion of each graph does not depend on the method of randomization, (the averages for each kernel with `nokaslr`, `kaslr`, and `fgkaslr` vary by a maximum of 4%), and is included for completeness. The portions of interest are *In-Monitor*, as it will include randomization time for our in-monitor approach, *Bootstrap Setup*, as it includes randomization for a bzImage boot, and *Decompression*, to show the extra time incurred by LZ4.

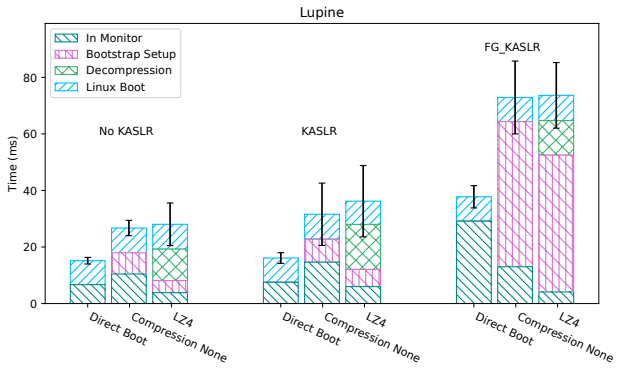
In computing boot times, we take the average boot time over 100 runs. The error bars in the graphs show the min and the max values over the 100 runs. Before each benchmark test we boot the kernel 5 times to warm up the cache, because in a model similar to a production deployment of Firecracker, it is reasonable to assume that kernels will be cached.

5.2 Firecracker Boot Time

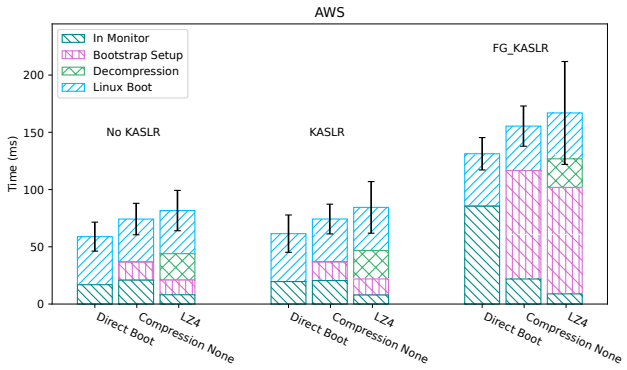
To evaluate the performance of our in-monitor randomization implementation, we compare against the best performance attainable by a bzImage, *optimized compression-none*, which from now on we will simply call *compression-none*. As described in Section 3.3, *compression-none* is our modified version of the bootstrap loader that removes the need for compression and inefficient artifacts to minimize boot times. As in-monitor randomization does not require modifications

[3]https://github.com/kaccardi/linux/tree/fg-kaslr

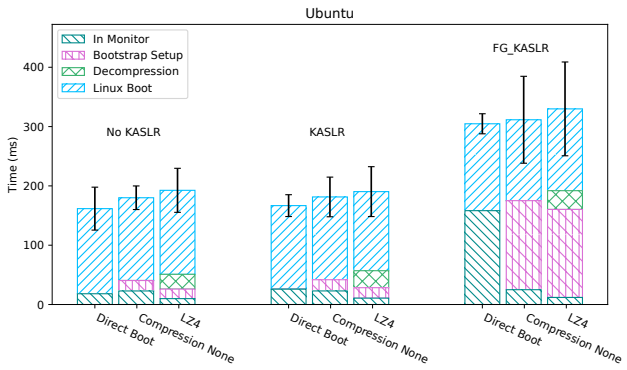
[4]https://github.com/firecracker-microvm/firecracker/pull/670



(a) Lupine kernel config



(b) AWS kernel config



(c) Ubuntu kernel config

Figure 9. Boot times for our three kernel configurations

to the kernel, we also compare against the best performance attainable by an unmodified bzImage bootstrap loader, which is booting an LZ4 kernel.

Figure 9 shows that across all three kernel configurations, in-monitor randomization is faster than both self-randomized approaches of *compression-none* and LZ4 kernels for all three sizes of kernels.

Looking first at KASLR, the *Lupine* kernel with in-monitor KASLR exhibits boot times that are 96% (15 ms) faster than *compression-none* with KASLR on average, *AWS* is 21% (13 ms) faster, and *Ubuntu* is 9% (16 ms) faster. Compared to an unmodified bzImage boot with LZ4, *Lupine* with in-monitor KASLR is 124% (20 ms) faster on average, *AWS* is 38% (23 ms) faster, and *Ubuntu* is 15% (25 ms) faster.

