This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout

Seongman Lee, Hyeonwoo Kang, Jinsoo Jang and Brent Byunghoon Kang, *Member, IEEE*

Abstract—Stack-based memory corruption vulnerabilities have been exploited, allowing attackers to execute arbitrary code and read/write arbitrary memory. Although several solutions have been proposed to prevent memory errors on the stack, they are either limited to a specific type of attack (either spatial or temporal attacks) or cause significant performance degradation. In this paper, we introduce SaVioR, an efficient and comprehensive stack protection mechanism. The key technique involves randomization of the stack layout to reduce its predictability and exploitability. SaVioR isolates an individual object from spatially and temporally adjacent vulnerable objects and randomizes each object's location, which prevents attackers from predicting the stack layout and thus reduces the likelihood of memory errors being exploited. We implemented SaVioR based on the LLVM compiler framework and applied it to the SPEC CPU2006 benchmarks and real-world applications. Our security evaluation showed that SaVioR provides a high degree of randomness in the stack layout and thus reduces the likelihood of successful exploitation of spatial and temporal memory errors on the stack. Our performance evaluation also demonstrated that it incurs a modest performance overhead (14%) with the SPEC CPU2006 benchmark suite, which improves performance compared to the state-of-the-art stack protection while achieving a comparable security level.

Index Terms—Stack, Exploit Mitigation, Randomization

1 Introduction

THE program call stack is an attractive target for adversaries because of the fundamental nature of its organization. That is, the call stack has a high degree of spatial and temporal locality and its allocation algorithm is statically determined. Because of this high spatial locality, a vulnerable buffer and a valuable target (e.g., a return address) can be spatially adjacent to each other on the stack. Moreover, the high temporal locality allows an attacker who recognizes the stack layout to control the content of an uninitialized variable. These features make stack-based attacks more attractive than heap exploitation, which requires complex heap memory layout manipulation [1].

A variety of approaches have been proposed to defend against stack-based memory errors. These approaches can be roughly categorized into four classes: canary-based, shadow stack-based, randomization-based, and comprehensive protection. The canary-based approach [2], [3], [4], [5], [6] has been widely deployed because of its low cost and high compatibility. However, it can be bypassed via memory disclosure of the canary's secret value [7]. Leveraging the shadow stack [8], [9], [10], [11], [12], [13], [14], [15] protects return addresses against buffer overflow attacks but does not protect other variables, e.g., function pointers. The randomization-based approach [16], [17], [18], [19], [20] introduces additional randomness into the stack layout by implanting a random-sized padding between stack frames, randomly allocating vulnerable buffers, and randomizing the base address of the stack region. However, these solutions do not randomize the order of the stack frames, which means that the randomness is not sufficient to stop the attacker from easily predicting the stack layout.

The major limitation of these three approaches is that they provide partial protection. In particular, they only focus on spatial attacks and do not hamper temporal or more advanced spatial attacks. To address both spatial and temporal attacks, the comprehensive randomization technique [21] focuses on increasing the unpredictability of the stack layout by randomly allocating stack-resident objects, e.g., vulnerable buffers, in the stack frame and stack frame itself, and then reducing the predictability of stack frame reuse. Unfortunately, this approach suffers from significant performance overhead (28%) and has poor compatibility with existing software such as C++ applications.

In this paper, we present SaVioR, an efficient and comprehensive stack protection mechanism that introduces unpredictability into the stack layout. More specifically, SaVioR takes advantage of the fact that 64-bit virtual address space is plentiful and creates multiple stacks distributed sparsely. Then, SaVioR randomly allocates stack-resident objects and stack frames itself into one of the multiple stacks at each function invocation. In particular, SaVioR introduces two techniques: Vulnerable Buffer Isolation (VBI) and Stack Frame Randomization (SFR). VBI randomizes and isolates the location of variables that might be vulnerable to spatial attacks (these variables are also known as *attack vectors*). This creates a huge gap between vulnerable variables and valuable variables (called *target variables* in this paper) and thus hinders spatial attacks. SFR randomizes the stack frame allocation and introduces random-sized padding, which randomizes the absolute and relative address of stack-resident objects and avoids the reuse of recently deallocated stack frames, to prevent temporal attacks.

We implemented SaVioR based on LLVM compiler

- *S. Lee, H. Kang, and B. Kang are with Korea Advanced Institute of Science and Technology.*
  E-mail: onlyreason, kanglib, brentkang@kaist.ac.kr
- *J. Jang is with Chungnam National University. E-mail: jisjang@cnu.ac.kr*

frameworks and have evaluated its efficiency using the SPEC CPU2006 benchmark suite and real-world web server applications. To demonstrate the effectiveness of SaVioR, we conducted statistical and empirical security analyses. In particular, we used four stack-based real-world vulnerabilities[1] in the empirical security analysis and observed that SaVioR effectively impedes them. Our evaluation demonstrates that SaVioR provides a high level of randomness in the stack layout and provides comprehensive protection against spatial and temporal attacks on the stack with only moderate performance impacts (13%). In summary, this paper makes the following contributions:

- SaVioR provides comprehensive stack protection against all existing spatial and temporal attacks on the stack: intra-frame, inter-frame, use-after-free (UaF), and uninitialized read (UR) attacks.
- We propose a new way to randomize stack-resident objects using pointer mirroring. For portability and applicability, we implemented SaVioR based on the LLVM compiler framework.
- To demonstrate the efficiency and effectiveness of SaVioR, we applied SaVioR to SPEC CPU2006 and PARSEC 3.0 benchmark suites and popular real-world applications. We show that SaVioR achieves better runtime performance than the state-of-the-art stack memory protection at a comparable security level.

2 Background & Related Work

We describe existing approaches to mitigate stack-based memory corruption attacks and their limitations.

Canary-based Defenses. To prevent stack-based buffer overflows, canary-based defenses [2], [3] place a secret value, called a canary, before return addresses and check its integrity upon returning from a function. Moreover, more advanced approaches [4], [5], [6], [22] have been proposed to compensate for the shortcomings of the previous defenses. For example, Dynaguard [5] and RAF SSP [22] have been proposed to prevent byte-by-byte brute-force attacks on the canary. They re-randomize the canary value in the forked process at runtime. As another example, PCan [4] introduces an unmodifiable and unbypassable canary by leveraging ARMv8.3-A Pointer Authentication.

Because of their low-overhead and high-compatibility, these approaches are widely adopted in practice. However, they assume that the return address is intact if and only if the canary is intact. Therefore, if the canary is exposed to an attacker through an information leak (e.g., buffer overread), the attacker can overwrite the canary with the leaked value. Hence, the attacker can modify the return address without spoiling the canary value on the stack. Moreover, these defenses cannot prevent non-contiguous buffer overflow attacks such as an arbitrary write through a stale pointer.

Randomization-based Defenses. Knowledge of the memory layout is a general prerequisite for successful exploitation. Therefore, researchers have proposed randomization of the memory layout to reduce its predictability. A well-known example of this approach is ASLR [16], which is widely deployed in modern operating systems. The ASLR randomizes the base address of the stack segment. However, ASLR is vulnerable to information leakage attacks. Moreover, the relative distance between two stack-resident objects is not randomized, thereby making it easy to exploit vulnerabilities on the stack without information leakage.

To address this shortcoming, fine-grained ASLR on the stack [17], [18], [19], [20], [23] has been proposed. They 1) introduce random padding between stack frames [17], [18], [20], [23], 2) randomize the relative distance between stack-resident objects or the alignment [19], [23], and 3) insert random padding between buffer type and non-buffer type objects [23]. However, these solutions provide weak stack protection. For example, placing random padding between stack frames might be effective against inter-frame attacks, but it is not effective against intra-frame attacks, since the layout of the stack-resident objects within the same frame is unchanged. Moreover, the entropy in gap randomization is limited. In addition, none of these methods randomize the order of stack frames. More seriously, [17] inserts random gaps that are too small (up to 64 bytes) and [23] determines the size of random gaps between each object at *compile-time*, not runtime. Therefore, an attacker can acquire the compiled binary and reverse engineer the stack allocation algorithm to predict the stack memory layout.

Shadow Stack-based Defenses. The aim of shadow stack techniques [8], [9], [10], [11], [12] is to protect the integrity of backward edges (i.e., returns). In these schemes, return addresses on the stack are copied and isolated in the shadow stack. Then, the integrity of the return addresses is verified before function returns, which effectively prevents control-flow hijacking attacks that attempt to modify return addresses. However, it does not protect other objects (e.g., function pointers) on the stack. Moreover, these approaches have to instrument `call/ret` instructions, which are frequently executed at runtime, hence incurring significant performance overhead.

To reduce the high overhead of the shadow stack technique [2], researchers have proposed a variant of shadow stack methods called safe stack [13], [14], [15], [18], [24], [25]. This approach isolates valuable stack-resident objects (e.g., return addresses and spilled registers) in the safe stack and thus protects them from an attack abusing the vulnerable buffers. However, the vulnerable buffers themselves can still be targets of an attack depending on which objects they maintain (e.g., a function pointer array). Therefore, the effectiveness of this approach is limited to preventing buffer-to-non-buffer attacks.

Comprehensive Protection. For a comprehensive solution against spatial and temporal memory safety violations, Chen et al. introduced StackArmor [21]. StackArmor randomizes the allocation of stack frames at each unsafe function invocation; hence, stack frames are randomly allocated on the stack in a non-contiguous and sparse way, which

[1] CVE-2013-2028, CVE-2019-11038, CVE-2019-9639, and CVE-2018-1000140

[2] Intel has recently introduced a new hardware-enforced shadow stack, called Intel Control-Flow Enforcement Technology (Intel CET), as part of the Tiger Lake architecture in 2020. This will be discussed in detail in Section 7.3.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

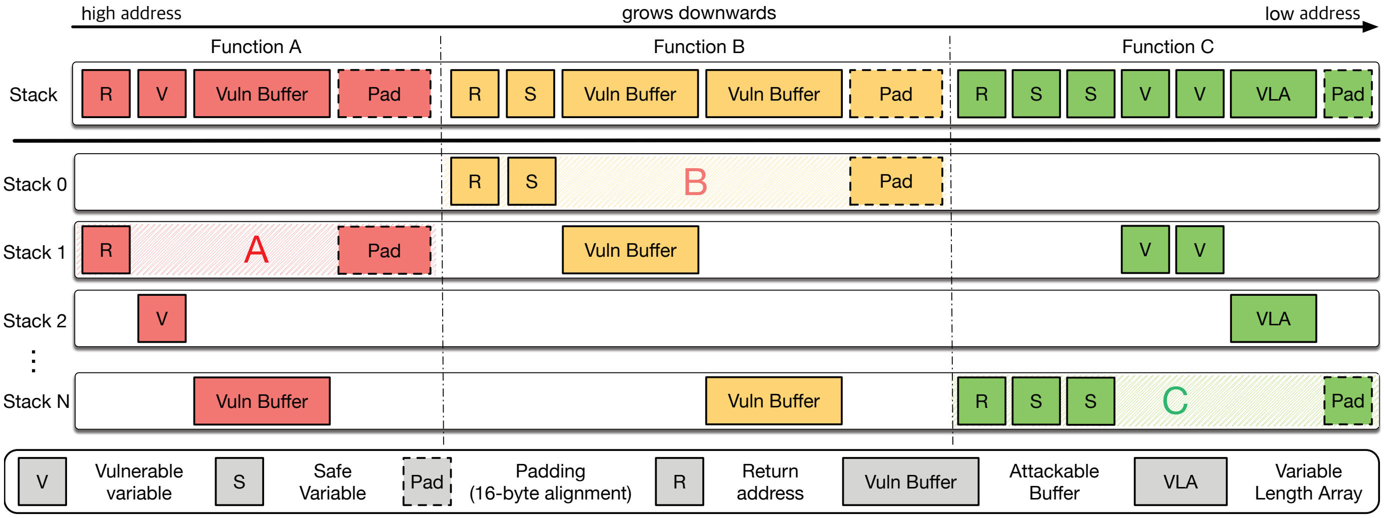


Fig. 1: SaVioR design overview. VBI ensures that attack vectors (e.g. `Vuln Buffer`) and target variables (e.g., `R`) are not adjacent and separated onto different virtual stacks. SFR modifies the conventional stack frame organization by spreading the stack frame over virtual stacks and introducing random padding between stack frames. Both VBI and SFR make the results of spatial and temporal attacks unpredictable. Note that the random padding scheme is not shown for simplicity, since the size of each random padding is normally bigger than that of a stack frame.

introduces unpredictability into the stack layout and the reuse of stack frames. However, it incurs an average runtime overhead of 22% (except for the zero initialization overhead) on the SPEC CPU2006 benchmarks. Moreover, StackArmor fails to support C++ applications because its implementation depends on a binary rewriter that is not sophisticated enough to handle complex C++ binaries. SmokeStack [19] also provides unpredictability to the stack layout. It randomizes the order of stack variables at each function invocation. However, it neither randomizes the stack frame order nor introduces a random gap between stack frames. Moreover, it does not protect control data (non-control data is considered).

