Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization

Cristiano Giuffrida
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Anton Kuijsten
Vrije Universiteit, Amsterdam
akuijst@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

In recent years, the deployment of many application-level countermeasures against memory errors and the increasing number of vulnerabilities discovered in the kernel has fostered a renewed interest in kernel-level exploitation. Unfortunately, no comprehensive and well-established mechanism exists to protect the operating system from arbitrary attacks, due to the relatively new development of the area and the challenges involved.

In this paper, we propose the first design for fine-grained address space randomization (ASR) inside the operating system (OS), providing an efficient and comprehensive countermeasure against classic and emerging attacks, such as return-oriented programming. To motivate our design, we investigate the differences with application-level ASR and find that some of the well-established assumptions in existing solutions are no longer valid inside the OS; above all, perhaps, that information leakage becomes a major concern in the new context. We show that our ASR strategy outperforms state-of-the-art solutions in terms of both performance and security without affecting the software distribution model. Finally, we present the first comprehensive *live re-randomization* strategy, which we found to be particularly important inside the OS. Experimental results demonstrate that our techniques yield low run-time performance overhead (less than 5% on average on both SPEC and syscall-intensive benchmarks) and limited run-time memory footprint increase (around 15% during the execution of our benchmarks). We believe our techniques can greatly enhance the level of OS security without compromising the performance and reliability of the OS.

1 Introduction

Kernel-level exploitation is becoming increasingly popular among attackers, with local and remote exploits surfacing for Windows [5], Linux [2], Mac OS X [3], BSD variants [37, 4], and embedded operating systems [25]. This emerging trend stems from a number of important factors. First, the deployment of defense mechanisms for user programs has made application-level exploitation more challenging. Second, the kernel codebase is complex, large, and in continuous evolution, with many new vulnerabilities inevitably introduced over time. Studies on the Linux kernel have shown that its codebase has more than doubled with a steady fault rate over the past 10 years [55] and that many known but potentially critical bugs are at times left unpatched indefinitely [29]. Third, the number of targets in large-scale attacks is significant, with a plethora of internet-connected machines running the same kernel version independently of the particular applications deployed. Finally, an attacker has generally more opportunities inside the OS, for example the ability to disable in-kernel defense mechanisms or the option to execute shellcode at the user level (similar to classic application-level attacks) or at the kernel level (approach taken by kernel *rootkits*).

Unfortunately, existing OS-level countermeasures fail to provide a comprehensive defense mechanism against generic memory error exploits. A number of techniques aim to thwart code injection attacks [65, 28, 60], but are alone insufficient to prevent *return-into-kernel-text* attacks [56] and *return-oriented programming* (ROP) in general [35]. Other approaches protect kernel hooks or generally aim at preserving control-flow integrity [69, 74, 44, 57]. Unfortunately, this does not prevent attackers from tampering with noncontrol data, which may lead to privilege escalation or allow other attacks. In addition, most of these techniques incur high overhead and require virtualization support, thus increasing the size of the trusted computing base (TCB).

In this paper, we explore the benefits of address space randomization (ASR) inside the operating system and present the first comprehensive design to defend against classic and emerging OS-level attacks. ASR is a well-established defense mechanism to protect user programs

against memory error exploits [12, 39, 14, 72, 73]; all the major operating systems include some support for it at the application level [1, 68]. Unfortunately, the OS itself is typically not randomized at all. Recent Windows releases are of exception, as they at least randomize the base address of the text segment [56]. This randomization strategy, however, is wholly insufficient to counter many sophisticated classes of attacks (e.g., noncontrol data attacks) and is extremely vulnerable to information leakage, as better detailed later. To date, no strategy has been proposed for comprehensive and fine-grained OS-level ASR. Our effort lays the ground work to fill the gap between application-level ASR and ASR inside the OS, identifying the key requirements in the new context and proposing effective solutions to the challenges involved.

Contributions. The contributions of this paper are threefold. First, we identify the challenges and the key requirements for a comprehensive OS-level ASR solution. We show that a number of assumptions in existing solutions are no longer valid inside the OS, due to the more constrained environment and the different attack models. Second, we present the first design for fine-grained ASR for operating systems. Our approach addresses all the challenges considered and improves existing ASR solutions in terms of both performance and security, especially in light of emerging ROP-based attacks. In addition, we consider the application of our design to component-based OS architectures, presenting a fully fledged prototype system and discussing real-world applications of our ASR technique. Finally, we present the first generic *live rerandomization* strategy, particularly central in our design. Unlike existing techniques, our strategy is based on run-time state migration and can transparently rerandomize arbitrary code and data with no state loss. In addition, our rerandomization code runs completely sandboxed. Any run-time error at rerandomization time simply results in restoring normal execution without endangering the reliability of the OS.

2 Background

The goal of address space randomization is to ensure that code and data locations are unpredictable in memory, thus preventing attackers from making precise assumptions on the memory layout. To this end, fine-grained ASR implementations [14, 39, 72] permute the order of individual memory objects, making both their addresses and their relative positioning unpredictable. This strategy attempts to counter several classes of attacks.

Attacks on code pointers. The goal of these attacks is to override a function pointer or the return address on the stack with attacker-controlled data and subvert control flow. Common memory errors that can directly allow these attacks are buffer overflows, format bugs, use-after-free, and uninitialized reads. In the first two cases, the attack requires assumptions on the relative distance between two memory objects (e.g., a vulnerable buffer and a target object) to locate the code pointer correctly. In the other cases, the attack requires assumptions on the relative alignment between two memory objects in case of memory reuse. For example, use-after-free attacks require control over the memory allocator to induce the allocation of an object in the same location of a freed object still pointed by a vulnerable dangling pointer. Similarly, attacks based on stack/heap uninitialized reads require predictable allocation strategies to reuse attacker-controlled data from a previously deallocated object. All these attacks also rely on the absolute location of the code the attacker wants to execute, in order to adjust the value of the code pointer correctly. In detail, code injection attacks rely on the location of attacker-injected shellcode. Attacks using *return-into-libc* strategies [22] rely on the location of a particular function—or multiple functions in case of chained *return-into-libc* attacks [52]. More generic attacks based on *return-oriented programming* [66] rely on the exact location of a number of *gadgets* statically extracted from the program binary.

Attacks on data pointers. These attacks commonly exploit one of the memory errors detailed above to override the value of a data pointer and perform an arbitrary memory read/write. Arbitrary memory reads are often used to steal sensitive data or information on the memory layout. Arbitrary memory writes can also be used to override particular memory locations and indirectly mount other attacks (e.g., control-flow attacks). Attacks on data pointers require the same assumptions detailed for code pointers, except the attacker needs to locate the address of some data (instead of code) in memory.

Attacks on nonpointer data. Attacks in this category target noncontrol data containing sensitive information (e.g., `uid`). These attacks can be induced by an arbitrary memory write or commonly originate from buffer overflows, format bugs, integer overflows, signedness bugs, and use-after-free memory errors. While unable to directly subvert control flow, they can often lead to privilege escalation or indirectly allow other classes of attacks. For example, an attacker may be able to perform an arbitrary memory write by corrupting an array index which is later used to store attacker-controlled data. In contrast to all the classes of attacks presented earlier, nonpointer data attacks only require assumptions on the relative distance or alignment between memory objects.

3 Challenges in OS-level ASR

This section investigates the key challenges in OS-level address space randomization, analyzing the differences with application-level ASR and reconsidering some of

the well-established assumptions in existing solutions. We consider the following key issues in our analysis.

W⊕X. A number of ASR implementations complement their design with W⊕X protection [68]. The idea is to prevent code injection attacks by ensuring that no memory page is ever writable and executable at the same time. Studies on the Linux kernel [45], however, have shown that enforcing the same property for kernel pages introduces implementation issues and potential sources of overhead. In addition, protecting kernel pages in a combined user/kernel address space design does not prevent an attacker from placing shellcode in an attacker-controlled application and redirecting execution there. Alternatively, the attacker may inject code into W∧X regions with double mappings that operating systems share with user programs (e.g., `vsyscall` page on Linux) [56].

Instrumentation. Fine-grained ASR techniques typically rely on code instrumentation to implement a comprehensive randomization strategy. For example, Bhaktar et al. [14] heavily instrument the program to create self-randomizing binaries that completely rearrange their memory layout at load time. While complex instrumentation strategies have been proven practical for application-level solutions, their applicability to OS-level ASR raises a number of important concerns. First, heavyweight instrumentation may introduce significant run-time overhead which is ill-affordable for the OS. Second, these load-time ASR strategies are hardly sustainable, given the limited operations they would be able to perform and the delay they would introduce in the boot process. Finally, complex instrumentation may introduce a lot of untrusted code executed with no restriction at runtime, thus endangering the reliability of the OS or even opening up new opportunities for attack.