Our in-monitor implementation of FGKASLR exhibits similar trends compared to *compression-none*: *Lupine* is 93% (35 ms) faster, *AWS* is 25% (33 ms) faster, and *Ubuntu* is 2% (6 ms) faster. Leaving the kernel unmodified, in-monitor FGKASLR is 95% faster with *Lupine*, 27% faster with *AWS*, and 9% faster with *Ubuntu*.

Because we can avoid the unnecessary overhead of a bootstrap loader and the computation involved in KASLR is minimal, the cost of in-monitor KASLR compared to an unmodified version of Firecracker is low. With the *Lupine* kernel, in-monitor KASLR increases boot times over *firecracker-baseline* by 1 ms, or 6.3%. For the *AWS* kernel, in-monitor KASLR adds 2 ms, or 3.7%, and *Ubuntu* with in-monitor KASLR adds 3.7 ms, or 2.2%. Overall we show that in-monitor KASLR adds a small cost over an unmodified version of Firecracker, allowing kernels with varying target workloads to enjoy the benefit of the widely adopted KASLR at a low cost.

However, while our in-monitor implementation of FGKASLR is faster then the self-bootstrapped versions, FGKASLR does add significant overhead to the boot process. Compared to *firecracker-baseline*, boot times are 2.33 times slower with *Lupine*, 2.15 times slower with *AWS*, and 1.84 times slower with *Ubuntu*. With its added complexity, as described in Section 3.2, it is not surprising that in-monitor FGKASLR has a higher cost than in-monitor KASLR, but as we have seen it incurs less cost than either of the self-bootstrapped FGKASLR options and still allows *AWS*, Firecracker's baseline microVM configured kernel, to boot within in 131 ms, which is under their 150 ms benchmark metric. Since the added overhead from in-monitor KASLR is low, there will be little effect on critical performance metrics such as the number of VMs instantiated per second. With FGKASLR however, there is a larger tradeoff between an increase in security, and a decrease in throughput. If FGKASLR gains popularity, cloud providers such as AWS may not choose to adopt it, as the increased overhead could be prohibitively large. However, with the *AWS* kernel, FGKASLR manages to hit boot times below their 150ms maximum, so we suspect that FGKASLR could be a welcome increase in security to cloud providers with target use cases that differ from that of services like AWS Lambda.

The variance in the *In-Monitor* portion of *No KASLR* and *KASLR* boots stems from the size of the kernel being read into memory because the cost of KASLR is so low. As seen in Table 1, a *compression-none* bzImage is larger than a bare vmlinux, which is caused by the addition of the bootstrap loader, relocation information, and the padding needed to align the

kernel to the bootstrap loader. Because of this, there is generally an increase in the *In-Monitor* time from uncompressed to compression-none, and the LZ4 kernel, being much smaller, decreases *In-Monitor* time over uncompressed. In all cases, this decrease in *In-Monitor* time is outweighed by *Decompression* , causing LZ4 to be the slowest of the three. In the case of FGKASLR, *In-Monitor* is largest with a direct boot, due to randomization occurring in-monitor and outweighing the time to read the kernel into memory.

The in-monitor implementation of (FG)KASLR shares the same algorithm used by the bootstrap loader, making the core difference which controlling principle is doing the randomization. When it is the responsibility of the monitor to bootstrap, it reads the kernel image one segment at a time directly into guest memory at the physical location specified by each program header. When a bootstrap loader is used, the monitor must read the entire kernel into memory, and only after we have undergone a full read of the kernel can the bootstrap loader parse the ELF and load each segment to its correct location. By booting an uncompressed kernel, we avoid a relocation of the kernel by allowing the monitor to directly load it to its final location. Making the monitor responsible for randomization allows us to benefit from its added security without relying on the bootstrap loader and suffering an unnecessary relocation.