3 Threat Model

We consider an attacker who can provide maliciously crafted input to a vulnerable piece of software that has spatial and temporal memory corruption vulnerabilities on the stack. Moreover, we assume that the attacker has full access to the compiled binary and is capable of reverse-engineering it to determine the stack allocation algorithm. Besides, the attacker can perform a limited number of brute-force attacks to bypass SaVioR. Attempts with incorrect addresses may cause the victim program to crash, and the system administrator will detect the failed attack. Our adversary model is consistent with those used in the related works in this area [19], [21].

4 Design

4.1 Overview

A traditional stack grows linearly, and stack-resident objects are spatially and temporally adjacent to each other, which means that the stack has a highly exploitable and predictable memory layout. Hence, using this characteristic of the stack, an adversary can easily predict the layout of the stack and can reliably exploit spatial and temporal errors on the stack.

To introduce uncertainty into the stack memory layout, SaVioR takes advantage of huge 64-bit virtual address space, which creates multiple stacks and spreads out the stacks far apart in the address space. Based on this, SaVioR randomly spread out stack-resident objects across multiple stacks. This enables stack-resident objects to be *spatially* and *temporally* away from each other. This means that it reduces the likelihood of memory errors being exploited because each object is isolated and rarely reused. In particular, SaVioR introduces two techniques, vulnerable buffer isolation (VBI) and stack frame randomization (SFR), to isolate the vulnerable variables and randomly shuffle the stack frame, respectively.

4.2 Vulnerable Buffer Isolation

VBI isolates vulnerable variables that could be used as an attack vector from target variables spatially. VBI considers any stack-resident object that might cause spatial memory errors to be vulnerable and thus randomizes the location of those variables at each function invocation.

Here, we describe in detail how VBI randomizes the stack layout. Figure 1 depicts the stack layout in a legacy application and a SaVioR-enabled application when three functions are invoked in a row. The SaVioR-enabled application has one main stack (Stack 0) and multiple virtual stacks (Stacks 1–N) while the legacy application has the only one. VBI randomly relocates all vulnerable variables and thus isolates target variables from spatial memory errors. For example, because `Function A` (highlighted in red) has the vulnerable variable `Vuln Buffer` that should be relocated to separate them from its frame, SaVioR relocated `Vuln Buffer` into the randomly chosen `Stack N`, and thus it is separated from the return address `R`. By doing so, the vulnerable variable is not *spatially* adjacent to the valuable targets. (The reason that the frame of Function A is placed on `Stack 1` will be explained later.)

Note that, although the locations of vulnerable objects are randomized, SaVioR ensures that the relocated objects

	Spatial attack	Temporal Attack		VBI	SFR
	Out of Bounds	UaF	UR		
Buffer	✔			✔	
Aggregate	✔			✔	
Addr-taken	✔	✔	✔	✔	✔
VLA	✔			✔	
Potential UR			✔		✔

TABLE 1: Summary of attack vectors and our defenses. The risk that a type of vulnerable object is used for an specific attack is marked in ✔. The table also indicates which defense is used to mitigate attacks, which is marked in ✔. In case of the address-taken type, SaVioR partially applies VBI to this type of variable; thus, it is marked in ✔.

are *always* relocated on a different stack rather than the current one. Therefore, it is hard to perform spatial attacks on the relocated vulnerable object because the object is completely separated from its stack frame. Furthermore, vulnerable variables are randomly placed on one of the stacks for each function call, which reduces the possibility of reusing the memory that was recently used. Hence, as well as spatial attacks, the VBI technique impedes *temporal* attacks which exploit temporal locality.

4.3 Stack Frame Randomization

Although VBI isolates vulnerable attack vectors from target variables, it does not randomize the other stack-resident objects, e.g., function pointers and decision-making variables, which would be an attractive target for an attacker. In other words, each variable that could be attacked remains in *its stack frame* without randomization. Hence, an attacker with access to the compiled binary can predict the location of stack-resident objects that remain in the stack frame because stack frames are (de)allocated and reused in a predictable and deterministic manner.

To handle this problem and complement the VBI technique, SaVioR introduces SFR, which inserts a random-sized padding between each stack frame and randomizes the allocation of stack frames in a non-contiguous and out-of-order way. One would think that VBI can also be deployed to randomize the location of every variable in the stack frames instead of randomizing the stack frames. However, randomizing every variable introduces significant runtime overheads because of the increased random number generations and has unacceptably poor locality. Therefore, we randomize the location of the stack frame that contains the potential target variables instead of randomizing individual variables. We argue that a randomized stack frame can be considered to be secure from spatial attacks despite the close proximity of each other, because vulnerable variables are separated from the stack frame by VBI.

Furthermore, SFR brings another benefit. In traditional stack, stack frames are *contiguously* (de)allocated and thus *frequently* reused on the stack according to the order of function invocations and returns at runtime, which allows an attacker to analyze the stack layout trivially. In contrast, SFR assures us that the stack frame is *not contiguous* and *not reused frequently*, which reduces the predictability of stack frame reuse and thus makes the temporal attacks unreliable.

To illustrate how SFR works in practice, we refer again to Figure 1 that depicts the layout of the SFR-applied stack frames. For `Functions A`, `B`, and `C`, the stack frames are located in `Stack 1`, `0`, and `N`, respectively. In addition, after randomly selecting one of the stacks in which the stack frame will be placed, SFR inserts a random-sized padding into the stack frame, as shown in the dotted box.

4.4 Identifying Stack Objects to be Randomized

During runtime, the call stack is extensively used, and `call/ret` pairs are frequently executed, which means that high locality of reference on the stack is necessary for performance. SaVioR sacrifices the spatial and temporal locality of reference on the stack for security. Moreover, SaVioR inserts several instructions to randomize stack-resident objects. This is another root cause of performance degradation. In particular, we observed that SFR instrumentation significantly decreases instructions per cycle (IPC) when compared with a naive function call in the original program. Hence, it is impractical to apply our approach to all functions and stack objects. A trade-off between security and performance needs to considered when these techniques are applied. With that in mind, we decided to selectively randomize the stack-resident objects.

VBI-applied objects. The identification of vulnerable objects to be VBI-applied is performed similarly to that of the well-known stack protector provided by the GCC and Clang compilers. Specifically, SaVioR adopts a policy that is similar to that of `-fstack-protector-strong` to identify vulnerable variables. This overestimates the set of objects that might be vulnerable to spatial attacks; for example, it protects all objects involved in pointer arithmetic operation with a non-constant, and this could include an object that could be accessed in an in-bound and timely manner as vulnerable. We argue that `-fstack-protector-strong`'s policy could cover numerous vulnerable objects and is sufficient for VBI to use for vulnerable object identification.

In detail, SaVioR applies VBI to 1) any type of array, 2) any aggregate type that contains an array, and 3) VLA. In the case of address-taken variables, SaVioR does not apply VBI to an individual address-taken variable like the other types (1–3) of variables. Instead, SaVioR creates a frame with only address-taken variables and applies SFR to it. This separates address-taken variables from its stack frame. Although, they remain vulnerable to spatial attacks among themselves, we argue that the likelihood that an address-taken variable will cause spatial attacks is lower than that of the others (1–3).

SFR-applied objects. The aim of SFR is to avoid the reuse of a previously returned stack frame by a newly created stack frame and randomize the location of target variables. By doing so, we can prevent an adversary from abusing normal stack behavior that linearly locates the stack frames for sub-function calls and reuses previous stack frames. In particular, temporal error-based attacks such as triggering UR and UaF can be hampered by SFR. Therefore, SFR is moderately applied to the functions that may have a temporal error, i.e., UaF or UR errors.

Stack-based UaF (use-after-return) vulnerability are rarely reported [26]. Most UaF attacks target *heap*-allocated objects because the adversary can explicitly manage them. Nevertheless, we still consider the stack-based UaF in our attack model because of its severity. The Towelroot exploit

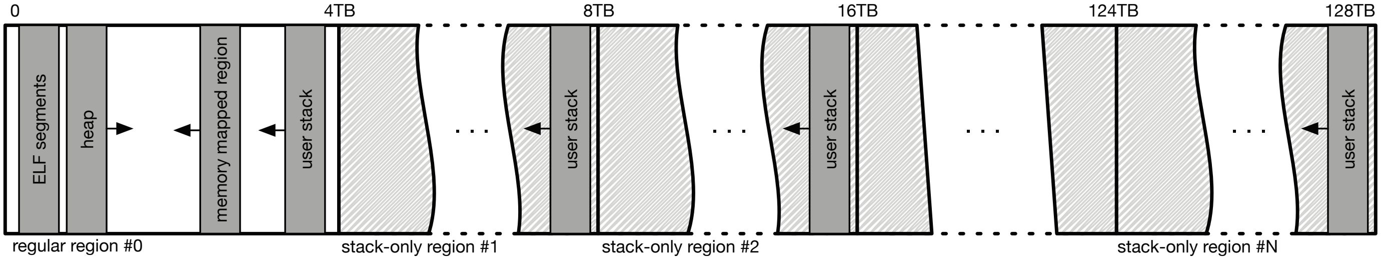


Fig. 2: Virtual address space in a SaVioR-protected application

(CVE-2014-3153) on Linux kernel 3.14.5, which exploits the kernel stack UaF vulnerability, allows users to obtain root access to Android devices. A dangling pointer on the stack can be created by a object deallocated by a sub-function return. Using an address-taken variable is one of the most likely situations to create the dangling pointer. Therefore, SaVioR regards functions with an *address-taken* variable as an attacker's preferred targets and applies SFR to them.

In the case of UR, statically detecting stack-allocated objects that are vulnerable to UR is challenging. Because the place to initialize a variable and the place to use the variable are separated across multiple functions, it is difficult to pinpoint an UR. Hence, modern compilers and language runtime systems zero-initialize *all* memory allocations. For example, an experimentally adopted solution in the LLVM compiler zero-initializes any local variable without giving it an initial value [27]. Unfortunately, recent publications [28], [29] have analyzed the cost of zero-initialization and have shown that it is high. Instead of zero-initializing all the variables, the `-Wunintialized` option in the Clang compiler is one way to detect the uninitialized use of stack variables. However, this option provides poor coverage; for example, in the Clang 8.0.0 compiler, this option is neither flow-sensitive nor field-sensitive for aggregate types. In GCC 5.5, it is flow-sensitive but not field-sensitive. Table 3 gives a comparison of whether our detection tool and recent compilers performs flow-, field-, or byte-sensitive analysis.