Run-time constraints. There are a number of constraints that significantly affect the design of an OS-level ASR solution. First, making strong assumptions on the memory layout at load time simplifies the boot process. This means that some parts of the operating system may be particularly hard to randomize. In addition, existing rerandomization techniques are unsuitable for operating systems. They all assume a stateless model in which a program can gracefully exit and restart with a fresh rerandomized layout. Loss of critical state is not an option for an OS and neither is a full reboot, which introduces unacceptable downtime and loss of all the running processes. Luckily, similar restrictions also apply to an adversary determined to attack the system. Unlike application-level attacks, an exploit needs to explicitly recover any critical memory object corrupted during the attack or the system will immediately crash after successful exploitation.

Attack model. Kernel-level exploitation allows for a powerful attack model. Both remote and local attacks are possible, although local attacks mounted from a compromised or attacker-controlled application are more common. In addition, many known attack strategies become significantly more effective inside the OS. For example, noncontrol data attacks are more appealing given the amount of sensitive data available. In addition, ROP-based control-flow attacks can benefit from the large codebase and easily find all the necessary gadgets to perform arbitrary computations, as demonstrated in [35]. This means that disclosing information on the locations of "useful" text fragments can drastically increase the odds of successful ROP-based attacks. Finally, the particular context opens up more attack opportunities than those detailed in Section 2. First, unchecked pointer dereferences with user-provided data—a common vulnerability in kernel development [18]—can become a vector of arbitrary kernel memory reads/writes with no assumption on the location of the original pointer. Second, the combined user/kernel address space design used in most operating systems may allow an attacker controlling a user program to directly leverage known application code or data for the attack. The conclusion is that making both the relative positioning between *any* two memory objects and the location of individual objects unpredictable becomes much more critical inside the OS.

Information leakage. Prior work on ASR has often dismissed information leakage attacks—in which the attacker is able to acquire information about the internal memory layout and carry out an exploit in spite of ASR—as relatively rare for user applications [14, 67, 72]. Unfortunately, the situation is completely different inside the OS. First, there are several possible entry points and a larger leakage surface than user applications. For instance, a recent study has shown that uninitialized data leading to information leakage is the most common vulnerability in the Linux kernel [18]. In addition, the common combined user/kernel address space design allows arbitrary memory writes to easily become a vector of information leakage for attacker-controlled applications. To make things worse, modern operating systems often disclose sensitive information to unprivileged applications voluntarily, in an attempt to simplify deployment and debugging. An example is the `/proc` file system, which has already been used in several attacks that exploit the exposed information in conventional [56] and nonconventional [76] ways. For instance, the `/proc` implementation on Linux discloses details on kernel symbols (i.e., `/proc/kallsyms`) and slab-level memory information (i.e., `/proc/slabinfo`). To compensate for the greater chances of information leakage, ASR at the finest level of granularity possible and continuous rerandomization become both crucial to minimize the knowledge acquired by an attacker while probing the system.

Brute forcing. Prior work has shown that many existing application-level ASR solutions are vulnerable to

simple brute-force attacks due to the low randomization entropy of shared libraries [67]. The attack presented in [67] exploits the crash recovery capabilities of the Apache web server and simply reissues the same return-into-libc attack with a newly guessed address after every crash. Unlike many long-running user applications, crash recovery cannot be normally taken for granted inside the OS. An OS crash is normally fatal and immediately hinders the attack while prompting the attention of the system administrator. Even assuming some crash recovery mechanism inside the OS [43, 27], brute-force attacks need to be far less aggressive to remain unnoticed. In addition, compared to remote clients hiding their identity and mounting a brute-force attack against a server application, the source of an OS crash can be usually tracked down. In this context, blacklisting the offensive endpoint/request becomes a realistic option.

4 A design for OS-level ASR

Our fine-grained ASR design requires confining different OS subsystems into isolated event-driven components. This strategy is advantageous for a number of reasons. First, this enables selective randomization and rerandomization for individual subsystems. This is important to fully control the randomization and rerandomization process with per-component ASR policies. For example, it should be possible to retune the rerandomization frequency of *only* the virtual filesystem after noticing a performance impact under particular workloads. Second, the event-driven nature of the OS components greatly simplifies synchronization and state management at rerandomization time. Finally, direct intercomponent control transfer can be more easily prevented, thus limiting the freedom of a control-flow attack and reducing the number of potential ROP gadgets by design.

Our ASR design is currently implemented by a microkernel-based OS architecture running on top of the MINIX 3 microkernel [32]. The OS components are confined in independent hardware-isolated processes. Hardware isolation is beneficial to overcome the problems of a combined user/kernel address space design introduced earlier and limit the options of an attacker. In addition, the MMU-based protection can be used to completely sandbox the execution of the untrusted rerandomization code. Our ASR design, however, is not bound to its current implementation and has more general applicability.

For example, our ASR design can be directly applied to other component-based OS architectures, including microkernel-based architectures used in common embedded OSes—such as L4 [41], Green Hills Integrity [7], and QNX [33]—and research operating systems using software-based component isolation schemes—such as Singularity [36]. Commodity operating systems, in contrast, are traditionally based on monolithic architectures and lack well-defined component boundaries. While this does not prevent adoption of our randomization technique, it does eliminate the ability to selectively rerandomize specific parts of the OS, yielding poorer flexibility and longer rerandomization times to perform whole-OS state migration. Encouragingly, there is an emerging trend towards allowing important commodity OS subsystems to run as isolated user-space processes, including filesystems [6] and user-mode drivers in Windows [50] or Linux [16]. Our end-to-end design can be used to protect all these subsystems as well as other operating system services from several classes of attacks. Note that, while running in user space, operating system services are typically trusted by the kernel and allowed to perform a variety of critical system operations. An example is `udev`, the device manager for the Linux kernel, which has already been target of several different exploits [17]. Finally, given the appropriate run-time support, our design could also be used to improve existing application-level ASR techniques and offer better protection against memory error exploits for generic user-space programs.

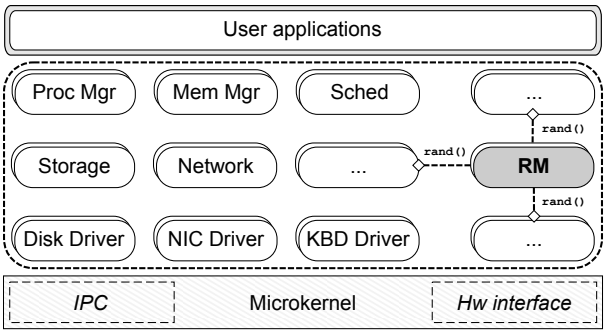

Figure 1: The OS architecture for our ASR design.

Figure 1 shows the OS architecture implementing our ASR design. At the heart lies the microkernel, providing only IPC functionalities and low-level resource management. All the other core subsystems are confined into isolated OS processes, including drivers, memory management, process management, scheduling, storage and network stack. In our design, all the OS processes (and the microkernel) are randomized using a link-time transformation implemented with the LLVM compiler framework [42]. The transformation operates on prelinked LLVM bitcode to avoid any lengthy recompilation process at runtime. Our link-time strategy avoids the need for fine-grained load-time ASR, eliminating delays in the boot process and the run-time overhead introduced by the indirection mechanisms adopted [14]. In addition, this strategy reduces the instrumentation complexity to the bare minimum, with negligible amount of untrusted code exposed to the runtime. The vast majority of our ASR

transformations are statically verified by LLVM at the bitcode level. As a result, our approach is also safer than prior ASR solutions relying on binary rewriting [39].

As pointed out in [14], load-time ASR has a clear advantage over alternative strategies: the ability to create self-randomizing binaries distributed to every user in identical copies, thus preserving today's software distribution model. Fortunately, our novel live rerandomization strategy can fully address this concern. In our model, every user receives the same (unrandomized) binary version of the OS, as well as the prelinked LLVM bitcode of each OS component. The bitcode files are stored in a protected disk partition inaccessible to regular user programs, where a background process periodically creates new randomized variants of the OS components using our link-time ASR transformation (and any valid LLVM backend to generate the final binary). The generated variants are consumed by the *randomization manager* (RM), a special component that periodically rerandomizes every OS process (including itself). Unlike all the existing solutions, rerandomization is applied transparently online, with no system reboot or downtime required. The conclusion is that we can directly leverage our live rerandomization technique to randomize the original OS binary distributed to the user. This strategy retains the advantages of link-time ASR without affecting the software distribution model.