FGKASLR benefits from the same principle, i.e., removing bootstrapping allows us to avoid an extra relocation of the kernel, but avoids more complexity when the system in control is already bootstrapped. In order to copy each function section to a random location, the bootstrap loader must make a copy of the entire kernel text section to avoid overwriting sections that have yet to be randomized in the ELF. This means that the size of the heap given to the bootstrap loader must be increased (up to eight times) over KASLR. Because the bootstrap loader has to initialize its own structures (stack, heap, page table, `bss`), it incurs extra cost when allocating and zeroing a larger boot heap. The monitor avoids this by virtue of the host kernel having already performed these steps. This extra time is incorporated in the *Bootstrap Setup* portion of Figure 9. In all cases, it is clear that *compression-none* does extra work during *Bootstrap Setup* that can be avoided by allowing the monitor to prepare the kernel for boot and handle randomization.

The key takeaway is that bootstrapping a kernel in an already bootstrapped system is unnecessary and adds cost that can be avoided by an in-monitor solution. Moreover, in-monitor randomization requires no modifications to Linux, and self-randomization in an unmodified kernel leaves even more performance on the table, despite using a fast compression scheme. These results also show we can implement in-monitor KASLR with minimal overhead.

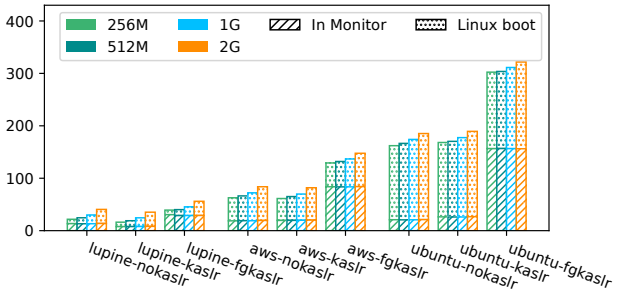


Figure 10. Evaluation of in-monitor randomization with respect to the amount of memory allocated to the guest.

5.3 Guest Memory Impact on Boot Times

We note that boot time is often affected by the amount of memory allocated to the guest VM. We ran an additional experiment to compute the difference between the baseline (without randomization) and in-monitor (FG)KASLR for all three kernels and guest memory amounts of 256 MB (baseline), 512 MB, 1 GB, and 2 GB. Figure 10 shows that the amount of memory allocated to the guest VM does not affect the portion of boot time spent in the monitor. The *Linux Boot* portion increases linearly with allocated memory, yet there is no obvious difference in this time between kernels with and without in-monitor randomization, therefore we conclude that boot time for kernels using in-monitor randomization is not affected by the amount of guest memory.

5.4 Impacts on System Performance

The main reason that microVM guest kernels do not employ KASLR is due to the inability of bootstrap self randomization to achieve microVM boot targets. For completeness, we also measure the effects of enabling randomization on general system performance after boot. Figure 11 shows the results from running the LEBench kernel microbenchmarks [59], which evaluate various performance-critical system calls.

The experimental setup is as follows. For a baseline, we run LEBench on the *aws-nokaslr* kernel with *firecracker-baseline* and run the LEBench program with the default number of iterations (10000). Then, we run LEBench on the *aws-kaslr* and *aws-fgkaslr* kernels with our modified versions of Firecracker that support in-monitor (FG)KASLR. Figure 11 shows the average performance of each test, normalized to the baseline.

We do not expect the Firecracker binary to affect overall guest kernel performance. Instead, we focus on differences in the guest kernels (*aws-nokaslr* vs. *aws-kaslr* and *aws-fgkaslr*). (FG)KASLR-capable guest kernels must be compiled with the CONFIG_RELOCATABLE configuration option. However, unlike PIE, where the code is compiled to be position independent with a performance penalty [57], a relocatable kernel adjusts necessary addresses during bootstrapping, as described in Section 3.2. As shown in Figure 11, results for

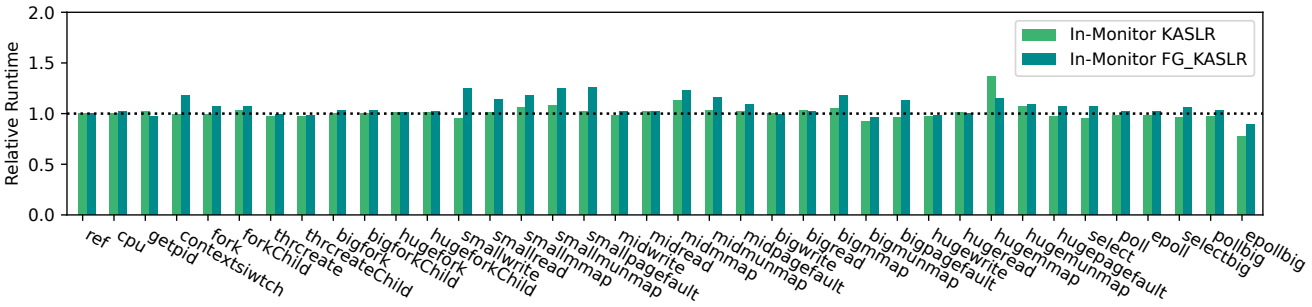


Figure 11. Relative runtime on LEBench kernel microbenchmarks between the baseline *aws-nokaslr* and a KASLR-enabled kernel needed by in-monitor KASLR *aws-kaslr* shows low impact on overall system performance.