To detect any variable that may cause an uninitialized use of stack memory, SaVioR performs a straightforward intra-procedural analysis that detects any variable that might be accessed without proper initialization. This analysis is byte-level, path-insensitive, flow-sensitive, and field-sensitive, but it is not context-sensitive. Hence, in the case of address-taken variables that are passed as an argument to a function, we conservatively consider these variables to be vulnerable to UR attacks and apply SFR to them.

Note that the SFR is also applied to the functions to which VBI has been applied. When only VBI is used, target variables are placed in the same region without randomization and thus become very predictable. If only SFR is used, only the location of the stack frames is randomized, and vulnerable and safe variables are left adjacent to each other in the stack frame. By applying the two techniques together, they complement each other's weaknesses and introduce additional randomness into the stack layout.

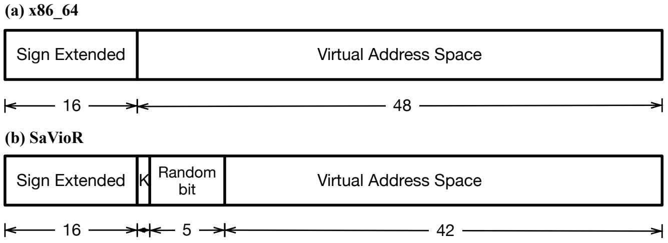


Fig. 3: Format of a SaVioR pointer

4.5 Address Space Configuration & Stack Layout Randomization

Here, we elaborate on how SaVioR configures its address space and randomizes the stack layout.

4.5.1 Setting Up the Virtual Address Space

To implement the VBI and SFR techniques, we leverage virtual address space on 64-bit systems. As shown in Figure 2, a SaVioR-enabled application's address space is divided into two types of regions: one *regular region* and multiple *stack-only regions*. The regular region is similar to the address space of a legacy process that maintains code, data, heap, memory-mapped, and stack segments, but the available address space is limited to b (<47) bits. From now on, we assume that the regular region is limited to 42 bits, which is configurable with a build option. (The reason why we chose a 42-bit address space for describing our design is explained in Section 4.5.3.) The remaining address space above 42 bits is for the stack-only regions that each holds a virtual stack corresponding to the original stack in the regular region.

Figure 3a shows the pointer representation in a `x86_64` Linux process. The size of a pointer is 64 bits, but only the least significant 48 bits are actually used; the 16 most significant bits are copies of bit 47 (sign-extension). Moreover, the 48-bit virtual address space is divided into two halves; the upper and lower halves are reserved for the kernel and user, respectively. We only consider the user address space.

Figure 3b shows the pointer representation in SaVioR. The 47-bit virtual address space is evenly divided into 32 42-bit regions, each of which has different upper 5 bits, which are referred to as *random bits*. The lowest address space (random bit 0) is for the regular region, and the other address spaces (random bits 1–31) are for the stack-only regions. For each stack only-region, at process initialization, SaVioR creates additional stacks in a way such that each created stack has the same address range, from bits 41 to 0 as the regular stack, excluding the random bits. Figure 2 shows the virtual address space configuration of a SaVioR-enabled application when implemented as described.

4.5.2 Randomizing Stack Objects using Pointer Mirroring

SaVioR makes use of the random bits for randomizing vulnerable variables and stack frames. More specifically, SaVioR prepares multiple stacks corresponding to the random bits. Then, when a VBI/SFR-protected function is invoked, stack-resident objects are randomly placed in one of the multiple stacks in the regular region and the stack-only regions. This is achieved by randomly changing the random bits of the randomized stack object's address. We call this mechanism *pointer mirroring*. Although the pointer mirroring technique was first introduced in the lowfat-stack [30], which provides deterministic memory safety, our approach is distinct in that it deploys this technique to randomly relocate the location of stack-resident objects, not to locate the region for placing objects.

For VBI-applied functions, SaVioR randomizes the vulnerable objects and modifies all stack access instructions to reference randomly relocated objects. In addition, to apply SFR protection, SaVioR instruments all call instructions invoking SFR-applied functions to relocate the stack frame by randomizing the stack pointer (i.e., `rsp` in `x86_64`) Hence, it does not need to modify stack access instructions.

4.5.3 Address Space Reduction

SaVioR, which makes use of some upper bits of the pointer to randomize the location of stack objects, provides randomization at the cost of addressable virtual address space.

This is similar to the existing tagged pointer techniques [31], [32], [33] that utilize the upper bits to hold metadata for enforcing a security policy. Because the addressable virtual address space limits the entropy of ASLR, pointer tagging-based defenses, unfortunately, have a low degree of ASLR entropy. Because these defenses provide deterministic spatial memory safety that eradicates all memory corruptions, it does not require ASLR which probabilistically mitigates the exploitation of memory corruption after this memory corruption has been triggered.

Similar to pointer tagging defenses, SaVioR encodes the stack identification (number) in which the randomized stack object will be located in the upper bits of a pointer. However, on the contrary, SaVioR should be completely compatible with and rely on ASLR because it introduces additional randomization onto the stack. Hence, it is necessary to avoid impacting the effectiveness of ASLR when reducing the available address space; as the randomness entropy of SaVioR (i.e., the number of stacks and random bits) is increased, the effectiveness of ASLR is increasingly undermined.

Here, we describe the minimum amount of available address space that is needed to provide the default ASLR entropy in an `x86_64` vanilla Linux kernel. A Linux kernel defines four regions: the executable and `brk` regions, which are placed from the lowest address, and the stack and memory-mapped region, which are placed from the highest address. The executable region contains an ELF binary and has 28 bits of entropy for a PIE-enabled binary. The brk region is usually for the heap and has 13 bits of entropy. Next, the memory-mapped region that contains libraries and the memory-mapped area has 28 bits of entropy. Finally, the stack region has 22 bits of entropy. Note that, because ASLR is based on page-level granularity, the 12 least significant bits are fixed. This means that the number of addressable bits for n-bit entropy ranges from 0 to 2^{n+12}. In summary, to support the entropy of ASLR in `x86_64` Linux, at least 40, 25, 40, and 34 bits of addressable address space must be reserved for the ELF, heap, and memory-mapped, and stack segments, respectively (this does not take into account the size of each segment). This means that an address space of at least 42 bits is required to provide the default level of ASLR entropy in `x86_64` Linux.

```
1  #define SLR_ADDRESS_SPACE 42
2
3  uint64_t slr_random();
4
5  void* slr_alloca(void *ptr, uint64_t rnd) {
6      uintptr_t slr_ptr;
7      slr_ptr = (uintptr_t) ptr ^ (rnd << SLR_ADDRESS_SPACE);
8      return (void*) slr_ptr;
9  }
```

Listing 1: Helper Functions

Given that five random bits only support 32 stacks, SaVioR cannot be considered to be secure against memory errors. Thankfully, Intel has proposed 5-level paging [34] that extends the addressable address space from 48 bits to 57 bits. This enables SaVioR to support up to 14 random bits, which can deploy up to 16,384 (2^{14}) stacks *per thread* while not diminishing the entropy of ASLR on 64-bit Linux. Recently, 5-level paging has become available in the latest Intel Ice Lake U/Y-series processors designed for laptops and mobile devices. This will be discussed in more detail in Section 7.

5 Implementation

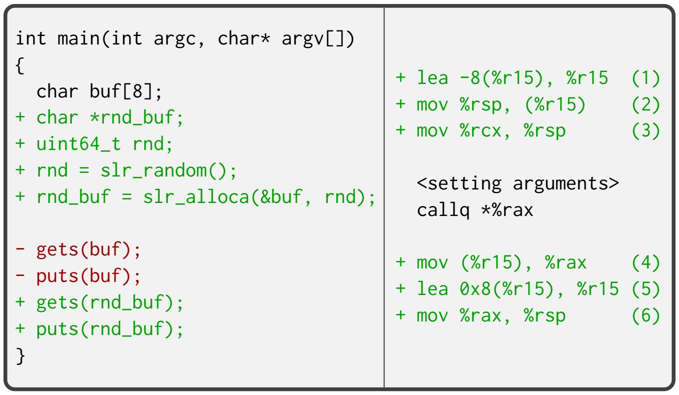
We implemented SaVioR for Linux-5.0.8 on the `x86_64` architecture based on the LLVM/Clang-8.0.0 compiler framework. The LLVM compiler framework is required for the static instrumentation and runtime library. Besides, the kernel is modified to shrink the virtual address space of a SaVioR-protected application.

5.1 Static Instrumentation in LLVM

Here, we explain how the static instrumentation modules for VBI and SFR are implemented.

Helper Functions. `slr_random` in line 3 in Listing 1 generates a random number. This is implemented as an LLVM x86 intrinsic, not as a function. We implemented an AES-based random number generation based on Intel AES-NI instruction sets in a leakage-resilient way; the generated random number and random secret key are stored in dedicated registers, `xmm14` and `xmm15`, respectively. Once the application has been compiled with SaVioR, SaVioR guarantees that the dedicated registers do not spill into memory.

To generate a random number, `slr_random` runs one `aesenc` instruction and then obtains a 128-bit random value. The generated value consists of the upper 64 bits and the lower 64 bits of the xmm14 register, which can be extracted using the `pextrq` and `movq` instructions, respectively. The



```
int main(int argc, char* argv[])
{
  char buf[8];
+ char *rnd_buf;
+ uint64_t rnd;
+ rnd = slr_random();
+ rnd_buf = slr_alloca(&buf, rnd);

- gets(buf);
- puts(buf);
+ gets(rnd_buf);
+ puts(rnd_buf);
}
```

```
+ lea -8(%r15), %r15  (1)
+ mov %rsp, (%r15)    (2)
+ mov %rcx, %rsp      (3)

  <setting arguments>
  callq *%rax

+ mov (%r15), %rax    (4)
+ lea 0x8(%r15), %r15 (5)
+ mov %rax, %rsp      (6)
```

Fig. 4: VBI & SFR instrumentation

generated random value is used for randomizing the stack-resident objects by VBI and SFR. One advantage of the compiler-based approach is that SaVioR is implemented as a compiler option so that a developer can selectively enable SFR and VBI. When both SFR and VBI are activated, SaVioR first uses SFR to randomize the stack frame. Then, vulnerable variables are randomized by VBI. This allows the VBI to use the random value remaining after SFR is complete, which avoids the need to regenerate the random value. That is, the lower and upper parts of the random number are used for applying SFR and VBI, respectively.

`slr_alloca` in line 5 is used to implement the pointer mirroring mechanism, which is used for VBI and SFR instrumentation to randomize vulnerable variables and stack frames, respectively. It takes two arguments: (1) pointer (`ptr`) which is either a vulnerable object or a stack pointer and (2) random number (`rnd`) which is used to choose the stack where the randomized object will be placed. `slr_alloca` mirrors the `ptr` into the one of the stacks by setting the random bits. In detail, `slr_alloca` enforces the location of an object to be randomized in a way that it is *always* relocated onto a different stack other than the current one, ensuring that vulnerable variables are completely separated from its original stack frame. To accomplish this, in line 7, `rnd` value is XORed with the current stack number in which the object is placed (the current stack number is embedded in the random bits of `ptr`).

VBI Pass. The VBI pass locates `alloca` instructions in LLVM IR, which contains information of the stack variable (e.g., type and length), and instruments all functions that contain a vulnerable variable, as defined in Section 4.4.