When the OS boots up for the first time, a full rerandomization round is performed to relinquish any unrandomized code and data present in the original binary. To avoid slowing down the first boot process, an option is to perform the rerandomization lazily, for example replacing one OS process at the time at regular time intervals. After the first round, we continuously perform live rerandomization of individual OS components in the background. Currently, the microkernel is the only piece of the OS that does not support live rerandomization. Rerandomization can only be performed after a full reboot, with a different variant loaded every time. While it is possible to extend our current implementation to support live rerandomization for the microkernel, we believe this should be hardly a concern. Microkernel implementations are typically in the order of 10kLOC, a vastly smaller TCB than most hypervisors used for security enforcement, as well as a candidate for formal verification, as demonstrated in prior work [40].

Our live rerandomization strategy for an OS process, in turn, is based on run-time state migration, with the entire execution state transparently transferred to the new randomized process variant. The untrusted rerandomization code runs completely sandboxed in the new variant and, in case of run-time errors, the old variant immediately resumes execution with no disruption of service or state loss. To support live migration, we rely on another LLVM link-time transformation to embed relocation and type information into the final process binary. This information is exposed to the runtime to accurately introspect the state of the two variants and migrate all the randomized memory objects in a layout-independent way.

5 ASR transformations

The goal of our link-time ASR transformation is to randomize all the code and data for every OS component. Our link-time strategy minimizes the time to produce new randomized OS variants on the deployment platform and automatically provides randomization for the program and all the statically linked libraries. Our transformation design is based on five key principles: (i) minimal performance impact; (ii) minimal amount of untrusted code exposed to the runtime; (iii) architecture-independence; (iv) no restriction on compiler optimizations; (v) maximum randomization granularity possible. The first two principles are particularly critical for the OS, as discussed earlier. Architecture-independence enhances portability and eliminates the need for complex binary rewriting techniques. The fourth principle dictates compiler-friendly strategies, for example avoiding indirection mechanisms used in prior solutions [12], which inhibit a number of standard optimizations (e.g., inlining). Eliminating the need for indirection mechanisms is also important for debuggability reasons. Our transformations are all debug-friendly, as they do not significantly change the code representation—only allocation sites are transformed to support live rerandomization, as detailed later—and preserve the consistency of symbol table and stack information. Finally, the last principle is crucial to provide lower predictability and better security than existing techniques.

Traditional ASR techniques [1, 68, 12] focus on randomizing the base address of code and data regions. This strategy is ineffective against all the attacks that make assumptions only about relative distances/alignments between memory objects, is prone to brute forcing [67], and is extremely vulnerable to information leakage. For instance, many examples of application-level information leakage have emerged on Linux over the years, and experience shows that, even by acquiring minimal knowledge on the memory layout, an attacker can completely bypass these basic ASR techniques [24].

To overcome these limitations, second-generation ASR techniques [14, 39, 72] propose fine-grained strategies to permute individual memory objects and randomize their relative distances/alignments. While certainly an improvement over prior techniques, these strategies are still vulnerable to information leakage, raising serious concerns on their applicability at the OS level. Unlike traditional ASR techniques, these strategies make it

normally impossible for an attacker to make strong assumptions on the locations of arbitrary memory objects after learning the location of a single object. They are completely ineffective, however, in inhibiting precise assumptions on the layout of the leaked object itself. This is a serious concern inside the OS, where information leakage is the norm rather than the exception.

To address all the challenges presented, our ASR transformation is implemented by an LLVM link-time pass which supports fine-grained randomization of both the relative distance/alignment between *any* two memory objects and the *internal layout* of individual memory objects. We now present our transformations in detail and draw comparisons with state-of-the-art techniques.

Code randomization. The code-transformation pass performs three primary tasks. First, it enforces a random permutation of all the program functions. In LLVM, this is possible by shuffling the symbol table in the intended order and setting the appropriate linkage to preserve the permutation at code generation time. Second, it introduces (configurable) random-sized padding before the first function and between any two functions in the bitcode, making the layout even more unpredictable. To generate the padding, we create dummy functions with a random number of instructions and add them to the symbol table in the intended position. Thanks to demand paging, even very large padding sizes do not significantly increase the run-time physical memory usage. Finally, unlike existing ASR solutions, we randomize the internal layout of every function.

To randomize the function layout, an option is to permute the basic blocks and the instructions in the function. This strategy, however, would hinder important compiler optimizations like branch alignment [75] and optimal instruction scheduling [49]. Nonoptimal placements can result in poor instruction cache utilization and inadequate instruction pipelining, potentially introducing significant run-time overhead. To address this challenge, our pass performs *basic block shifting*, injecting a dummy basic block with a random number of instructions at the top of every function. The block is never executed at runtime and simply skipped over, at the cost of only one additional jump instruction. Note that the order of the original instructions and basic blocks is left untouched, with no noticeable impact on run-time performance. The offset of every instruction with respect to the address of the function entry point is, however, no longer predictable.

This strategy is crucial to limit the power of an attacker in face of information leakage. Suppose the attacker acquires knowledge on the absolute location of a number of kernel functions (e.g., using `/proc/kallsyms`). While return-into-kernel-text attacks for these functions are still conceivable (assuming the attacker can subvert control flow), arbitrary ROP-based computations are structurally prevented, since the location of individual gadgets is no longer predictable. While the dummy basic block is in a predictable location, it is sufficient to cherrypick its instructions to avoid giving rise to any new useful gadget. It is easy to show that a sequence of `nop` instructions does not yield any useful gadget on the x86 [54], but other strategies may be necessary on other architectures.

Static data randomization. The data-transformation pass randomly permutes all the static variables and read-only data on the symbol table, as done before for functions. We also employ the same padding strategy, except random-sized dummy variables are used for the padding. Buffer variables are also separated from other variables to limit the power of buffer overflows. In addition, unlike existing ASR solutions, we randomize the internal layout of static data, when possible.

All the aggregate types in the C programming language are potential candidates for layout randomization. In practice, there are a number of restrictions. First, the order of the elements in an array cannot be easily randomized without changing large portions of the code and resorting to complex program analysis techniques that would still fail in the general case. Even when possible, the transformation would require indirection tables that translate many sequential accesses into random array accesses, sensibly changing the run-time cache behavior and introducing overhead. Second, `unions` are currently not supported natively by LLVM and randomizing their layout would introduce unnecessary complications, given their rare occurrence in critical system data structures and their inherent ambiguity that already weakens the assumptions made by an attacker. Finally, `packed structs` cannot be randomized, since the code makes explicit assumptions on their internal layout.

In light of these observations, our transformation focuses on randomizing the layout of regular `struct` types, which are pervasively used in critical system data structures. The layout randomization permutes the order of the `struct` members and adds random-sized padding between them. To support all the low-level programming idioms allowed by C, the type transformations are operated uniformly for all the static and dynamic objects of the same `struct` type. To deal with code which treats nonpacked `structs` as implicit `unions` through pointer casting, our transformation pass can be instructed to detect unsafe pointer accesses and refrain from randomizing the corresponding `struct` types.

Layout randomization of system data structures is important for two reasons. First, it makes the relative distance/alignment between two `struct` members unpredictable. For example, an overflow in a buffer allocated inside a `struct` cannot make precise assumptions about which other members will be corrupted by the overflow. Second, this strategy is crucial to limit the assumptions

of an attacker in face of information leakage. Suppose an attacker is armed with a reliable arbitrary kernel memory write generated by a missing pointer check. If the attacker acquires knowledge on the location of the data structure holding user credentials (e.g., `struct cred` on Linux) for an attacker-controlled unprivileged process, the offset of the `uid` member is normally sufficient to surgically override the user ID and escalate privileges. All the existing ASR solutions fail to thwart this attack. In contrast, our layout randomization hinders any precise assumptions on the final location of the `uid`. While brute forcing is still possible, this strategy will likely compromise other data structures and trigger a system crash.

Stack randomization. The stack randomization pass performs two primary tasks. First, it randomizes the base address of the stack to make the absolute location of any stack object unpredictable. In LLVM, this can be accomplished by creating a dummy `alloca` instruction—which allocates memory on the stack frame of the currently executing function—at the beginning of the program, which is later expanded by the code generator. This strategy provides a portable and efficient mechanism to introduce random-sized padding for the initial stack placement. Second, the pass randomizes the relative distance/alignment between any two objects allocated on the stack. Prior ASR solutions have either ignored this issue [39, 72] or relied on a shadow stack and dynamically generated random padding [14], which introduces high run-time overhead (10% in the worst case in their experiments for user applications).