the KALSR-enabled kernel are less than 1% slower on average, which is within noise levels. The runtime of FGKASLR-cabable kernels has been seen to be affected by a slightly higher percentage of L1 cache misses, and performance has been observed to vary per-workload due to frequently used functions that are usually grouped together being separated, [5] and our benchmarks show that the performance of in-monitor FGKASLR is about 7% slower than our baseline, which is less than the performance regressions across kernel versions highlighted by the LEBench authors [59].

6 Discussion

Unikernel models. Unikernels exhibit a model in which the guest kernel is not distinct from the guest application; they are linked together in the same address space. While unikernels typically do not yet employ ASLR, performing randomization in the monitor would be more efficient than self-randomization. This mirrors the current design of application ASLR, in which the kernel provides randomization for user-space processes.

More paravirtualization. Paravirtualization, explored first by systems like Xen [21] and Denali [69], fundamentally changes the contract between the virtual machine monitor and the guest kernel. In this way, in-monitor KASLR is an instance of paravirtualization driven by the microVM trend. Other bootstrap steps could be moved to the monitor or removed entirely however, these approaches must weigh the impacts of changing the contract on the host or the guest. For example, in-monitor randomization requires relocs to be managed with guest binaries. Other approaches, like unikernel monitors [70] require changes or adaptation in the guest. Our experience illustrates the importance of both host and guest.

Reusing code modules in monitor or guest. Though we used Rust instead of C to better fit into Firecracker, our implementation of in-monitor randomization shared algorithms—if not code—with the existing implementations in the bootstrap loader. While KASLR remains relatively simple, FGKASLR is more complex. Ideally, both implementations could share code. If the relevant (FG)KASLR modules were implemented as `rustvmm` crates, it is even possible that the Linux bootstrap loader could be rewritten in Rust to seamlessly reuse the same implementation. Such an undertaking could be feasible as the bootstrap loader is already logically separate from the rest of Linux, and interest is growing for in-tree Linux modules written in Rust [23].

Memory density despite KASLR. Cloud providers have a business incentive to maximize the utilization of their resources. One resource dimension is memory density: the host can transparently share pages between VMs using content-based page merging [8, 39].[6] However, fine-grained randomization has been shown to nullify page-sharing benefits [66] as fine-grained variations of page contents prevent merging. With in-monitor randomization, the host could manage this tradeoff. For example, the host could select a particular random seed for a group of related VMs [65].

Trust models that do not include the host. In our threat model, we assume a cloud environment in which the host, including the VM monitor, is trusted. However, increasing interest in confidential computing platforms, such as Intel SGX or AMD SEV, suggest a different threat model, one in which the host is not trusted. If the (untrusted) host is compromised by or colluding with an attacker, in-monitor KASLR is trivially broken, since random offsets are generated by the host. In this environment, some form of self

[5] https://lwn.net/Articles/824307/

[6] Due to easy-to-use side channels like flush and reload [72] that are enabled with page sharing, it is unclear how much providers use page sharing across tenants in practice [22], though a case could be made for same-tenant page merging.

randomization [29] (e.g., in-guest KASLR) and its associated bootstrapping/copying costs may be required.

6.1 Validity of kernel image caching.

As we detail in Section 2, it makes sense for modern hypervisors to have moved away from a compressed kernel boot if we can assume that the (uncompressed) kernel image is able to be warm in the cache. However, if guest kernels become more specialized and unikernel-like [48], more kernels may exist in the system and this assumption may begin to fail. In this case (depending on storage size/performance), compression may once again become attractive.

7 Related Work

One of the major drivers of lightweight virtualization is serverless computing. SAND [19] proposes two level fault isolation for serverless functions, where VMs persist for multiple functions or function instances that belong to the same application. This alleviates some of the pressure for VMs to boot quickly but, without in-monitor KASLR, loses the opportunity to periodically efficiently reboot kernels for re-randomization between function invocations. Many other projects have investigated caches of memory snapshots of execution contexts to reduce startup times.