How the VBI instrumentation works is presented as pseudo code in Figure 4 (left). It instruments the `main` function prologue to randomize the location of the buffer (`buf`). First, the instrumented `main` function invokes `slr_random` to obtain a random value. For simplicity in the figure, we assume that `slr_random` returns a non-zero random number in the range $[1, N)$, where N represents the number of deployed stacks. This is because if `slr_alloca` takes a zero value as an argument, it XORs the current stack number with the zero value and thus does not separate vulnerable buffers from its stack frame. However, in a real implementation, the LLVM compiler inserts some instructions that (1) mask out all bits of a random value except those bits required to randomly select a stack from among the available stacks and (2) ensure that a non-zero random number is passed to `slr_alloca`, which can be expressed in the following way: "`rnd = rnd %` (2^n - 1) + 1," where `rnd` and `n` represent a random value and the number of random bits, respectively. Next, `slr_alloca` is invoked with `buf` and `rnd` as arguments, which returns the randomized buffer (`rnd_buf`). Lastly, the pass instruments all instructions that reference the original `buf` to use the randomized one `rnd_buf` through the `replaceAllUsesWith` LLVM API. Note that helper function calls are inlined to reduce the performance overheads.

SFR Pass. The SFR pass identifies all functions that have an address-taken variable or an uninitialized use of a variable. Besides, it instruments every call site of a VBI/SFR-protected function for stack frame randomization.

To identify address-taken variables, the SFR instrumentation module performs an intra-procedure data-flow analysis that tracks all uses of each `alloca` instruction and considers it address-taken if it is passed as a function argument and is involved in pointer arithmetic operations and integer casting as a source operand.

To detect the use of uninitialized variables, our analysis builds on top of UniSan [35]. Our analysis performs intra-procedure data-flow analysis in a flow-sensitive, path-insensitive, and field-sensitive way. For every stack object, this analysis tracks every execution path and checks whether all uses of a stack object are dominated by the proper initialization point. By doing so, the SFR pass can determine whether a variable might be used without the proper initialization (UR).

Before explaining the SFR instrumentation in detail, we need to elucidate the `stack pointer restore buffer` (SPRB) that holds the address of the caller's stack frames. SFR randomizes a stack frame before jumping to the function. Hence, SFR should ensure that a function return has to restore the caller's stack frame. Otherwise, the caller will lose it when returning. To this end, the SaVioR runtime environment maintains SPRB. Whenever an SFR-protected function call occurs, the current stack pointer is saved onto the top of SPRB's entry, and the corresponding stack pointer is restored when returning the function.

We hide the SPRB from an adversary through information hiding, which takes advantage of the huge 64-bit virtual address space that is sufficient to hide a sensitive region. Moreover, SaVioR ensures that there are no references to it that do not go through the dedicated register. In our implementation, we chose the `r15` register, which is a callee-saved register and the least-used register in hand-written assembly, as the dedicated register for the SPRB. Previous studies [14], [15] have influenced this decision.

Figure 4 (right) shows how SaVioR instruments the call site of the SFR-protected function in assembly code. In this figure, we assume that the stack pointer `rsp` is already randomized through `slr_alloca`, which takes as its argument the stack pointer (`rsp`) and returns the randomized one in the `rcx` register. The SPRB is a full descending stack, and thus the dedicated register `r15` points to the highest address containing the valid one. Hence, (1) `lea` instruction decrements `r15` by 8 bytes before storing the current stack pointer, (2) the stack pointer is stored in the top entry of

SPRB, and (3) the stack pointer is set to the randomized stack pointer stored in rcx. Likewise, the restoration of the stack pointer is executed in reverse order, except the randomization of the stack pointer (steps 4–6).

The SFR pass needs to access the stack pointer (i.e., rsp) and the dedicated r15 register to randomize the stack frame. To this end, we use the existing intrinsics llvm.stacksave and llvm.stackrestore, which retrieve and restore the stack pointer at IR-level, respectively. In the case of the r15 register, we utilize LLVM InlineAsm, which allows us to insert inline assembly instructions to access the r15 register. Although the llvm.read_register and llvm.write_register intrinsics, which respectively get and set a specified register, exist, LLVM 8.0.0 does not fully implement them.

Difference between VBI and SFR passes. Like the VBI pass, the SFR pass operates on the LLVM IR level. The main difference between them is that the SFR pass must perform a whole-program analysis to identify all the functions to be protected and applies SFR instrumentation to the corresponding call-sites, whereas the VBI pass does not. Therefore, while the VBI pass runs on each individual IR file, the SFR pass does not perform the analysis and instrumentation until the link-time optimization (LTO) pass links all IR files into a single IR file.

Callee Identification. The SFR pass applies the SFR instrumentation to all direct and indirect call-sites whose target is potentially a VBI- or SFR-protected function. In detail, for direct call-sites, the SFR pass clearly identifies the target of the direct call-sites. However, for indirect calls, the SFR pass performs type-based analysis to identify all potential targets of the indirect call-sites. More specifically, it applies SFR to indirect call sites if the types of the arguments of the indirect call site are the same as one of the functions to be SFR-protected (or VBI-protected). Moreover, we conservatively assume that universal pointer types such as char* and void* are equivalent.

Backend Modification. We modified the LLVM backend to support the following: (1) random number generation and (2) register reservation. First, both the VBI and SFR passes must generate a random number for randomization. To this end, we implemented the slr_random x86 intrinsic, which leverages Intel AES-NI instructions to generate random numbers. Second, SaVioR ensures that only instrumented code can access the dedicated register. In contrast to the GCC compiler, which provides the -ffixed flag to prevent the compiled binary from accessing the specified register, the LLVM compiler does not support the equivalent compile options. Therefore, we modified the LLVM backend for the x86_64 architecture to reserve the r15, xmm14, and xmm15 registers.

5.2 Kernel Modification

The x86_64 Linux kernel provides a user process with a 47-bit virtual address space. SaVioR trades the available space for the randomization. However, we observed that Linux kernel loads the PIE-enabled ELF binary above ELF_ET_DYN_BASE (0x555555554AAA $\approx 2^{46}$), which means that there are no remaining bits of a pointer for SaVioR. Therefore, we modified the Linux kernel to map the ELF segments of a process further down.

The kernel is responsible for loading the main executable and the dynamic loader while creating a new process. The ELF_ET_DYN_BASE specifies the lowest address where the dynamic loader will be loaded to maintain sufficient distance between the dynamic loader and the non-PIE main binary. However, given that most Linux distributors compile prebuilt packages as PIEs by default, we conclude that it is reasonable to lower ELF_ET_DYN_BASE to 0x100000000 (4 GB). This is in line with previous studies [36], [37].

In addition, we have to lower the address of the user stack. The user stack is also created by the kernel loader and is loaded with a negative random offset from STACK_TOP (= TASK_SIZE). We defined STACK_TOP_SLR (2^{42}), which is configurable upon the number of deployed virtual stacks, and is used instead of STACK_TOP when the kernel loader initializes a SaVioR-protected application. In total, these changes consist of 19 lines of code in the Linux kernel diff.

5.3 Runtime Library

After the Linux kernel configures the address space of a SaVioR-protected application, the compiler-rt runtime support library is responsible for 1) creating multiple virtual stacks in the stack-only regions, 2) setting the dedicated register r15 to point to the SPRB, 3) initializing dedicated registers, xmm14 and xmm15, for storing the generated random number and AES secret key, respectively, and 4) supporting multi-threading and non-local jumps.

The runtime support library includes an initialization function that leverages the constructor attribute, which is executed before the main function. This initialization function creates multiple stacks for each stack-only region and initializes the dedicated register r15. More specifically, it locates the main stack created by the kernel using pthread_attr_getstack, which returns several attributes associated the stack (e.g., the address and length). Then, it mirrors the regular stack to the corresponding stack in the stack-only Region 1, and the same procedure is repeated until all mirrored stacks are created in every stack-only region. Next, the dedicated register r15 is initialized to point the SPRB. Lastly, to initialize xmm14 and xmm15 registers, the runtime library uses Intel's RdRand instruction which returns a random number from a hardware-based entropy source.

To support multi-threading in a SaVioR-enabled application, SaVioR creates multiple stacks and initializes dedicated registers to hold per-thread metadata (i.e., the SPRB, random number, and AES secret key) when a new thread is created. Note that our implementation is thread-safe because each per-thread metadata is allocated independently and stored in per-thread hardware registers (i.e., r15, xmm14, and xmm15). In detail, the LLVM compiler replaces pthread_create calls with slr_pthread_create to perform a procedure similar to that in the initialization function (constructor). It also releases the SaVioR-generated memory via pthread_cleanup_push/pop when a thread is terminated.

Finally, non-local jumps are safely handled by modifying Glibc's setjmp/longjmp implementation. The setjmp function saves the current execution context into a jmp_buf structure. When longjmp is invoked with jmp_buf as an argument, the execution context is restored from jmp_buf. Thus, the control flow goes back to the point at which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

`setjmp` was invoked. Originally, a `jmp_buf` structure contained callee-saved registers, which includes the `r15` register. However, to guarantee the confidentiality and integrity of SPRB, it should not be pushed onto memory. Therefore, we modified non-local jump implementation to not save the `r15` register in `jmp_buf`. In detail, when `longjmp` unwinds the stack, SaVioR also has to unwind the SPRB and adjust `r15`, which points to the valid top entry of SPRB. Without proper adjustment of `r15`, SPRB still holds the invalid entries that point the unwound stack frame after the stack unwinding has ended. Hence, SaVioR removes the entries that have lower addresses than the current stack pointer because the stack always grows downward, and thus, the value of the caller's stack pointer is always higher than that of the callee. (The comparisons are done with a random-bit masked value of the stack pointers.)

5.4 Compatibility

Given that it is common for an application to be linked with (uninstrumented) third-party libraries, binary compatibility and modularity support are key features needed for wide adoption. In short, our approach achieves full binary compatibility with uninstrumented libraries, but limited modularity support.

Binary Compatibility. A SaVioR-enabled application can invoke uninstrumented functions in a legacy library with randomized variables as arguments without performance penalties. Likewise, an uninstrumented code can invoke VBI and SFR-protected functions.

Modularity Support. The VBI scheme is applied to the function's prologue and does not require a whole program analysis. Therefore, if only VBI is applied, it supports modularity. For example, it does not require a main executable to be re-compiled when a new VBI-enabled library is linked. In contrast, because the SFR scheme is applied to the call-site and requires a whole program analysis, it does not support modularity. For example, assume that a main executable is linked with a new SaVioR-enabled library without recompilation. In this case, it can still invoke a SFR-protected function in the new library; however, the stack frame is not randomized.

Assembly code. The implementation of SaVioR is based on the LLVM compiler and thus cannot handle hand-written assembly (e.g., inline assembly or assembly files). For example, there are several assembly codes in glibc (e.g., setjmp/longjmp, getcontext/setcontext, and multiarch files) Currently, we only support non-local jumps by redirecting the invocation of `setjmp/longjmp` to the safe counterparts (i.e., `slr_setjmp/slr_longjmp`). However, with additional engineering effort, the remaining hand-written assembly code can be handled in a similar manner.

5.5 Library Integration Level

To integrate third-party libraries into a SaVioR-enabled environment incrementally, we provide three library integration levels: (1) protected, (2) aware, and (3) unprotected.

- **Protect:** We call a library hardened with SaVioR a SaVioR-protected library. This instrumented library can enjoy the security guarantees provided by SaVioR.
- **Aware:** We call a library that is aware of which registers are used for SaVioR-related secrets a SaVioR-aware library. While this library is not compiled with SaVioR and hence enjoys none of the memory safety guarantees, it ensures that the dedicated registers are never spilled into memory or inadvertently accessed. An example of such a library is glibc, which currently is not yet compiled with Clang/LLVM. Therefore, we compiled glibc using the gcc compiler with the `-ffixed` options (i.e., `-ffixed-r15`, `-ffixed-xmm14`, and `-ffixed-xmm15`) to achieve SaVioR awareness.
- **Unprotect:** This library is only compatible with a SaVioR-protected application, but there is no guarantee that the dedicated registers are never spilled into memory.