To overcome these limitations, our approach is completely static, resulting in good performance and code which is statically verified by LLVM. In addition, this strategy makes it realistic to use cryptographically random number generators (e.g., `/dev/random`) instead of pseudo-random generators to generate the padding. While care should be taken not to exhaust the randomness pool used by other user programs, this approach yields much stronger security guarantees than pseudo-random generators, like recent attacks on ASR demonstrate [24]. Our transformations can be configured to use cryptographically random number generators for code, data, and stack instrumentation, while, similar to prior approaches [14], we always resort to pseudo-random generation in the other cases for efficiency reasons.

When adopting a static stack padding strategy, great care should be taken not to degrade the quality of the randomization and the resulting security guarantees. To randomize the relative distances between the objects in a stack frame, we permute all the `alloca` instructions used to allocate local variables (and function parameters). The layout of every stack-allocated `struct` is also randomized as described earlier. Nonbuffer variables are all grouped and pushed to the top of the frame, close

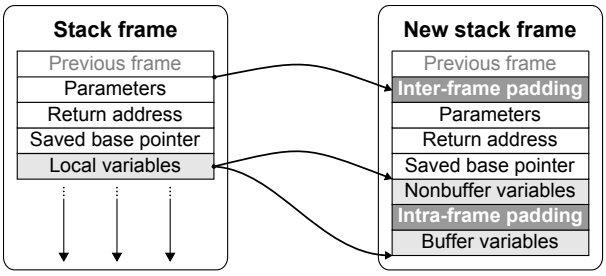

Figure 2: The transformed stack layout.

to the base pointer and the return address. Buffer variables, in turn, are pushed to the bottom, with random-sized padding (i.e., dummy `alloca` instructions) added before and between them. This strategy matches our requirements while allowing the code generator to emit a maximally efficient function prologue.

To randomize the relative alignment between any two stack frame allocations of the same function (and thus the relative alignment between their objects), we create random-sized padding before every function call. Albeit static, this strategy faithfully emulates dynamically generated padding, given the level of unpredictability introduced across different function calls. Function calls inside loops are an exception and need to be handled separately. Loop unrolling is a possible solution, but enforcing this optimization in the general case may be expensive. Our approach is instead to precompute N random numbers for each loop, and cycle through them before each function call. Figure 2 shows the randomized stack layout generated by our transformation.

Dynamic data randomization. Our operating system provides `malloc/mmap`-like abstractions to every OS process. Ideally, we would like to create memory allocation wrappers to accomplish the following tasks for both heap and memory-mapped regions: (i) add random-sized padding before the first allocated object; (ii) add random-sized padding between objects; (iii) permute the order of the objects. For memory-mapped regions, all these strategies are possible and can be implemented efficiently [39]. We simply need to intercept all the new allocations and randomly place them in any available location in the address space. The only restriction is for fixed OS component-specific virtual memory mappings, which cannot be randomized and need to be explicitly reserved at initialization time.

For heap allocations, we instrument the code to randomize the heap base address and introduce random-sized padding at allocation time. Permuting heap objects, however, is normally impractical in standard allocation schemes. While other schemes are possible—for example, the slab allocator in our memory manager randomizes block allocations within a slab page—state-of-

the-art allocators that enforce a fully and globally randomized heap organization incur high overhead (117% worst-case performance penalty) [53]. This limitation is particularly unfortunate for kernel *Heap Feng Shui* attacks [25], which aim to carefully drive the allocator into a deterministic exploitation-friendly state. While random interobject padding makes these attacks more difficult, it is possible for an attacker to rely on more aggressive exploitation strategies (i.e., heap spraying [59]) in this context. Suppose an attacker can drive the allocator into a state with a very large unallocated gap followed by only two allocated buffers, with the latter vulnerable to underflow. Despite the padding, the attacker can induce a large underflow to override all the traversed memory locations with the same target value. Unlike stack-based overflows, this strategy could lead to successful exploitation without the attacker worrying about corrupting other critical data structures and crashing the system. Unlike prior ASR solutions, however, our design can mitigate these attacks by periodically rerandomizing every OS process and enforcing a new unpredictable heap permutation. We also randomize (and rerandomize) the layout of all the dynamically allocated `structs`, as discussed earlier.

Kernel modules randomization. Traditional loadable kernel module designs share many similarities—and drawbacks, from a security standpoint—with application-level shared libraries. The attack presented in [61] shows that the data structures used for dynamic linking are a major source of information leakage and can be easily exploited to bypass any form of randomization for shared libraries. Prior work on ASR [67, 14] discusses the difficulties of reconciling sharing with fine-grained randomization. Unfortunately, the inability to perform fine-grained randomization on shared libraries opens up opportunities for attacks, including probing, brute forcing [67], and partial pointer overwrites [23].

To overcome these limitations, our design allows only statically linked libraries for OS components and inhibits any form of dynamic linking inside the operating system. Note that this requirement does by no means limit the use of loadable modules, which our design simply isolates in independent OS processes following the same distribution and deployment model of the core operating system. This approach enables sharing and lazy loading/unloading of individual modules with no restriction, while allowing our rerandomization strategy to randomize (and rerandomize) every module in a fine-grained manner. In addition, the process-based isolation prevents direct control-flow and data-flow transfer between a particular module and the rest of the OS (i.e., the access is always IPC- or capability-mediated). Finally, this strategy can be used to limit the power of untrusted loadable kernel modules, an idea also explored in prior work on commodity operating systems [16].

6 Live rerandomization

Our live rerandomization design is based on novel automated run-time migration of the execution state between two OS process variants. The variants share the same operational semantics but have arbitrarily different memory layouts. To migrate the state from one variant to the other at runtime, we need a way to remap all the corresponding global state objects. Our approach is to transform the bitcode with another LLVM link-time pass, which embeds metadata information into the binary and makes run-time state introspection and automated migration possible.

Metadata transformation. The goal of our pass is to record metadata describing all the static state objects in the program and instrument the code to create metadata for dynamic state objects at runtime. Access to these objects at the bitcode level is granted by the LLVM API. In particular, the pass creates static metadata nodes for all the static variables, read-only data, and functions whose address is taken. Each metadata node contains three key pieces of information: node ID, relocation information, and type. The node ID provides a layout-independent mechanism to map corresponding metadata nodes across different variants. This is necessary because we randomize the order and the location of the metadata nodes (and write-protect them) to hinder new opportunities for attacks. The relocation information, in turn, is used by our run-time migration component to locate every state object in a particular variant correctly. Finally, the type is used to introspect any given state object and migrate the contained elements (e.g., pointers) correctly at runtime.

To create a metadata node for every dynamic state object, our pass instruments all the memory allocation and deallocation function calls. The node is stored before the allocated data, with canaries to protect the in-band metadata against buffer overflows. All the dynamic metadata nodes are stored in a singly-linked list, with each node containing relocation information, allocation flags, and a pointer to an allocation descriptor. Allocation flags define the nature of a particular allocation (e.g., heap) to reallocate memory in the new variant correctly at migration time. The allocation descriptors, in turn, are statically created by the pass for all the allocation sites in the program. A descriptor contains a site ID and a type. Similar to the node ID, the site ID provides a layout-independent mechanism to map corresponding allocation descriptors (also randomized and write-protected) across different variants. The type, in contrast, is determined via static analysis and used to correctly identify the run-time type of the allocated object (e.g., a `char` type with an allocation of 7 bytes results in a `[7 x char]` run-time type). Our static analysis can automatically identify the type for all the standard memory allocators and custom allocators that use simple allocation wrappers. More

advanced custom allocation schemes, e.g., region-based memory allocators [11], require instructing the pass to locate the proper allocation wrappers correctly.

The rerandomization process. Our OS processes follow a typical event-driven model based on message passing. At startup, each process initializes its state and immediately jumps to the top of a long-running event-processing loop, waiting for IPC messages to serve. Each message can be processed in cooperation with other OS processes or the microkernel. The message dispatcher, isolated in a static library linked to every OS process, can transparently intercept two special system messages sent by the *randomization manager* (RM): *sync* and *init*. These messages cannot be spoofed by other processes because the IPC is mediated by the microkernel.