So called *zygote*-based approaches leverage checkpoint/restore to avoid redundant startup delays and have been explored in the context of the JVM [67], lightweight containers [54], unikernels [26], and VMs [25, 32]. However, zygote-based approaches typically rely on page sharing and copy-on-write to maintain reasonably low memory overhead, which can result in identical memory layouts, nullifying ASLR. Morula suggests maintaining a pool of zygotes with different memory layouts to combat this [49]. The complexity and security issues of doing so may be offset by fast-booting VMs that support randomization via in-monitor KASLR, reducing the need for zygotes.

Related projects in the unikernel space, especially *unikernel monitors* [70] apply similar in-monitor techniques to achieve fast boot of unikernels. For example, `ukvm` sets up page tables before jumping to the kernel. In the most extreme case, for some unikernels, all bootstrapping can be eliminated, resulting in unikernels running as processes instead of VMs [71]. While we are not aware of ASLR-enabled unikernels, unikernels may present additional opportunities for whole-system ASLR [50] and they are apparently considering in-monitor approaches.[7] Regardless, our proposal of in-monitor KASLR is more general as it supports to unmodified Linux-based guest VMs.

Finally, there is a rich literature exploring ASLR in userspace, including various granularities and binary formats [20, 43, 47, 56, 68]. It is possible that some of these approaches could be applied to guest kernels via in-monitor KASLR.

[7] https://github.com/Solo5/solo5/issues/304

8 Conclusion

In this paper, we have identified an incompatibility between the fast-booting requirements of microVMs in the cloud and the bootstrap self-randomization technique that OS kernels use to achieve kernel address space layout randomization (KASLR). Fortunately, we have shown that, through a technique we refer to as in-monitor KASLR, unmodified Linux guest kernels can utilize KASLR while booting with little overhead on a state of the art monitor. Furthermore, it is possible that performing randomization in the monitor may reduce constraints and create opportunities for greater randomization in the future. More generally, as the cloud and virtualization community continues to move towards lighter-weight VMs, we believe it is time to re-evaluate what the guest should do itself and what should be left to the monitor.

References

[1] [n. d.]. Apache OpenWhisk: Open Source Serverless Cloud Platform. http://openwhisk.apache.org/. (Accessed on 2021-01-04).

[2] [n. d.]. AWS Lambda. https://aws.amazon.com/lambda/. (Accessed on 2016-03-04).

[3] [n. d.]. Azure Functions Serverless Compute. https://azure.microsoft.com/en-us/services/functions/. (Accessed on 2021-01-04).

[4] [n. d.]. Docker. http://docs.docker.io/en/latest/.

[5] [n. d.]. IBM Cloud Functions. https://www.ibm.com/cloud/functions. (Accessed on 2021-01-04).

[6] [n. d.]. Intel NEMU: Modern Hypervisor for the Cloud. https://github.com/intel/nemu.

[7] [n. d.]. Kata Containers: The speed of containers, the security of VMs. https://katacontainers.io/. (Accessed on 2021-01-04).

[8] [n. d.]. Kernel Samepage Merging. https://www.linux-kvm.org/page/KSM.

[9] [n. d.]. LING. http://erlangonxen.org.

[10] [n. d.]. Linux Containers. https://linuxcontainers.org/.

[11] 2015. Clive: Removing (most of) the software stack from the cloud. http://lsub.org/ls/clive.html.

[12] 2015. JavaScript library operating system for the cloud. http://runtimejs.org/.

[13] 2015. The Rumprun unikernel and toolchain for various platforms. https://github.com/rumpkernel/rumprun.

[14] 2020. pvh. https://xenbits.xen.org/docs/unstable/misc/pvh.html.

[15] 2020. THE LINUX/x86 BOOT PROTOCOL. https://www.kernel.org/doc/Documentation/x86/boot.rst.

[16] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proc. of ACM CCS*. Alexandria, VA.

[17] Kristen Carlson Accardi. 2020. Function Granular KASLR. https://lkml.org/lkml/2020/7/17/947.

[18] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. of USENIX NSDI*. Santa Clara, CA.

[19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proc. of USENIX Annual Technical Conf.* Boston, MA.