6 Evaluation

We evaluated the performance impact of SaVioR on the SPEC CPU2006 and PARSEC 3.0 benchmark suites and real-world applications, including Nginx and PHP web servers, Memcached server, and p7zip. All experiments were performed on an Intel i7-9700K CPU at 3.6 GHz and 32GB of RAM, running Ubuntu 18.04 with Linux kernel version 5.0.21. We then evaluated the effectiveness of SaVioR using a statistical security analysis and performed an empirical security analysis against several common vulnerabilities and exposures (CVEs).

6.1 Runtime overheads

6.1.1 SPEC CPU2006 Integer and Floating Point

Runtime overhead will be introduced by our VBI and SFR instrumentations because they insert hardening instructions and decrease the locality of references on the stack. To quantify this performance degradation, we evaluated SaVioR using all the C/C++ programs of the SPEC CPU2006 benchmark suite. This benchmark suite contains a set of applications that are considered to be representative of a wide range of real-world applications, e.g., a Perl interpreter and bzip2. The benchmarks include both compute-bound and memory-bound workloads; this is suitable for showing an example of the worst-case overhead introduced by SaVioR. We ran each benchmark test using its reference inputs and measured the average runtime over five executions. The baseline was compiled with `-O3` optimization and LTO enabled.

We ran all experiments with 1024 stacks and a maximum random padding size of 1024 bytes because it provides a reasonable degree of unpredictability in the stack layout. Currently, to run with these parameter settings on Intel `x86_64` processors, a SaVioR-protected application only has a 37-bit virtual address space for the regular region. In such an environment, the process may crash unless the entropy of ASLR is lowered or ASLR is turned off. Hence, we turned off ASLR when running an application whose bits available for the regular region were less than 42.

We summarize our SPEC measurements in Figure 5. Overall, SaVioR has a geometric mean of 14.9% performance overhead. Besides, to accurately compare the results with

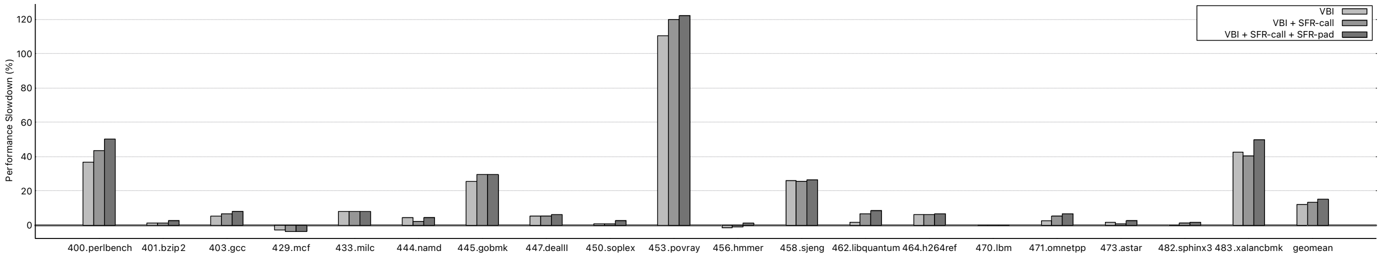
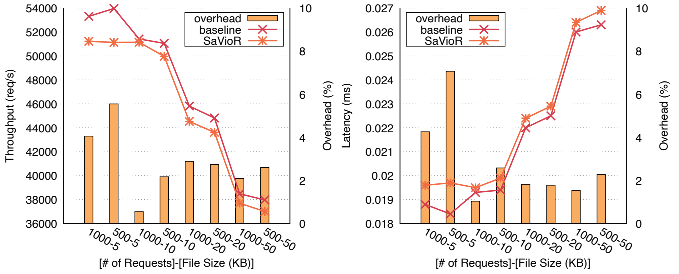


Fig. 5: SaVioR's performance overhead on SPEC CPU2006 Integer and Floating Point benchmarks with 1024 stacks. We measured the runtime overheads with VBI, VBI + SFR-call, and VBI + SFR-call + SFR-pad independently. SFR-call randomizes the stack frame without introducing random padding, and SFR-pad inserts random padding up to 1024 bytes.



(a) Throughput **(b) Latency**

Fig. 6: Performance impact on Nginx

StackArmor, we have analyzed result from the C subset of SPEC CPU2006 benchmarks specifically supported by StackArmor. We found that SaVioR introduces a 10% runtime overhead compared to the 22% introduced by StackArmor (note that zero initialization overhead is not included), which is approximately half of the overhead of StackArmor with 1024 stack frames. The results show that 5 out of 12 benchmarks, `400.perlbench`, `445.gobmk`, `453.povray`, `458.sjeng`, and `483.xalancbmk`, incur significant runtime overhead. `400.perlbench`, `458.sjeng` and `483.xalancbmk` intensively allocate/deallocate stack-resident objects and access these objects much more frequently than the other benchmarks. Interestingly, although `453.povray` suffers the highest performance overhead (around 122%), the stack is rarely used. Instead, `453.povray` frequently executes direct and indirect function calls and the IPC is substantially decreased from 2.43 to 1.55 (-36%) by the massive number of SFR calls. In addition, this benchmark has the highest increase in the page fault rate (×146) because the original one has the smallest memory footprint, and our solution increases that footprint substantially. In the case of `455.gobmk`, it is the most compute-bound benchmark in this benchmark suite and its results are influenced by the number of executed instructions, which is much higher than those of the other benchmarks.

Likewise, as in `453.povray` benchmark, `400.perlbench` and `483.xalancbmk` [3] suffer huge performance penalties, 44% and 50%, respectively. Both benchmarks intensively use indirect function calls. The current implementation of SaVioR finds the targets of indirect calls based on type analysis and applies SFR to the call site if the type of the arguments of the indirect call site are the same as one of the VBI- and SFR-protected functions. Hence, an SFR-applied indirect call site could execute the SFR hardening instruction even when calling an unprotected (safe) function. Instead of type-based analysis, point-to analysis could be used to improve the performance of SaVioR in this case.

Benchmark	1-Thread	2-Thread	4-Thread	8-Thread
PARSEC 3.0	7.81%	11.00%	11.89%	11.83%
p7zip 16.02	3.91%	11.50%	10.69%	9.93%
Memcached 1.6.8	0.42%	0.48%	0.47%	0.66%

TABLE 2: Performance overhead on 1, 2, 4, and 8 threads

Surprisingly, two of the benchmarks (`429.mcf`, `456.hmmer`) perform better than the original ones. These benchmarks are memory-intensive, and a substantial portion of their execution time is spent executing loops that continuously access large arrays on the *heap*. SaVioR randomly relocates and spread out vulnerable buffers and address-taken variables, but reorders the rest of the safe variables to be collected and become adjacent to the return address and callee-saved register. This may increase the locality of frequently accessed objects on the stack. In contrast, although `458.sjeng` is also a memory-intensive benchmark, the frequently accessed large arrays are allocated on the *stack* and thus considerable runtime overhead is added (25%).

6.1.2 I/O-intensive Web Server

To evaluate the efficiency of SaVioR on I/O-intensive applications, we have measured the throughput and latency of a SaVioR-protected Nginx web server using Apache HTTP server benchmarking tool. The benchmark simulates 8 workers receiving 500–1000 `Keep-Alive` requests of a file 5 to 50 KB in size. Our setting resembles the common usage [38] and the whole procedure is repeated 10 times. The result is shown in Figure 6, which reveals that the performance degradation imposed by SaVioR (<7%) is acceptable.

6.2 Scalability

We also evaluated the scalability of SaVioR on the PARSEC 3.0 multithreaded benchmark suite as well as real-world applications such as p7zip 16.02 and Memcached 1.6.8. All experiments were performed on an Intel i7-9700K 8-core

[3] As described in Section 5.1, VBI can use the bits of the random number remaining after SFR has been applied. However, if applying VBI only, it has to generate new random number. Hence, in the Xalancbmk benchmark, the overhead of the VBI-only one is comparable with that of full protection.

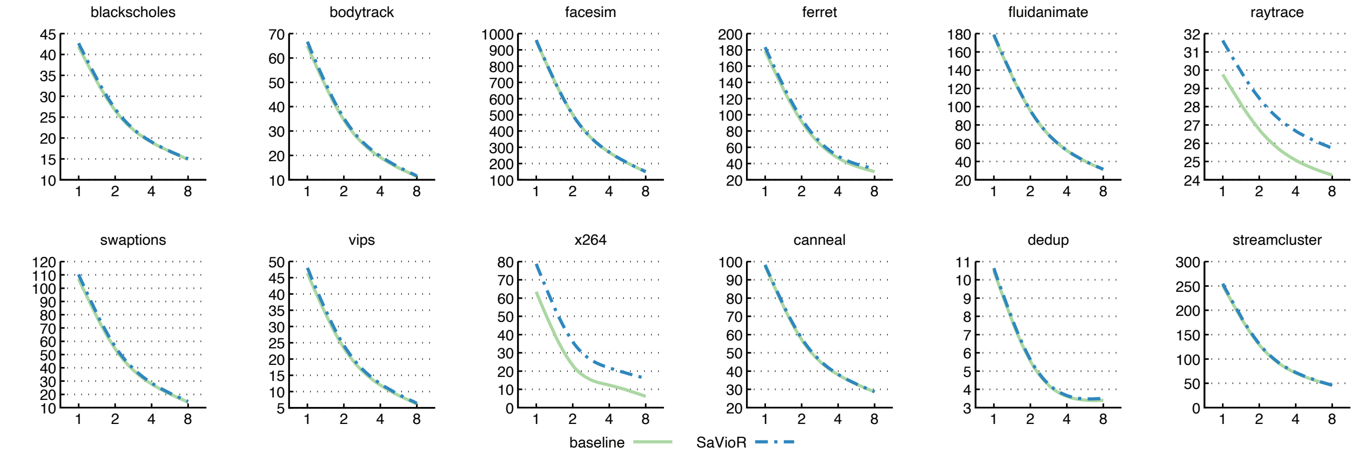


Fig. 7: Performance overhead on PARSEC 3.0

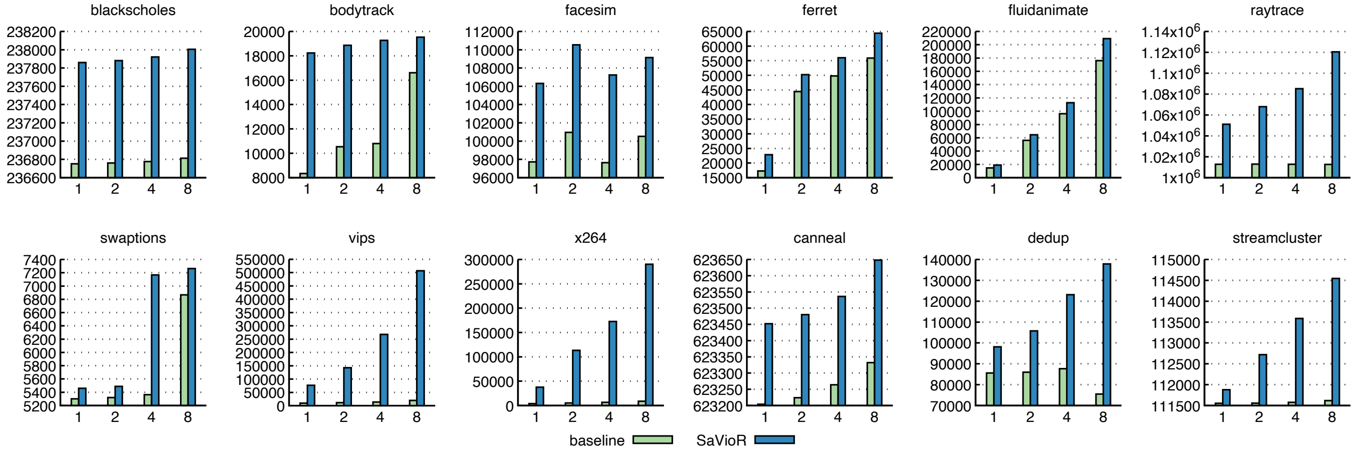


Fig. 8: Memory overhead on PARSEC 3.0 (KB)

processor. On the processor, up to 8 threads can run simultaneously without interfering with each other. Therefore, we measured the performance overhead of all benchmarks on 1, 2, 4, and 8 threads, as summarized in Table 2.