The rerandomization process starts with RM loading a new variant in memory, in cooperation with the microkernel. Subsequently, it sends a sync message to the designated OS process, which causes the current variant to immediately block in a well-defined execution point. A carefully selected synchronization point (e.g., in `main`) eliminates the need to instrument transient stack regions to migrate additional state, thus reducing the run-time overhead and simplifying the rerandomization strategy. The new variant is then allowed to run and delivered an init message with detailed instructions. The purpose of the init message is to discriminate between fresh start and rerandomization init. In the latter scenario, the message contains a capability created by the microkernel, allowing the new variant to read arbitrary data and metadata from the old variant. The capability is attached to the IPC endpoint of the designated OS process and can thus only be consumed by the new variant, which by design inherits the old variant's endpoint. This is crucial to transparently rerandomize individual operating system processes without exposing the change to the rest of the system.

When the rerandomization init message is intercepted, the message dispatcher requests the run-time migration component to initialize the new variant properly and then jumps to the top of the event-processing loop to resume execution. This preserves the original control flow semantics and transparently restores the correct execution state. The migration component is isolated in a library and runs completely sandboxed in the new variant. RM monitors the execution for run-time errors (i.e., panics, crashes, timeouts). When an error is detected, the new variant is immediately cleaned up, while the old variant is given back control to resume execution normally. When the migration completes correctly, in contrast, the old variant is cleaned up, while the new variant resumes execution with a rerandomized memory layout. We have also implemented rerandomization for RM itself, which only required some extra microkernel changes to detect run-time errors and arbitrate control transfer between the two variants. Our run-time error detection mechanism allows for safe rerandomization without trusting the (complex) migration code. Moreover, the reversibility of the rerandomization process makes detecting semantic errors in the migration code a viable option. For example, one could transparently migrate the state from one variant to another, migrate it again to another instance of the original variant, and then compare the results. Figure 3 depicts the proposed rerandomization process.

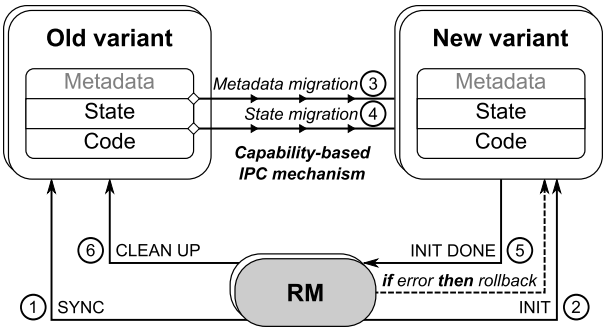

Figure 3: The rerandomization process.

State migration. The migration starts by transferring all the metadata from the old variant to a local cache in the new variant. Our capability-based design allows the migration code to locate a root metadata descriptor in the old variant and recursively copy all the metadata nodes and allocation descriptors to the new variant. To automate the metadata transfer, all the data structures copied use a fixed and predetermined layout. At the end, both the old and the new metadata are available locally, allowing the code to arbitrarily introspect the state of the two variants correctly. To automate the data transfer, we map every old metadata node in the local cache with its counterpart in the new variant. This is done by pairing nodes by ID and carefully reallocating every old dynamic state object in the new variant. Reallocations are performed in random order, thus enforcing a new unpredictable permutation of heap and memory-mapped regions. An interesting side effect of the reallocation process is the compaction of all the live heap objects, an operation that reduces heap fragmentation over time. Our reallocation strategy is indeed inspired by the way a compacting garbage collector operates [70].

The mapping phase generates all the perfect pairs of state objects in the two variants, ready for data migration. Note that paired state objects may not reflect the same type or size, due to the internal layout rerandomization. To transfer the data, the migration code introspects every state object in the old variant by walking its type recursively and examining each inner state element found. Nonpointer elements are simply transferred by value, while pointer elements require a more careful transfer strategy. To deal with layout randomization,

each recursive step requires mapping the current state element to its counterpart (and location) in the new variant. This can be easily accomplished because the old type and the new type have isomorphic structures and only differ in terms of member offsets for randomized `struct` types. For example, to transfer a `struct` variable with 3 primitive members, the migration code walks the original `struct` type to locate all the members, computes their offsets in the two variants, and recursively transfers the corresponding data in the correct location.

Pointer migration. The C programming language allows several programming constructs that make pointer migration particularly challenging in the general case. Our approach is to fully automate migration of all the common cases and only delegate the undecidable cases to the programmer. The first case to consider is a pointer to a valid static or dynamic state object. When the pointer points to the beginning of the object, we simply reinitialize the pointer with the address of the pointed object in the new variant. Interior pointers (i.e., pointers into the middle of an object) in face of internal layout rerandomization require a more sophisticated strategy. Similar to our introspection strategy, we walk the type of the pointed object and recursively remap the offset of the target element to its counterpart. This strategy is resilient to arbitrary layout rerandomization and makes it easy to reinitialize the pointer in the new variant correctly.

Another scenario of interest is a pointer which is assigned a special integer value (e.g., `NULL` or `MAP_FAILED` (-1)). Our migration code can explicitly recognize special ranges and transfer the corresponding pointers by value. Currently, all the addresses in reserved memory ranges (e.g., zero pages) are marked as special values.

In another direction, memory addresses or other layout-specific information may be occasionally stored in integer variables. This is, unfortunately, a case of unsolvable ambiguity which cannot be automatically settled without programmer assistance. To this end, we support annotations to mark "hidden" pointers in the code.

Pointers stored in `unions` are another case of unsolvable ambiguity. Since C does not support tagged `unions`, it is impossible to resolve these cases automatically. In our experiments with OS code, `unions` with pointers were the only case of ambiguity that required manual intervention. Other cases are, however, possible. For example, any form of pointer encoding or obfuscation [13] would require knowledge on the particular encoding to migrate pointers correctly. Other classes of pointers—guard pointers, uninitialized pointers, dangling pointers—are instead automatically handled in our implementation. In the last two cases, the general strategy is to try to transfer the pointer as a regular pointer, and simply reinitialize it to `NULL` in the new variant whenever our dynamic pointer analysis reports an error.

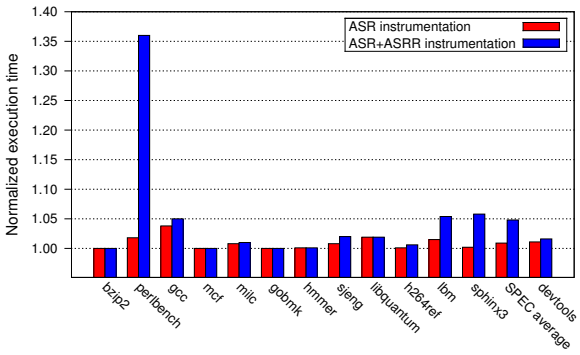

Figure 4: Execution time of the SPEC CPU 2600 benchmarks and our *devtools* benchmark normalized against the baseline (no OS/benchmark instrumentation).

7 Evaluation

We have implemented our ASR design on the MINIX 3 microkernel-based operating system [32], which already guarantees process-based isolation for all the core operating system components. The OS is x86-based and exposes a complete POSIX interface to user applications. We have heavily modified and redesigned the original OS to implement support for our ASR techniques for all the possible OS processes. The resulting operating system comprises a total of 20 OS processes (7 drivers and 13 servers), including process management, memory management, storage and network stack. Subsequently, we have applied our ASR transformations to the system and evaluated the resulting solution.

7.1 Performance

To evaluate the performance of our ASR technique, we ported the C programs in the SPEC CPU 2006 benchmark suite to our prototype system. We also put together a *devtools* macrobenchmark, which emulates a typical syscall-intensive workload with the following operations performed on the OS source tree: compilation, `find`, `grep`, copying, and deleting. We performed repeated experiments on a workstation equipped with a 12-core 1.9Ghz AMD Opteron "Magny-Cours" processor and 4GB of RAM, and averaged the results. All the OS code and our benchmarks were compiled using `Clang/LLVM` 2.8 with `-O2` optimization level. To thoroughly stress the system and identify all the possible bottlenecks, we instrumented *both* the OS and the benchmarks using the same transformation in each run. The default padding strategy used in the experiments extends the memory occupancy of every state object or `struct` member by 0-30%, similar to the default values suggested in [14]. Figure 4 depicts the resulting execution times.

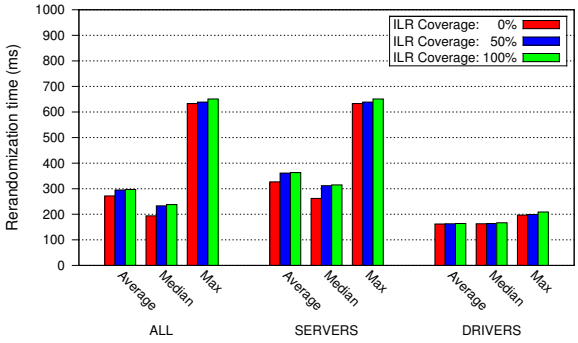

Figure 5: Rerandomization time against coverage of internal layout rerandomization.