[20] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proc. of USENIX Security*. San Diego, CA.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen

and the Art of Virtualization. In *Proc. of ACM SOSP*. Bolton Landing, NY.

[22] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud. In *Proc. of USENIX WOOT*. Washington, D.C.

[23] John Baublitz, Nick Desaulniers, Alex Gaynor, Geoffrey Thomas, Josh Triplett, and Miguel Ojeda. 2020. Barriers to in-tree Rust. In *Linux Plumbers Conference*. Virtual Conference.

[24] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proc. of IEEE CloudCom*. Vancouver,Canada.

[25] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, H. Andrés Lagar-Cavilla, and Eyal de Lara. 2011. Kaleidoscope: Cloud Micro-Elasticity via VM State Coloring. In *Proc. of ACM EuroSys*. Salzburg, Austria.

[26] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proc. of ACM EuroSys*. Heraklion, Greece.

[27] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proc. of ACM CCS*. London, United Kingdom.

[28] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break it, Fix it, Repeat. In *Proc. of ACM ASIA CCS*. Taipei, Taiwan.

[29] 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.

[30] Jonathan Corbet. 2011. Kernel address randomization. https://lwn.net/Articles/444503/.

[31] Lizzie Dixon. 2017. Breaking KASLR with perf. https://blog.lizzie.io/kaslr-and-perf.html.

[32] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting *(ASPLOS '20)*. 467–481.

[33] Jake Edge. 2013. Randomizing the kernel. https://lwn.net/Articles/546686/.

[34] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proc of IEEE/ACM MICRO*. Taipei, Taiwan.

[35] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *Proc. of RAID*. Atlanta, GA.

[36] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX Security*. Bellevue, WA.

[37] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Proc. of ESSoS*. Bonn, Germany.

[38] Daniel Gruss, Clémentine Maurice, Andreas Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proc. of ACM CCS*. Vienna, Austria.

[39] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proc. of USENIX OSDI*. San Diego, CA.

[40] Baoquan He. 2015. randomize kernel physical address and virtual address separately. https://lwn.net/Articles/635901/.

[41] R. Hund, C. Willems, and T. Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proc. of IEEE Security and Privacy*. San Francisco, CA.

[42] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proc. of ACM CCS*. Vienna, Austria.

[43] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proc. of ACSAC*. Miami Beach, FL.

[44] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.

[45] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: optimizing the operating system for virtual machines. In *Proc. of USENIX Annual Technical Conf.* Philadelphia, PA.

[46] Amit Klein and Benny Pinkas. 2019. From IP ID to Device ID and KASLR Bypass. In *Proc. of USENIX Security*. Santa Clara, CA.

[47] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the Gadgets: Binary-Level Code Randomization Using Instruction Displacement. In *Proc. of ACM ASIA CCS*. Xi'an, China.

[48] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proc. of ACM EuroSys*. Heraklion, Greece.

[49] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Proc. of IEEE Security and Privacy*. San Jose, CA.

[50] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proc. of ACM ASPLOS*. Houston, TX.

[51] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proc. of ACM SOSP*. Shanghai, China.

[52] Ming Mao and Marty Humphrey. 2012. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. 423–430. https://doi.org/10.1109/CLOUD.2012.103

[53] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proc. of USENIX NSDI*. Seattle, WA.

[54] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proc. of USENIX Annual Technical Conf.* Boston, MA.

[55] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-compatible Unikernel. In *Proc. of ACM VEE* (Providence, RI).

[56] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proc. of IEEE Security and Privacy*. San Francisco, CA.

[57] Mathias Payer. 2012. *Too much PIE is bad for performance.* Technical Report 766. ETH Zurich, Zurich, Switzerland.

[58] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *Proc. of USENIX Security*. Austin, TX.

[59] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance

Evolution of Linux's Core Operations. In *Proc. of ACM SOSP*. Huntsville, Ontario, Canada.

[60] Dan Rosenberg. 2010. kptr_restrict for hiding kernel pointers. https://lwn.net/Articles/420403/.

[61] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proc. of ACM CCS*. Alexandria, VA.

[62] Klaus Stengel, Florian Schmaus, and Rüdiger Kapitza. 2013. EsseOS: Haskell-based Tailored Services for the Cloud. In *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware* (Beijing, China) *(ARM '13)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2541583.2541587

[63] The PaX team. 2013. KASLR: An Exercise in Cargo Cult Security. https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security.

[64] Arjan van de Ven. 2015. An introduction to Clear Containers. https://lwn.net/Articles/644675/.