The experiment results on PARSEC 3.0 are shown in Figure 7 and Figure 8 for 12 out of the 13 benchmarks. [4] For memory overhead, we measured the maximum resident set size (RSS) for the tested benchmarks. The geometric mean of the memory overhead with 8 threads is 96.5%, as shown in Figure 8. Because SaVioR randomly relocates stack-resident objects into multiple stacks, it increases the memory footprint overhead. Despite this high memory overhead, it has moderate performance overheads on the PARSEC 3.0 benchmark suite. The geometric mean of performance overhead with 8 threads incurred by SaVioR is 11.8%. As shown in Figure 7, SaVioR scales fairly well with multi-threaded applications.

The p7zip benchmark includes LZMA compression and decompression tests, which represent CPU- and memory-intensive workloads. We executed the integrated benchmark of p7zip with the default settings. The result showed a pattern similar to that of the PARSEC benchmark and an average runtime overhead of 9.9% for 8-threads. To benchmark Memcached (which is a key-value storage server and is I/O intensive), we selected the memcslap benchmark tool [39] to issue a 9:1 ratio of 100,000 get/set operations and measure the time taken to load/store the requested data. The concurrency level of the benchmark matched the number of server threads. The increases in the measured data load times were under 1%, which indicates that the impact of SaVioR on I/O-intensive multithreaded server applications is negligible.

6.3 Uninitialized-read Detection

We evaluated Clang's `-Wuninitialized` option and our detection tool against a regression test suite of the `-Wuninitialized` option in the Clang source code (i.e., clang/testsuite/Sema/uninit-*). This test suite includes 70 test cases, i.e., 40 positive and 30 negative cases. In the case of Clang, as expected, the results included neither false positives nor false negatives. In contrast, our tool obtained false

[4] The remaining benchmark (`freqmine`) utilizes OpenMP directives (e.g., `#pragma omp parallel`) to execute a code block in a parallel manner; when a compiler encounters such a directive during compile-time, it generates an outlined code block that internally invokes `pthread_create` to execute it concurrently. Unfortunately, our current implementation of SaVioR cannot handle an outlined OpenMP code, that is, it cannot substitute `pthread_create` with `slr_pthread_create`.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

```
struct mystruct {
    int a;
    int b;
} st;


st.a = 0xdeadbeef;
x = st.b;
```

(a) field-sensitive

```
struct mystruct {
    int a;
    int b;
} st;


printf("%d", st.a);
st.a = 0xdeadbeef;
```

(b) flow-sensitive

```
struct mystruct {
    int a;
    int b;
    char c;
} st; // 12 bytes

st.c ='a';
x = *((char*)&st+10);
```

(c) byte-sensitive

```
int a;

*((char *)&a) = 'c';
x = a;
```

(d) byte-sensitive for primitive types

Fig. 9: Details of the sensitivity test suite. (a) Field-sensitive test, (b) flow-sensitive test, and (c) byte-sensitive test for aggregate types, (d) Byte-sensitive test for primitive types.

positives in 11 negative cases, but no false negatives. (Note that these false positives do not weaken our protection. Instead, they increase the number of SFR-protected functions, but may harm performance.) Three of them occurred because of a non-standard C extension (i.e., `__block`). Five of them occurred because of unrealistic coding in real cases. For instance, they include a self-initialization (e.g., `int x = x;`) and an uninitialized read in an infinite loop (e.g., `unsigned long flag; for(;;) (void) flag;`), cases for which Clang intentionally suppresses warnings. (Moreover, such cases are optimized out or replaced with a unreachable instruction in the optimization phases.) Lastly, three of them occurred because the test cases assume that a function that takes as an argument the address of a variable will initialize that variable (e.g., `int a; use(&a);`), whereas our tool considers such a function to be vulnerable.

In addition, we also ran our detection tools against a test case for Clang's `-Wsometimes-uninitialized` (i.e., clang/test/analysis/uninit-sometimes.cpp). This option is equivalent to gcc's `-Wmaybe-uninitialized` option, which performs a more aggressive analysis than the `-Wuninitialized` option, and this could lead to a high number of false positives. This test suite includes 30 test cases: 25 positive and 5 negative. In this case, Clang yields no false positives and two false negatives. (Two false negative cases serve as a to-do list.) Our tool produces no false negatives, but does not pass all negative cases. Because our detection analysis is path-insensitive, it assumes that all paths are feasible and thus has poor precision, resulting in a high number of false positives. However, we note again that these false positives do not reduce the security guarantees provided by SaVioR.

The above testing demonstrates that our uninitialized-read detection tool is comparable to Clang's `-Wuninitialized` option. However, the test suite is tailored to the detection of the uninitialized use of primitive types (e.g., int or double), and thus does not guarantee that our detection tool performs field-sensitive, flow-sensitive, and byte-sensitive analyses for aggregate types (e.g., struct, array, and union). Therefore, we manually created a test suite in which a test case checks whether the analysis of a given tool is field-sensitive, flow-sensitive, and byte-sensitive. For example, as shown in Figure 9c, the use of a partially uninitialized object is detected if a given tool is byte-sensitive.

	field-sensitive	flow-sensitive	byte-sensitive
gcc (<6)	✗	✓	✗
gcc (≥6)	✓	✓	✓
Clang	✗	✗	✗
SaVioR	✓	✓	✓

TABLE 3: Comparison of the sensitivity of various compilers when handling aggregate types.

We tested gcc-5, gcc-6, Clang, and SaVioR on this test suite. The results are summarized in Table 3. Although this test suite does not contain any conditional statements (e.g., if, for, or while), our tool can handle such test cases. We model all kinds of memory objects such as primitive types and aggregate types at *byte-granularity*. For example, as shown in Figure 9d, our tool models a primitive type variable (i.e., a 4-byte integer) as a byte sequence and thus can detect the use of the partially initialized integer variable. Likewise, our tool simply applies the byte-level memory model to aggregate types and thus considers any aggregate type variable to be a byte sequence. That is, the primitive and the aggregate types are handled in the same manner by our tool. Because our tool successfully passes the test suites for the primitive type, which contains the complex cases with conditional statements, we expect that our tool will work properly for aggregate types in such test cases.

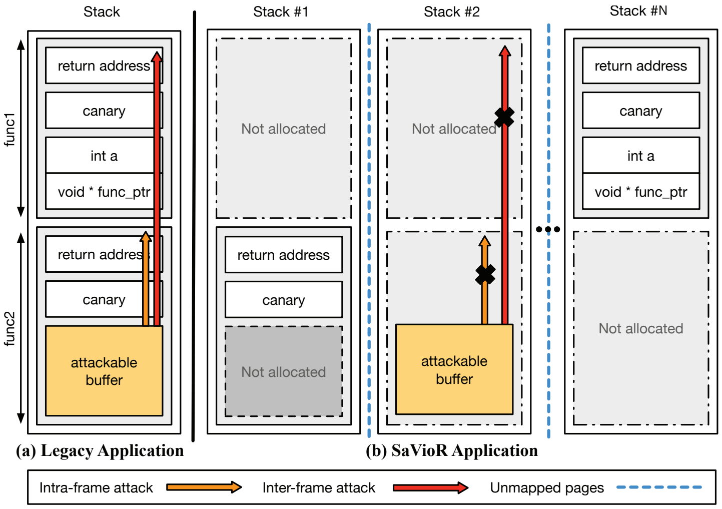
6.4 Security Analysis

To evaluate the effectiveness of SaVioR, we performed statistical and empirical security analyses. The statistical analysis indicates how secure SaVioR is in theory. This analyzes the probability that an attacker can mount a successful memory corruption attack. In contrast, the empirical analysis presents how secure SaVioR is in practice. We applied the SaVioR defense against a set of CVEs in two real-world applications, Nginx and PHP web servers.

6.4.1 *Statistical Security Analysis*

Because SaVioR is a comprehensive and probabilistic protection that mitigates all kinds of spatial and temporal memory corruption attacks, we show the probability that successful memory corruption attacks can be mounted for spatial and temporal memory errors independently.

Let N be the number of deployed stacks and let P be the maximum size of the random padding. We configured the values of N and P to 1024 and 1024, respectively. This



(a) Legacy Application

(b) SaVioR Application

Fig. 10: Spatial attack

configuration is consistent with that of the performance evaluation. Because the stack pointer must be aligned on a 16-byte boundary in 86_64, the effective entropy is $P/16$. In addition, we assume that an attacker has full knowledge of our defense semantics.

Spatial Attacks. If an attacker misusing attackable variables corrupts or leaks the intended data exactly, we consider the spatial attack to be successful. We classify spatial attacks into four categories: 1) contiguous intra-frame, 2) contiguous inter-frame, 3) non-contiguous intra-frame, and 4) non-contiguous inter-frame attacks.

Figure 10a illustrates two spatial attacks on legacy applications: intra-frame and inter-frame attacks. Intra-frame attacks (highlighted in orange) can be mounted if an attackable buffer and a target variable are located in the same stack frame. In contrast, inter-frame attacks (highlighted in red) can be mounted if an attackable buffer and a target variable are located in the different stack frames. There are two ways to mount spatial attacks: contiguously and non-contiguously. Contiguous spatial attacks corrupt the target object while corrupting all unintended memory that ranges from the end of the attackable object (i.e., a buffer) to the intended object (e.g., the return address). One example of a contiguous spatial attack is the traditional buffer overflow attack. On the contrary, non-contiguous spatial attacks could corrupt the desired object only without corrupting other objects unnecessarily. Non-contiguous attacks are more severe than contiguous ones, which, for example, trivially bypass the stack canary protection.

With respect to contiguous intra-frame attacks, because VBI always isolates vulnerable variables from its stack frame, the attacks cannot corrupt the target object such as a return address. As shown in Figure 10b, the attackable buffer is separated from its stack frame (Stack 1). It is relocated to Stack 2, which prevents intra-frame attacks from corrupting stack objects in func2's stack frame. The contiguous inter-frame attacks can be prevented by SaVioR as well. For example, in Figure 10b, assume that an attacker attempts to corrupt func_ptr in func1's stack frame by abusing attackable buffer. With SaVioR, attackable buffer and func_ptr are separately placed in Stack 2 and Stack N, which prevents this contiguous inter-frame attack. In particular, this attack will inevitably touch a non-accessible page (i.e., an unmapped or guard page) and will cause the victim program to crash.