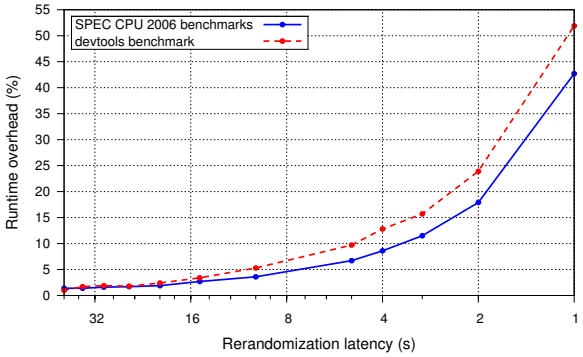

Figure 6: Run-time overhead against periodic rerandomization latency.

The ASR instrumentation alone introduces 0.9% run-time overhead on average on SPEC benchmarks and 1.1% on *devtools*. The average run-time overhead increases to 4.8% and 1.6% respectively with ASRR instrumentation. The maximum overhead reported across all the benchmarks was found for `perlbench` (36% ASRR overhead). Profiling revealed this was caused by a massive amount of dynamic memory allocations. This test case pinpoints a potential source of overhead introduced by our technique, which, similar to prior approaches, relies on memory allocation wrappers to instrument dynamically allocated objects. Unlike prior comprehensive solutions, however, our run-time overhead is barely noticeable on average (1.9% for ASRR without `perlbench`). The most comprehensive second-generation technique presented in [14]—which, compared to other techniques, also provides fine-grained stack randomization—introduces a run-time overhead of 11% on average and 23% in the worst case, even by instrumenting *only* the test programs. The main reasons for the much higher overheads are the use of heavyweight stack instrumentation and indirection mechanisms that inhibit compiler optimizations and introduce additional pointer dereferences for every access to code and data objects. Their stack instrumentation, however, includes a shadow stack implementation that could complement our techniques to offer stronger protection against stack spraying attacks.

Although we have not observed strong variations in our macrobenchmark performance across different runs, our randomization technique can potentially affect the original spatial locality and yield nonoptimal cache usage at runtime. The possible performance impact introduced—inherent in all the fine-grained ASR techniques—is subject to the particular compiler and system adopted and should be carefully evaluated in each particular deployment scenario.

Figure 5 shows the rerandomization time (average, median, max) measured across all the OS components. With no internal layout rerandomization (ILR), a generic component completes the rerandomization process in 272ms on average. A higher ILR coverage increases the average rerandomization time only slightly (297ms at 100% coverage). The impact is more noticeable for OS servers than drivers, due to the higher concentration of complex rerandomized `structs` (and pointers to them) that need to be remapped during migration. Albeit satisfactory, we believe these times can be further reduced, for example using indexes to speed up our dynamic pointer analysis. Unfortunately, we cannot compare our current results against existing solutions, given that no other live rerandomization strategy exists to date.

Finally, Figure 6 shows the impact of periodic rerandomization on the execution time of SPEC and *devtools*. The experiment was performed by rerandomizing a single OS component at the end of every predetermined time interval. To ensure uniform coverage, the OS components were all rerandomized in a round-robin fashion. Figure 6 reports a barely noticeable overhead for rerandomization latencies higher than 20s. For lower latencies, the overhead increases steadily, reaching the value of 42.7% for SPEC and 51.9% for *devtools* at 1s. The rerandomization latency defines a clear tradeoff between performance and unobservability of the system. Reasonable choices of the rerandomization latencies introduce no performance impact and leave a small window with a stable view of the system to the attacker. In some cases, a performance penalty may also be affordable to offer extra protection in face of particularly untrusted components.

7.2 Memory usage

Table 1 shows the average run-time virtual memory overhead introduced by our technique inside the OS during the execution of our benchmarks. The overhead measured is comparable to the space overhead we observed

Type	Overhead
ASRR state	16.1%
ASRR overall	14.6%
ASR padding$_a$	$((8a_s+2a_h+4a_f)\cdot 10^{-4}+c_{base})\%$
ASR padding$_r$	$((2r_s+0.6r_h+3r_f)\cdot 10^{-1}+c_{base})\%$

Table 1: Average run-time virtual memory overhead measured during the execution of our benchmarks.

for the OS binaries on the disk. In the table, we report the virtual memory overhead to also account for dynamic state object overhead at runtime. For the average OS component, support for rerandomization introduces 16.1% state overhead (the extra memory necessary to store state metadata w.r.t. the original memory occupancy of all the static and dynamic static objects) and 14.6% overall memory overhead (the extra memory necessary to store state metadata and migration code w.r.t. the original memory footprint) on average. The virtual memory overhead (not directly translated to physical memory overhead, as noted earlier) introduced by our randomization strategy is only due to padding. Table 1 reports the overhead for two padding schemes using byte granularity (but others are possible): (i) padding$_a$, generating an inter-object padding of a bytes, with a uniformly distributed in $[0;a_{s,h,f}]$ for static, heap, and function objects, respectively; (ii) padding$_r$, generating an inter-object padding of $r\cdot s$ bytes, with a preceding object of size s, and r uniformly distributed in $[0;r_{s,h,f}]$ for static, heap, and function objects, respectively. The coefficient c_{base} is the overhead introduced by the one-time padding used to randomize the base addresses. The formulations presented here omit stack frame padding, which does not introduce persistent memory overhead.

7.3 Effectiveness

As pointed out in [14], an analytical analysis is more general and effective than empirical evaluation in measuring the effectiveness of ASR. Bhaktar et al. [14] present an excellent analysis on the probability of exploitation for different vulnerability classes. Their entropy analysis applies also to other second-generation ASR techniques, and, similarly, to our technique, which, however, provides additional entropy thanks to internal layout randomization and live rerandomization. Their analysis, however, is mostly useful in evaluating the effectiveness of ASR techniques against guessing and brute-force attacks. As discussed earlier, these attacks are far less attractive inside the operating system. In contrast, information leakage dominates the scene.

For this reason, we explore another direction in our analysis, answering the question: "*How much information does the attacker need to acquire for successful exploitation?*". In this respect, Table 2 compares our ASR technique (ASR_3) with first-generation techniques like PaX [68] and comprehensive second-generation techniques like the one presented in [14]. Most attacks require at least some knowledge of a memory area to corrupt and of another target area to achieve the intended effect (missing kernel pointer checks and non control data attacks are examples of exceptions in the two cases). Table 2 shows that first-generation techniques only require the attacker to learn the address of a memory region (e.g., stack) to locate the target correctly. Second-generation techniques, in turn, allow the attacker to corrupt the right memory location by learning the relative distance/alignment between two memory objects.

	$\mathbf{ASR_1}$	$\mathbf{ASR_2}$	$\mathbf{ASR_3}$
Vulnerability			
Buffer overflows	A_r	R_o	R_e
Format string bugs	A_r	R_o	R_e
Use-after-free	A_r	R_o	R_e
Uninitialized reads	A_r	R_o	R_e
Effect			
Arbitrary memory R|W	A_r	A_o	A_e
Controlled code injection	A_r	A_o	A_e
Return-into-libc/text	A_r	$N\cdot A_o$	$N\cdot A_o$
Return-oriented programming	A_r	$N\cdot A_o$	-

A_r = Known region address
A_o = Known object address
A_e = Known element address
R_o = Known relative distance/alignment between objects
R_e = Known relative distance/alignment between elements

Table 2: Comparison of ASR techniques.

In this respect, our internal layout randomization provides better protection, forcing the attacker to learn the relative distance/alignment between two memory elements in the general case. For example, if the attacker learns the relative alignment between two heap-allocated data structures S_1 and S_2 and wants to exploit a vulnerable dangling pointer to hijack a write intended for a member of S_1 to a member of S_2, he still needs to acquire information on the relative positioning of the members.