[65] Fernando Vano-Garcia and Hector Marco-Gisbert. 2020. KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant cloud systems. *J. Parallel and Distrib. Comput.* 137 (2020), 77 – 90.

[66] F. Vañó-García and H. Marco-Gisbert. 2018. How Kernel Randomization is Canceling Memory Deduplication in Cloud Computing Systems. In *Proc. of IEEE NCA"*. Cambridge, MA.

[67] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proc. of ACM EuroSys*. Dresden, Germany.

[68] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code. In *Proc. of ACM CCS*. Raleigh, North Carolina, USA.

[69] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. In *Proc. of USENIX OSDI*. Boston, MA.

[70] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *Proc. of USENIX HotCloud*. Denver, CO.

[71] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels As Processes. In *Proc. of ACM SoCC*. Carlsbad, CA.

[72] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. of USENIX Security*. San Diego, CA.

A Artifact Appendix

Abstract

The artifact provided with this paper comprises a benchmarking suite to evaluate the performance of booting guest kernels with Firecracker VMM modified to support in-monitor (FG)KASLR, as well as the data/scripts used to generate figures used in the paper. We leverage *perf* (Linux profiling with performance counters), and small patches to the Linux kernel to issue I/O writes to a unique port that are traced as KVM events by *perf*[8]. Benchmarking begins when Firecracker is executed, timestamps are taken before and after relevant function calls/code blocks (e.g., decompression, (FG)KASLR functionality, loading kernel segments, etc.), and the final timestamp is taken after the call to execute the guest's `init` process.

[8]The idea to use *perf* to trace I/O writes was found here: https://github.com/stefano-garzarella/qemu-boot-time

A.1 Description & Requirements

A.1.1 How to access. Artifacts can be accessed via: https://github.com/bencw12/in-monitor-rando-benchmarking

A.1.2 Hardware dependencies. Firecracker requires either Intel x86_64, or AMD x86_64, CPUs that offer hardware virtualization support. All experiments for the paper were run on a machine with an Intel Core i7-4790 CPU @ 3.60 GHz.

A.1.3 Software dependencies. Currently, Firecracker recommends either Linux kernel version 4.14 or 5.10, as those are the versions they currently use to validate source code. We ran all experiments on a machine running Ubuntu 18.04 using a Linux 4.15 kernel.

A.1.4 Benchmarks. All guest kernels, file systems, and relocation information needed to boot VMs with and without our modifications to Firecracker are included in the artifact repository. The data collected for our experiments is in the `results-paper` directory, with subdirectories containing the results for each experiment, and the included scripts will generate the graphs shown in the paper.

A.2 Set-up

Firecracker requires KVM access which can be granted with: `sudo setfacl -m u:$USER:rw /dev/kvm`. All scripts are designed to be run from a standard Linux shell with root permissions with no additional set-up.

A.3 Evaluation workflow

A.3.1 Major Claims.

- (C1): When kernels are not warm in the cache, a compressed `bzImage` achieves optimal performance due to the image being smaller than an uncompressed image, but when kernels are cached, the increase in I/O time to load an uncompressed kernel over that of a `bzImage` is small compared to the overhead incurred by the `bzImage`'s bootstrap loader. This is shown in the experiment (E2) described in Section 2.2 with results shown in Figure 4.
- (C2): The majority of the extra overhead from a `bzImage` bootstrap loader stems from decompression, which is why microVMs have moved toward directly booting uncompressed kernels. The data supporting this is also generated from (E2), and results are shown in Figure 5.
- (C3): Optimizing the `bzImage` bootstrap loader to remove decompression and redundant kernel relocations still leaves performance on the table and does not justify booting a `bzImage` over an uncompressed kernel. This experiment (E3) is described in Section 3.3 with results shown in Figure 6.
- (C4): In-monitor randomization achieves up 22% to better performance than existing/optimized methods

of self-randomization where a bootstrap loader, rather than the monitor, is the controlling principle. On average, in-monitor KASLR adds a small overhead of 4% (2ms) compared to stock Firecracker. This is shown in the experiment (E4) described in Section 5.2. Results are illustrated in Figure 9.
- (C5): In-monitor randomization does not affect kernel performance outside of boot. The experiments (E5) described in Section 5.4 verify this and results are shown in Figure 10.