On the other hand, non-contiguous attacks could corrupt target objects in other stack regions beyond its own stack region, which allows the attacker to perform more advanced attacks. That is, the non-contiguous attack can bypass the defensive facilities (e.g., canary and guard page) and corrupt the desirable objects. However, in the SaVioR-applied application, the non-contiguous attack can also be hindered due to the unpredictability added to the stack objects. Even if the attacker abuses a write-what-where vulnerability, she/he cannot corrupt the intended object without the exact memory layout of the stack objects, e.g., the exact distance between the vulnerable buffer and a target object.

In both the contiguous and non-contiguous attacks, the probability that an attackable variable and a target variable are located in the same stack region is $1/N$. In addition, the probability of predicting the offset between objects in a same stack frame is $16/P$. Therefore, both attacks can succeed with a probability of $16/(N \times P)$. In our evaluation, $N = 1024$ and $P = 1024$ yields a probability of 0.00001 that the attacker corrupts the intended memory.

Temporal Attacks. Figure 11 illustrates the organization of the stack layout during UR attacks in legacy and SaVioR-protected applications. Suppose that func1 calls func2, which has an attacker-controlled memory, and then func1 calls func3, which has an UR on the function pointer func_ptr. If the variable func_ptr is allocated in the memory that was previously allocated to the attacker-controlled memory and reuses that memory without proper initialization, an attacker can hijack the control flow.

For legacy applications, Figure 11a shows two organizations of the stack layout after func2 (left) and after func3 (right) have been called. By calling func2, the attacker can leave a malicious contents in attacker-controlled memory. Next, func3 dereferences the uninitialized variable func_ptr, allowing arbitrary control-flow hijacking. This example shows that because the UaF and UR attacks depend on stack frame reuse, preventing this reuse lies at the heart of SaVioR.

By applying SaVioR, the probability that a newly created stack frame is placed into the region where the previously-used stack frame was located is $1/N$. Moreover, the probability that the newly created frame starts at the same location as the previous stack frame which might contain attacker-controllable values is $16/P$ (see Figure 11b). Thus, the introduced entropy of stack frame reuse is $16/(N \times P)$. Like the case with the spatial attack, $N = 1024$ and $P = 1024$ yields a probability of 0.00001 that a temporal attack overlaps the attacker-controlled memory with a vulnerable variable.

Limitations. Note that the aforementioned odds are the probability of success in a single trial of an attack and for pinpointing (corrupting) the exact location of a target variable. If an attacker is able to conduct attacks multiple times, exploit a vulnerability using partial overwrites, or perform de-randomization attacks (e.g., stack spray), our defense can be weakened. For example, during the exploitation of uninitialized-read vulnerability, even if the attacker fails to cause a vulnerable object to reuse the memory that is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

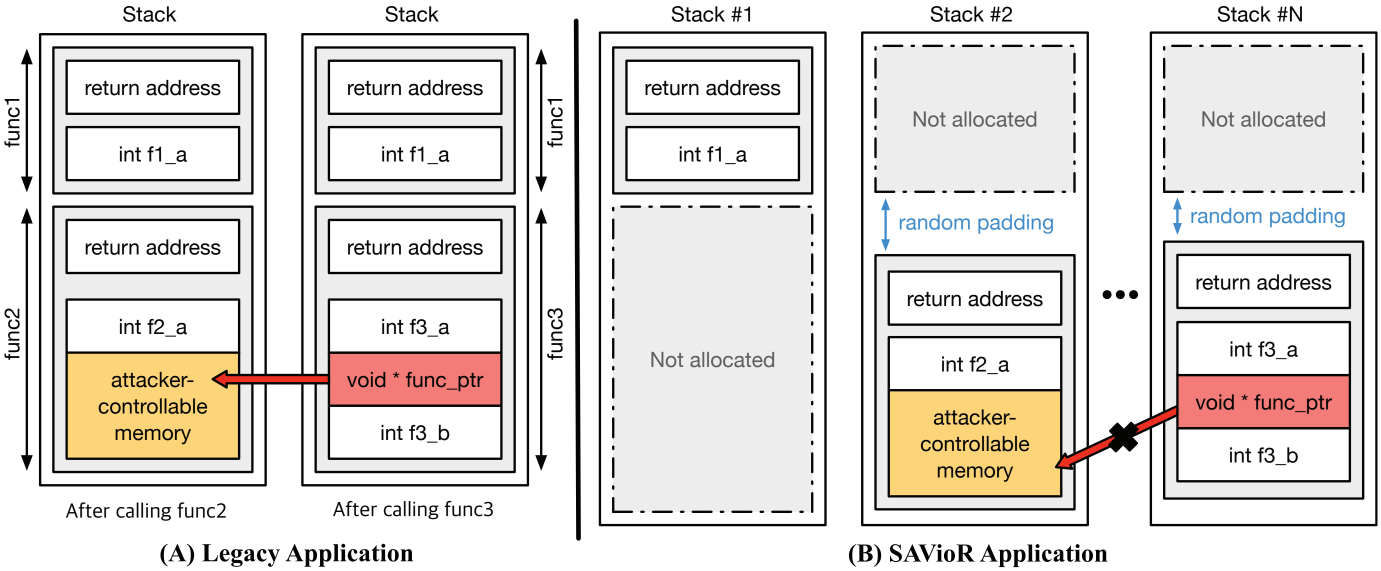


Fig. 11: Temporal attack

controlled by the attacker, the attacker has further chances to try as long as the uninitialized-read does not cause a crash.

Stack Spraying. An attacker could also mount advanced de-randomization attacks, e.g., by filling the stack with a number of malicious data. Such a strategy is essentially similar to that of heap spray. Such de-randomization attacks on the stack are referred to as stack spraying attacks [40]. In detail, to reduce the randomization entropy and bypass SaVioR's protection, an attacker could launch a stack spraying attack to fill the stack frame with a large number of malicious data in the hope that the content of memory where the uninitialized variable is located might overlap with the sprayed data.

However, contrary to heap spraying attacks, stack spraying attacks are more difficult to carry out because of the nature of the stack allocation algorithm. For a heap, an attacker typically can control the size of and number of allocated (spray) data, because the heap is explicitly managed via a heap management API such as `malloc()` or `free()`. However, in the case of stack, the stack allocation is determined at compiler time and the size of the allocated variables are usually fixed and small. Moreover, the stack is shared among all the executions in a thread and the stack's resident objects are *implicitly* allocated, deallocated, and reused on the stack. This leaves the prepared spray data open to being overwritten before the use of uninitialized value and can lead to undefined behavior (e.g., program crash).

6.4.2 Empirical Security Analysis

We performed an effectiveness evaluation on four CVEs. In our experiments, we collected proof-of-concept (PoC) exploit files for each CVE and manually confirmed the exploitability of each vulnerability when SaVioR was applied.

CVE-2013-2028 in Nginx 1.4.0. is a contiguous stack-based buffer overflow vulnerability in the Nginx web server. It triggers the integer signedness error, the value of which is used for the size read by `recv()`; then a stack-based buffer overflow attack can be mounted. In the SaVioR-enabled application, the location of the vulnerable buffer is randomly relocated in each function invocation and the attackable buffer is always separated from its original stack frame. Because of this separation, the attackable buffer is located far from the canary and return address. Hence, the intra-frame attack is hampered, that is, it cannot guess the canary and fails to overwrite the return address. Even if an attacker performs inter-frame attacks, it is difficult for him or her to predict the exact location of the canary due to the random padding and thus the attempt to mount the byte-by-byte brute force attack fails.

CVE-2019-11038 in PHP-7.3.5. is an UR vulnerability in a PHP server using libgd. The libgd library creates images based on the input XBM format. When parsing XBM format with a given bug-triggering PoC input file, if `sscanf(h,"%x",&b)` in function `gdImageCreateFromXbm` is not able to read a hex value from `h`, the `b` variable will not be properly initialized and thus contain an uninitialized value, the content of which could be controlled by the attacker. Therefore, the use of these uninitialized variables could be used to trigger other vulnerabilities. With SaVioR, it is unlikely that an uninitialized variable will reuse a memory area in which content is controlled by the attacker.

CVE-2019-9639 in PHP-7.3.2. is also an UR vulnerability. The vulnerability is caused by mishandling of the `data_len` variable, which is not initialized on declaration. In particular, `data_len` may not be initialized properly in some cases in the switch statement, including in the default case. The intra-procedural analysis of SaVioR can detect this variable and will consider it to be vulnerable to UR. Hence, SaVioR applies SFR to the stack frame that contains `data_len` and introduces the unpredictability into the stack frame reuse.

CVE-2018-1000140 in librelp 1.2.14 is non-contiguous stack-based buffer overflow vulnerability. We just analyzed the vulnerability manually to demonstrate SaVioR's defense against non-contiguous spatial attacks. The vulnerability is caused by misusing `snprintf`, which returns the size of a written value *regardless of whether the write attempt is successful or not*. Function `snprintf` continues to process the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

crafted input in a loop and under the mistaken assumption that the amount of data written to the buffer is the same as the returned value. This allows an attacker to trigger a non-contiguous buffer overflow and gives him or her the arbitrary write primitive. However, without knowledge of the exact location of 1) the attackable buffer and 2) the target variable, the attacker cannot fully abuse this arbitrary write primitive for successful exploitation. The only plausible way to bypass SaVioR and exploit the vulnerability is by combining this vulnerability with a powerful information leakage attack. Besides, SaVioR creates a leakage and reverse-engineering resilient stack layout by introducing a randomized absolute/relative distance between stack-resident objects. For example, it is difficult for an attacker to accurately distinguish whether a leaked address is a valid object's address or an invalid one (a garbage value), which prevents the attacker from fully exploiting the information leakage vulnerabilities.

7 Discussion & Limitations

7.1 Extended virtual address space

Intel recently proposed a new 5-level paging mode (LA57), which enables 56-bit user virtual address space. Some other architectures support more than 48-bit virtual addresses as well. For instance, recent SPARC processors support at least 52-bit virtual addresses. In addition, 64-bit ARM architecture supports 52-bit user virtual address space with the ARMv8.2-LVA extension. Thanks to the extended virtual address space, our SaVioR design is reasonably compatible with those architectures, and hence is not limited to x86_64.

7.2 Entropy of ASLR

To show the SaVioR's compatibility across modern operating systems, we present the entropy of ASLR implementation on OpenBSD, HardenedBSD, and Windows (x86_64). We manually analyzed the ASLR implementation on OpenBSD and HardenedBSD. For Windows, we refer to Microsoft's article [41], [42]. Note that ASLR is commonly performed at page-level granularity. Thus, we ignored the randomization entropy below the 4 KB page size. Recall that Linux offers entropies of 28, 13, 28, and 22 bits for the ELF, brk (malloc), mmap, and stack segments.

OpenBSD and HardenedBSD. OpenBSD offers entropies of 32, 20, and 6 bits for the ELF, mmap/malloc, and stack segments, providing a lower entropy than Linux. Thus, SaVioR is allowed more randomization entropy with OpenBSD. In contrast, HardenedBSD has aggressively adopted the latest security features [43] that offer ASLR entropies of 30, 30, and 33 bits for the ELF, mmap/malloc, and stack segments. In this case, to be compatible with ASLR in HardenedBSD, SaVioR must reserve a 46-bit virtual address space and thus can only deploy 1024 stacks with 5-level paging.

Windows. In default, Windows offers entropies of 17, 19, 8, and 17 bits on the ELF, DLL, heap, and stack segments, which does not fully take advantage of the huge 64-bit virtual address space available. To tackle this problem, High Entropy ASLR (HEASLR) was introduced and offers entropies of 17, 19, 24, and 33 bits for each respective segment. Interestingly, except for the stack, Linux provides more ASLR entropy than Windows with HEASLR, albeit it is an improvement. In HEASLR-enabled applications, SaVioR also has to reserve 46-bit virtual address space and can deploy 1024 stacks with 5-level paging, which introduces a reasonable degree of unpredictability into the stack layout.