Similarly, our technique constraints attacks based on arbitrary memory reads/writes to learn the address of the target element. In contrast, second-generation techniques only require knowledge of the target memory object. This is easier to acquire, because individual objects can be exposed by symbol information (e.g., `/proc/kallsyms`) and are generally more likely to have their address taken (and leaked) than interior elements. Controlled code injection shows similar differences—spraying attacks are normally "A_r", in contrast. Return-

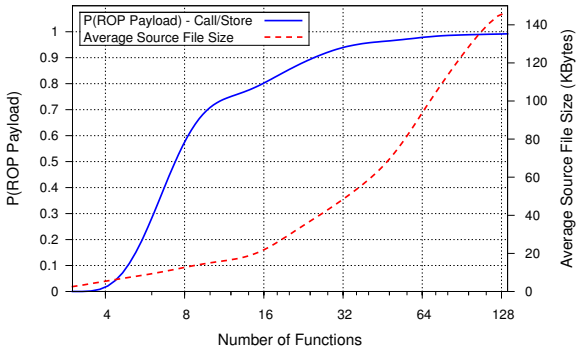

Figure 7: The probability that state-of-the-art techniques [64] can successfully generate ROP payloads to call linked functions or perform attacker-controlled arbitrary memory writes. The (fitted) distribution is plotted against the number of known functions in the program.

into-libc/text, in turn, requires the attacker to learn the location of N chosen functions in both cases, because our function layout randomization has no effect.

Things are different in more general ROP-based attacks. Our strategy completely hinders these attacks by making the location of the gadgets inside a function unpredictable. Given that individual gadgets cannot have their address taken and function pointer arithmetic is generally disallowed in a program, the location of a gadget cannot be explicitly leaked. This makes information leakage attacks ineffective in acquiring any useful knowledge for ROP-based exploitation. In contrast, prior techniques only require the attacker to learn the address of *any* N functions with useful gadgets to mount a successful ROP-based attack. To estimate N, we made an analysis on GNU coreutils (v7.4), building on the results presented in [64]. Figure 7 correlates the number of program functions with the probability of locating all the necessary ROP gadgets, and shows, for example, that learning 16 function addresses is sufficient to carry out an attack in more than 80% of the cases.

Another interesting question is: "*How fast can the attacker acquire the required information*?". Our live rerandomization technique can periodically invalidate the knowledge acquired by an attacker probing the system (e.g., using an arbitrary kernel memory read). Shacham et al. [67] have shown that rerandomization slows down single-target probing attacks by only a factor of 2. As shown in Table 2, however, many attacks require knowledge of multiple targets when fine-grained ASR is in place. In addition, other attacks (e.g., Heap Feng Shui) may require multiple probing rounds to assert intermediate system states. When multiple rounds are required, the attacker is greatly limited by our rerandomization strategy because *any* knowledge acquired is only useful in the current rerandomization window. In particular, let us assume the duration of every round to be distributed according to some probability distribution $p(t)$ (e.g., computed from the probabilities given in [14]). Hence, the time to complete an n-round probing phase is distributed according to the convolution of the individual $p_i(t)$. Assuming the same $p_i(t)$ in every round for simplicity, it can be shown that the expected time before the attacker can complete the probing phase in a single rerandomization window (and thus the attack) is:

$$T_{attack} = T \cdot \left(\int_0^T p^{*n}(\tau) d\tau \right)^{-1},$$

where T is the size (ms) of the rerandomization window, n is the number of probing rounds, and $p^{*n}(t)$ is the n-fold convolution power of $p(t)$. Since the convolution power decreases rapidly with the number of targets n, the attack can quickly become impractical. Given a vulnerability and an attack model characterized by some $p(t)$, this formula gives a practical way to evaluate the impact of a given rerandomization frequency on attack prevention. When a new vulnerability is discovered, this formula can also be used to retune the rerandomization frequency (perhaps accepting a performance penalty) and make the attack immediately impractical, even before an appropriate patch is developed and made available. This property suggests that our ASR design can also be used as the first "live-workaround" system for security vulnerabilities, similar, in spirit, to other systems that provide immediate workarounds to bypass races at runtime [71].

8 Related work

Randomization. Prior work on ASR focuses on randomizing the memory layout of user programs, with solutions based on kernel support [39, 1, 68], linker support [73], compiler-based techniques [12, 14, 72], and binary rewriting [39, 15]. A number of studies have investigated attacks against poorly-randomized programs, including brute forcing [67], partial pointer overwrites [23], and return-oriented programming [64, 61]. Our ASR design is more fine-grained than existing techniques and robust against these attacks and information leakage. In addition, none of the existing approaches can support stateful live rerandomization. The general idea of randomization has also been applied to instruction sets (to thwart code injection attacks) [38, 58, 34], data representation (to protect noncontrol data) [13], data structures (to mitigate rootkits) [46], memory allocators (to protect against heap exploits) [53]. Our struct layout randomization is similar to the one presented in [46], but our ASR design generalizes this strategy to the internal layout of any memory object (including code) and also

allows live layout rerandomization. Finally, randomization as a general form of diversification [26] has been proposed to execute multiple program variants in parallel and detect attacks from divergent behavior [20, 62, 63].

Operating system defenses. Prior work on OS defenses against memory exploits focuses on control-flow attacks. SecVisor [65] is a hypervisor-based solution which uses memory virtualization to enforce W⊕X protection and prevent code injection attacks. Similarly, NICKLE [60] is a VMM-based solution which stores authenticated kernel code in guest-isolated shadow memory regions and transparently redirects execution to these regions at runtime. Unlike SecVisor, NICKLE can support unmodified OSes and seamlessly handle mixed kernel pages with code and data. hvmHarvard [28] is a hypervisor-based solution similar to NICKLE, but improves its performance with a more efficient instruction fetch redirection strategy at the page level. The idea of memory shadowing is also explored in HookSafe [69], a hypervisor-based solution which relocates kernel hooks to dedicated memory pages and employs a hook indirection layer to disallow unauthorized overrides. Other techniques to defend against kernel hook hijacking have suggested dynamic monitoring strategies [74, 57] and compiler-based indirection mechanisms [44]. Finally, Dalton et al. [21] present a buffer overflow detection technique based on dynamic information flow tracking and demonstrate its practical applicability to the Linux kernel. None of the techniques discussed here provides a comprehensive solution to OS-level attacks. Remarkably, none of them protects noncontrol data, a common target of attacks in local exploitation scenarios.

Live rerandomization. Unlike our solution, none of the existing ASR techniques can support live rerandomization with no state loss. Prior work that comes closest to our live rerandomization technique is in the general area of dynamic software updating. Many solutions have been proposed to apply run-time updates to user programs [51, 47, 8, 19] and operating systems [48, 10, 9]. Our rerandomization technique shares with these solutions the ability to modify code and data of a running system without service interruption. The fundamental difference is that these solutions apply run-time changes in place, essentially assuming a fixed memory layout where any state transformation is completely delegated to the programmer. Our solution, in contrast, is generic and automated, and can seamlessly support arbitrary memory layout transformations between variants at runtime. Other solutions have proposed process-level run-time updates to release some of the assumptions on the memory layout [30, 31], but they still delegate the state transfer process completely to the programmer. This completely hinders their applicability in live rerandomization scenarios where arbitrary layout transformations are allowed.

9 Conclusion

In this paper, we introduced the first ASR design for operating systems. To fully explore the design space, we presented an analysis of the different constraints and attack models inside the OS, while highlighting the challenges of OS-level ASR. Our analysis reveals a fundamental gap with long-standing assumptions in existing application-level solutions. For example, we show that information leakage, traditionally dismissed as a relatively rare event, becomes a major concern inside the OS. Building on these observations, our design takes the first step towards truly fine-grained ASR for OSes. While our prototype system is targeted towards component-based OS architectures, the principles and the techniques presented are of much more general applicability. Our technique can also be applied to generic user programs, improving existing application-level techniques in terms of both performance and security, and opening up opportunities for third-generation ASR systems. The key to good performance (and no impact on the distribution model) is our link-time ASR strategy used in combination with live rerandomization. In addition, this strategy is more portable and much safer than existing techniques, which either rely on complex binary rewriting or require a substantial amount of untrusted code exposed to the runtime. In our technique, the complex rerandomization code runs completely sandboxed and any unexpected run-time error has no impact on normal execution. The key to good security is the better randomization granularity combined with periodic live rerandomization. Unlike existing techniques, we can (re)randomize the internal layout of memory objects and periodically rerandomize the system with no service interruption or state loss. These properties are critical to counter information leakage attacks and truly maximize the unobservability of the system.

10 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work has been supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

References

[1] ASLR: leopard versus vista. `http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html`.

[2] Linux vmsplice vulnerabilities. `http://isec.pl/vulnerabilities/isec-0026-vmsplice_to_kernel.txt`.

[3] The story of a simple and dangerous kernel bug. `http://butnotyet.tumblr.com/post/175132533/the-story-of-a-simple-and-dangerous-kernel-bug`.

[4] OpenBSD's IPv6 mbufs remote kernel buffer overflow. http://www.securityfocus.com/archive/1/462728/30/0/threaded, 2007.