A.3.2 Experiments. All kernels, file systems, relocation information, and binaries are included with our artifacts, so all experiments except for (E5) can be run by executing one shell script from the root of the repository with no additional preparation. All guest kernels are Linux version 5.11, since this is the version FG-KASLR was originally patched into. Each VM is allocated 256M and 1 CPU, and the cache is warmed by booting each kernel 5 times before recording data unless otherwise specified. Each experiment finishes all 100 boots of a kernel before moving on to the next. All new data is saved in a directory separate from the data used in the paper, and will be used instead of our results by graph generation scripts if present.

Experiment (E1): *Compression Bakeoff* [1.5 compute-hours]: A comparison of overall boot times for `bzImages` compressed with six different compression schemes supported by Linux.

[Execution] Executing `run_compression_bakeoff.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/compression-bakeoff/` for each kernel during execution. To use the new data to generate a graph like Figure 3, run `scripts/fig-3.py`. LZ4 is expected to have the lowest overhead.

Experiment (E2): *Cache-Effects* [1 compute-hour]: An experiment used to demonstrate the effects of caching on overall boot time when booting a `bzImage` versus an uncompressed kernel.

[Execution] Executing `run_cache_effects.sh 100` will boot each kernel 100 times to replicate the results used in the paper. First each kernel is allowed to be warm in the cache, then each kernel is run after dropping the caches (pagecache, dentries, and inodes) to see the affect of a cold cache on boot performance.

[Results] Results are collected and saved to the directory `results/cache-effects/` automatically for each kernel during execution. The results from this experiment are used to generate Figures 4 and 5. To use the new data to generate them, run `scripts/fig-4.py` and `scripts/fig-5.py`. Figure 4 is expected to show that `bzImages` will have faster boot times than uncompressed kernels when the cache is cold, but uncompressed kernels boot faster than `bzImages` when they can be cached. Figure 5 is expected to show that decompression makes up the majority of bootstrapping time.

Experiment (E3): *Bootstrap Method Comparison* [1 compute-hour]: A comparison of four methods of bootstrapping Linux: *none, lz4, none-optimized,* and *uncompressed. none* kernels are patched to simply leave the kernel uncompressed when linking into a `bzImage`, *lz4* is an unmodified `bzImage` using LZ4 compression, *none-optimized* kernels remove decompression and extra relocations, and *uncompressed* is the uncompressed kernel natively supported by Firecracker.

[Execution] Executing `run_bootstrap_comparison.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/bootstrap-comparison/` for each kernel during execution. To use the new data to generate a graph like Figure 6, run `scripts/fig-6.py`. *none* kernels are expected to have the highest overhead, followed by *lz4*, *none-optimized*, and *uncompressed* with the lowest overhead.

Experiment (E4): *Evaluation* [2.5 compute-hours]: This experiment evaluated the performance of in-monitor (FG)KASLR by comparing in-monitor randomization with uncompressed kernels to self-randomization methods using *none-optimized* and LZ4. Each kenrel is also compared against its unrandomized counterpart as a baseline.

[Execution] Executing `run_eval.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/evaluation/` for each kernel during execution. To use the new data to generate a graph like Figure 9, run `scripts/fig-9.py`. In-monitor randomization with uncompressed kernels is expected to have the lowest overhead compared to kernels with *none-optimized* and LZ4. Firecracker with in-monitor KASLR is expected to exhibit minimal overhead compared to stock Firecracker.

Experiment (E5): *LEBench* [5 human-minutes, 75 compute-minutes]: This experiment uses `LEBench`[9] to evaluate the performance of important kernel functions for an unrandomized kernel, and kernels with (FG)KASLR.

[9] https://github.com/LinuxPerfStudy/LEBench

[Execution] Executing `run_lebench.sh` will boot an unrandomized kernel (`nokaslr`), a kernel with KASLR (`kaslr`), and a kernel with FG-KASLR (`fgkaslr`). At each boot, the LEBench process runs and the kernel will shutdown when it is finished.

[Results] Results are collected and saved automatically to the directory `results/lebench/` after LEBench finishes for each kernel. To use the new data to generate a graph like Figure 10, run `scripts/fig-10.py`. The performance of kernels with in-monitor (FG)KASLR for each kernel function is not expected to deviate significantly from the baseline of `nokaslr`.

A.4 Notes on Reusability

The methods we used to benchmark the performance of the Linux bootstrap process can be extended to any part of the kernel by defining more tracepoints and placing I/O writes in the kernel code.