7.3 Intel Control-flow Enforcement Technology

Intel Control-flow Enforcement Technology (CET) provides two techniques to protect the integrity of forward-edge and backward-edge control-flow: (1) a coarse-grained forward-edge CFI policy ensuring that an indirect function call only can transfer control flow to the function entry points (i.e., label-based CFI); and (2) backward-edge protection using a hardware-enforced shadow stack. In detail, for the forward-edge protection, Intel CET introduces a new instruction (ENDBRANCH), which is placed into function entry points and is used to mark valid branch targets. For the backward-edge protection, it changes the semantics of call/ret instructions to store and retrieve the return address into and from the hardware-isolated shadow stack.

However, forward-edge protection is still vulnerable to recent CFI bypass attacks [44] because of its coarse-grained policy. To overcome this limitation, Intel CET can be combined with SaVioR to raise the bar against function pointer corruption attacks by randomizing the location of function pointers on the stack. Backward-edge protection only guarantees the integrity of the return addresses, which does not include non-control and other control data (e.g., function pointers) on the stack. Further, because it changes the semantics of the call/ret instruction, it is tailored to the protection of return addresses. Therefore, it is not easy to naively repurpose this technique to protect other data. This limitation can be overcome by adopting SaVioR to protect all the non-control and control data comprehensively.

7.4 Comparison with StackArmor

StackArmor provides comprehensive stack protection similar to SaVioR and demonstrates the practical feasibility of its protection for legacy/COTS binaries. In detail, StackArmor is built on top of a binary writer and thus can be directly applied to legacy binaries. In contrast, SaVioR, which relies on a compiler, needs to recompile the source code to apply protection. The differences in the deployed instrumentation tool, i.e., the binary rewriter and compiler, lead them to have distinct strengths and weaknesses. We argue that both approaches are complementary, and the choice of which defense to leverage depends on the ease of recompilation and accessibility to the source code.

When legacy binaries cannot be recompiled and redeployed, StackArmor is more suitable than SaVioR in terms of deployment. However, the implementation of StackArmor is based on PEBIL [45], which assumes the presence of relocation information and debugging symbols. Unfortunately, given that modern compilers remove these symbols by default, stripped binaries are quite common in real-world cases. In that case, there is no alternative way for StackArmor to apply its protection to the legacy binaries.

The authors of StackArmor recognized this problem and thus proposed DynInst [46], which relaxes these assumptions, as an alternative. However, DynInst's static instrumentation incurs an average 35% increase in performance overhead compared to that of PEBIL. Given that StackArmor already yields a significant performance overhead (28%) on SPEC CPU2006, an implementation based on DynInst will incur an even more substantial performance overhead, which makes it impractical.

Although StackArmor attempts to address an important problem, SaVioR still has clear advantages over StackArmor in terms of performance and robustness when the source code is available. Moreover, SaVioR provides more practical and secure protection than StackArmor. As shown in Section 6.1, SaVioR yields a lower performance overhead (10%) than StackArmor (28%). Furthermore, SaVioR provides a leakage-resilient way to protect security-sensitive metadata; SaVioR guarantees that there is no reference to sensitive metadata except for the dedicated registers (i.e., `r15`, `xmm14`, and `xmm15`) and these registers are not spilled into memory.

In contrast to SaVioR, to access its security-sensitive metadata, StackArmor relies on PEBIL's functionality, which allows a developer to instrument the access to Thread-Local Storage (TLS). However, except for the main thread, the thread control block (TCB), which points to its TLS, is located on the thread stack. (In the case of the main thread, its TCB is created by a dynamic loader using mmap() and thus is also placed in the mmaped region.) Therefore, this approach leaves a reference to security-sensitive metadata in memory, leaving the metadata open to information leakage attacks.

8 Conclusion

We presented SaVioR, which introduces unpredictability into the stack layout and provides comprehensive protection against stack-based spatial and temporal memory corruption attacks for C/C++ applications. To achieve this goal, it leverages the huge 64-bit virtual address space and pointer mirroring to efficiently randomize the location of stack-resident objects at each function invocation. Our statistical and empirical security evaluation verified that SaVioR effectively mitigates spatial and temporal attacks with high probability. Performance and compatibility were evaluated by porting SaVioR to the SPEC CPU2006 and PARSEC 3.0 benchmark suites as well as real-world applications including the Nginx and PHP web server. The evaluation showed that our approach is feasible and practical enough to be adopted by various architectures and operating systems.

Acknowledgment

The authors would like to thank all the anonymous reviewers for their valuable insights and feedback that helped us improve our paper. This work was supported by the Office of Naval Research (ONR) through award N00014-18-1-2661. This work also was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. IITP-2019-0-01570, IITP-2019-0-01343, and IITP-2020-0-01840) and National Research Foundation (NRF) grant funded by Korea government (MSIT) (No. NRF-2020R1F1A1058305). Jinsoo Jang and Brent Byunghoon Kang are corresponding authors.

References

[1] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[2] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, 1998.

[3] H. Etoh and K. Yoda, "Propolice: Improved stack-smashing attack detection," 2001.

[4] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with paced canaries," in *Proc. of the 4th Workshop on System Software for Trusted Execution*, 2019, pp. 1–6.

[5] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks," in *Proc. of the 31st Annual Computer Security Applications Conf.*, 2015, pp. 351–360.

[6] J. Sun, X. Zhou, W. Shen, Y. Zhou, and K. Ren, "Pesc: A per system-call stack canary design for linux kernel," in *Proc. of the Tenth ACM Conf. on Data and Application Security and Privacy*, 2020.

[7] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proc. of the Second European Workshop on System Security*, 2009.

[8] T.-c. Chiueh and F.-H. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Proc. 21st International Conf. on Distributed Computing Systems*. IEEE, 2001.

[9] A. Baratloo, N. Singh, T. K. Tsai *et al.*, "Transparent run-time defense against stack-smashing attacks." in *USENIX Annual Technical Conf., General Track*, 2000.

[10] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. of the 6th ACM Symp. on Information, Computer and Communications Security*, 2011.

[11] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. of the 10th ACM Symp. on Information, Computer and Communications Security*, 2015.

[12] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *IEEE Symp. on Security and Privacy*, 2019.

[13] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018.

[14] P. Zieris and J. Horsch, "A leak-resilient dual stack scheme for backward-edge control-flow integrity," in *Proc. of the 2018 on Asia Conf. on Computer and Communications Security*, 2018.

[15] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proc. of the 22nd ACM Conf. on Computer and Communications Security*, 2015.

[16] P. Team, "Pax address space layout randomization (aslr)," 2003.

[17] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proc. of Sixth Workshop on Hot Topics in Operating Systems*, 1997.

[18] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *USENIX Security Symposium*, 2005.

[19] M. T. Aga and T. Austin, "Smokestack: thwarting dop attacks with runtime stack layout randomization," in *IEEE/ACM International Symp. on Code Generation and Optimization*, 2019.

[20] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *USENIX Security Symposium*, 2003.

[21] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS*, 2015.

[22] H. Marco-Gisbert and I. Ripoll, "Preventing brute force attacks against stack canary protection on networking servers," in *IEEE 12th International Symp. on Network Computing and Applications*, 2013, pp. 243–250.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012.

[24] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *22nd Annual Computer Security Applications Conf.*, 2006.

[25] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proc. of the 7th Symp. on Operating systems design and implementation*, 2006.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3063843, IEEE Transactions on Dependable and Secure Computing

[26] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in *Proc. of the 2012 International Symp. on Software Testing and Analysis*, 2012.

[27] “Automatic variable initialization,” 2018. [Online]. Available: https://reviews.llvm.org/D54604?id=174471

[28] G. Novark, E. D. Berger, and B. G. Zorn, “Exterminator: Automatically correcting memory errors with high probability,” in *Proc. of the 28th ACM Conf. on Programming Language Design and Implementation*, 2007.

[29] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, “Why nothing matters: the impact of zeroing,” *Acm Sigplan Notices*, 2011.

[30] G. J. Duck, R. H. Yap, and L. Cavallaro, “Stack bounds protection with low fat pointers.” in *NDSS*, 2017.

[31] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, “Delta pointers: Buffer overflow checks without the checks,” in *Proc. of the Thirteenth EuroSys Conf.*, 2018.

[32] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.” in *USENIX Security Symposium*, 2009.

[33] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, “Sgxbounds: Memory safety for shielded execution,” in *Proce. of the 12th European Conf. on Computer Systems*, 2017.

[34] Intel, “5-level paging and 5-level ept,” 2016.

[35] K. Lu, C. Song, T. Kim, and W. Lee, “Unisan: Proactive kernel memory initialization to eliminate data leakages,” in *Proc. of the 2016 ACM Conf. on Computer and Communications Security*, 2016.

[36] “arm64: Move elf_et_dyn_base to 4gb / 4mb,” 2017. [Online]. Available: https://patchwork.kernel.org/patch/9807319/

[37] “binfmt_elf: Use elf_et_dyn_base only for pie,” 2017. [Online]. Available: https://patchwork.kernel.org/patch/9807325/

[38] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, “Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation,” *Proc. of the ACM Conf. on Computer and Communications Security*, 2015.

[39] M. Zhuang and B. Aker, “memslap - load testing and benchmarking tool for memcached,” 2010. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man1/memcslap.1.html

[40] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, “Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying.” in *NDSS*, 2017.

[41] M. Miller, “Software defense: mitigating common exploitation techniques,” 2013.

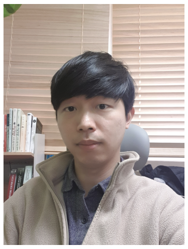
[42] K. Johnson and M. Miller, “Exploit mitigation improvements in windows 8,” *Black hat USA*, 2012.

[43] HardenedBSD, “Easy feature comparison.” [Online]. Available: https://hardenedbsd.org/content/easy-feature-comparison

[44] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity,” in *IEEE Symp. on Security and Privacy*, 2014.

[45] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “Pebil: Efficient static binary instrumentation for linux,” in *2010 IEEE International Symp. on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010.

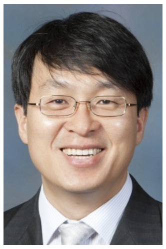
[46] A. R. Bernat and B. P. Miller, “Anywhere, any-time binary instrumentation,” in *Proc. of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011, pp. 9–16.



Seongman Lee received the B.S. degree in Computer Science from Chungnam National University in 2015. He also received the M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2017. He is currently pursing his Ph.D. at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests are software exploit mitigation and intra-kernel/process privilege separation.

Hyeonwoo Kang received the B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2019. He also received the M.S. degree in Information Security from KAIST in 2021. He is currently working toward the Ph.D. degree at KAIST Graduate School of Information Security. His research interest includes memory safety, operating systems, and trusted execution environments.

Jinsoo Jang received the B.S. degree from Ajou University and the M.S. and Ph.D. degrees in information security from the Korea Advanced Institute of Science and Technology (KAIST). He is currently an Assistant Professor with the Department of Computer Science and Engineering, Chungnam National University (CNU). He has been working on systems security areas, particularly in hardening the trusted execution environment (TEE) and leveraging general hardware features to build various defensive measures.



Brent Byunghoon Kang received the B.S. degree from Seoul National University, the M.S. degree from the University of Maryland, College Park, and the Ph.D. degree in computer science from the University of California at Berkeley. He is currently an Associate Professor with the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST). Before KAIST, he has been with George Mason University as an Associate Professor. He has been working on systems security area including botnet defense, OS kernel integrity monitors, trusted execution environment, and hardware assisted security. He is currently a member of the USENIX and the ACM.