[5] Microsoft windows TCP/IP IGMP MLD remote buffer overflow vulnerability. http://www.securityfocus.com/bid/27100, 2008.

[6] FUSE: filesystem in userspace. http://fuse.sourceforge.net/, 2012.

[7] Green hills integrity. http://www.ghs.com/products/rtos/integrity.html, 2012.

[8] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. OPUS: online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.* (2005), vol. 14, pp. 19–19.

[9] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth European Conf. on Computer Systems* (2009), pp. 187–198.

[10] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.* (2007), pp. 1–14.

[11] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. In *Proc. of the 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications* (2002), pp. 1–12.

[12] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.* (2003), p. 8.

[13] BHATKAR, S., AND SEKAR, R. Data space randomization. In *Proc. of the Fifth Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), pp. 1–22.

[14] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th USENIX Security Symp.* (2005), p. 17.

[15] BOJINOV, H., BONEH, D., CANNINGS, R., AND MALCHEV, I. Address space randomization for mobile devices. In *Proc. of the Fourth ACM Conf. on Wireless network security* (2011), pp. 127–138.

[16] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating malicious device drivers in linux. In *Proc. of the USENIX Annual Tech. Conf.* (2010), pp. 9–9.

[17] C-SKILLS. Linux udev trickery. http://c-skills.blogspot.com/2009/04/udev-trickery-cve-2009-1185-and-cve.html.

[18] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of the Second Asia-Pacific Workshop on Systems* (2011).

[19] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P. POLUS: a POwerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Engineering* (2007), pp. 271–281.

[20] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: a secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symp.* (2006), pp. 105–120.

[21] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proc. of the 17th USENIX Security Symp.* (2008), pp. 395–410.

[22] DESIGNER, S. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63.

[23] DURDEN, T. Bypassing PaX ASLR protection.

[24] EDGE, J. Linux ASLR vulnerabilities. http://lwn.net/Articles/330866/, 2009.

[25] ESSER, S. Exploiting the iOS kernel. In *Black Hat USA* (2011).

[26] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building diverse computer systems. In *Proc. of the 6th Workshop on Hot Topics in Operating Systems* (1997), pp. 67–.

[27] GIUFFRIDA, C., CAVALLARO, L., AND TANENBAUM, A. S. We crashed, now what? In *Proc. of the 6th Workshop on Hot Topics in System Dependability* (2010), pp. 1–8.

[28] GRACE, M., WANG, Z., SRINIVASAN, D., LI, J., JIANG, X., LIANG, Z., AND LIAKH, S. Transparent protection of commodity OS kernels using hardware virtualization. In *Proc.of the 6th Conf. on Security and Privacy in Communication Networks* (2010), pp. 162–180.

[29] GUO, P. J., AND ENGLER, D. Linux kernel developer responses to static analysis bug reports. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 285–292.

[30] GUPTA, D., AND JALOTE, P. On line software version change using state transfer between processes. *Softw. Pract. and Exper. 23*, 9 (1993), 949–964.

[31] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades* (2011), pp. 179–184.

[32] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-Pacific Conf. on Advances in Computer Systems Architecture* (2006), pp. 81–94.

[33] HILDEBRAND, D. An architectural overview of QNX. In *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures* (1992), pp. 113–126.

[34] HU, W., HISER, J., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., EVANS, D., KNIGHT, J. C., NGUYEN-TUONG, A., AND ROWANHILL, J. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments* (2006), pp. 2–12.

[35] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 383–398.

[36] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[37] JANMAR, K. FreeBSD 802.11 remote integer overflow. In *Black Hat Europe* (2007).

[38] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conf. on Computer and Commun. Security* (2003), pp. 272–280.

[39] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): towards Fine-Grained randomization of commodity software. In *Proc. of the 22nd Annual Computer Security Appl. Conf.* (2006), pp. 339–348.

[40] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.* (2009), ACM, pp. 207–220.

[41] LABS, O. K. OKL4 community site. http://wiki.ok-labs.com/, 2012.

[42] LATTNER, C., AND ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization* (2004), p. 75.

[43] LENHARTH, A., ADVE, V. S., AND KING, S. T. Recovery domains: an organizing principle for recoverable operating systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 49–60.

[44] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Trans. on Information Forensics and Security 6*, 4 (2011), 1404–1417.

[45] LIAKH, S., GRACE, M., AND JIANG, X. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 271–280.

[46] LIN, Z., RILEY, R. D., AND XU, D. Polymorphing software by randomizing data structure layout. In *Proc. of the 6th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2009), pp. 107–126.

[47] MAKRIS, K., AND BAZZI, R. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 397–410.

[48] MAKRIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second European Conf. on Computer Systems* (2007), pp. 327–340.

[49] MALIK, A. M., MCINNES, J., AND BEEK, P. V. Optimal basic block instruction scheduling for Multiple-Issue processors using constraint programming. In *Proc. of the 18th IEEE Int'l Conf. on Tools with Artificial Intelligence* (2006), pp. 279–287.

[50] MICROSOFT. Windows User-Mode driver framework. `http://msdn.microsoft.com/en-us/windows/hardware/gg463294`, 2010.

[51] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for C. *ACM SIGPLAN Notices 41*, 6 (2006), 72–83.

[52] NERGAL. The advanced return-into-lib(c) exploits. *Phrack Magazine 4*, 58 (2001).

[53] NOVARK, G., AND BERGER, E. D. DieHarder: securing the heap. In *Proc. of the 17th ACM Conf. on Computer and Commun. Security* (2010), pp. 573–584.

[54] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 49–58.

[55] PALIX, N., THOMAS, G., SAHA, S., CALVES, C., LAWALL, J., AND MULLER, G. Faults in linux: ten years later. In *Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 305–318.

[56] PERLA, E., AND OLDANI, M. *A guide to kernel exploitation: attacking the core*. 2010.

[57] PETRONI,JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proc. of the 14th ACM Conf. on Computer and Commun. Security* (2007), pp. 103–115.

[58] PORTOKALIDIS, G., AND KEROMYTIS, A. D. Fast and practical instruction-set randomization for commodity systems. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 41–48.

[59] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. NOZZLE: a defense against heap-spraying code injection attacks. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 169–186.

[60] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent prevention of kernel rootkits with VMM-Based memory shadowing. In *Proc. of the 11th Int'l Conf. on Recent Advances in Intrusion Detection* (2008), pp. 1–20.

[61] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically returning to randomized lib(c). In *Proc. of the 2009 Annual Computer Security Appl. Conf.* (2009), pp. 60–69.

[62] SALAMAT, B., GAL, A., JACKSON, T., MANIVANNAN, K., WAGNER, G., AND FRANZ, M. Multi-variant program execution: Using multi-core systems to defuse Buffer-Overflow vulnerabilities. In *Proc. of the 2008 Int'l Conf. on Complex, Intelligent and Software Intensive Systems* (2008), pp. 843–848.

[63] SALAMAT, B., JACKSON, T., GAL, A., AND FRANZ, M. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the Fourth European Conf. on Computer Systems* (2009), pp. 33–46.

[64] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: exploit hardening made easy. In *Proc. of the 20th USENIX Security Symp.* (2011), p. 25.

[65] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM Symp. on Oper. Systems Prin.* (2007), pp. 335–350.

[66] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Commun. Security* (2007), pp. 552–561.

[67] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conf. on Computer and Commun. Security* (2004), pp. 298–307.

[68] TEAM, P. Overall description of the PaX project. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

[69] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *Proc. of the 16th ACM Conf. on Computer and Commun. Security* (2009), pp. 545–554.

[70] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proc. of the Int'l Workshop on Memory Management* (1992), pp. 1–42.

[71] WU, J., CUI, H., AND YANG, J. Bypassing races in live applications with execution filters. In *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation* (2010), pp. 1–13.

[72] XU, H., AND CHAPIN, S. J. Improving address space randomization with a dynamic offset randomization technique. In *Proc. of the 2006 ACM Symp. on Applied Computing* (2006), pp. 384–391.

[73] XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. In *Proc. of the 22nd Int'l Symp. on Reliable Distributed Systems* (2003), pp. 260– 269.

[74] YIN, H., POOSANKAM, P., HANNA, S., AND SONG, D. HookScout: proactive binary-centric hook detection. In *Proc. of the 7th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2010), pp. 1–20.

[75] YOUNG, C., JOHNSON, D. S., SMITH, M. D., AND KARGER, D. R. Near-optimal intraprocedural branch alignment. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (1997), pp. 183–193.

[76] ZHANG, K., AND WANG, X. Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 17–32.