ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection

Haehyun Cho
Soongsil University
Republic of Korea
haehyun@ssu.ac.kr

Jinbum Park
Samsung Research
Republic of Korea
jinb.park@samsung.com

Adam Oest
PayPal
USA
aoest@paypal.com

Tiffany Bao
Arizona State University
USA
tbao@asu.edu

Ruoyu Wang
Arizona State University
USA
fishw@asu.edu

Yan Shoshitaishvili
Arizona State University
USA
yans@asu.edu

Adam Doupé
Arizona State University
USA
doupe@asu.edu

Gail-Joon Ahn
Arizona State University and
Samsung Research
USA
gahn@asu.edu

ABSTRACT

Temporal memory safety violations, such as use-after-free (UAF) vulnerabilities, are a critical security issue for software written in memory-unsafe languages such as C and C++.

In this paper, we introduce ViK, a novel, lightweight, and widely applicable runtime defense that can protect both operating system (OS) kernels and user-space applications against temporal memory safety violations. ViK performs *object ID inspection*, where it assigns a random identifier to every allocated object and stores the identifier in the *unused bits* of the corresponding pointer. When the pointer is used, ViK inspects the value of a pointer before dereferencing, ensuring that the pointer still references the original object. To the best of our knowledge, this is the first mitigation against temporal memory safety violations that scales to OS kernels. We evaluated the software prototype of ViK on Android and Linux kernels and observed runtime overhead of around 20%. Also, we evaluated a hardware-assisted prototype of ViK on Android kernel, where the runtime overhead was as low as 2%.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; **Software security engineering**.

KEYWORDS

Temporal Memory Safety Violations, Operating System Kernels

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507780

ACM Reference Format:
Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507780

1 INTRODUCTION

Ideally, temporal memory safety violations, such as use-after-free (UAF) and double-free vulnerabilities would be automatically found during software development and testing. However, the spatial separation of allocation and deallocation code as well as the size and complexity of software have kept automated, effective detection techniques from working. As a result, recent research has been looking for approaches to *preventing the exploitation* of the vulnerabilities. Recent research has aimed to forbid unsafe memory reuse [6, 7, 12, 26, 31, 35], validate memory accesses and detect the use of dangling pointers [23, 25, 32], or prevent the creation of dangling pointers [11, 18, 20, 30, 34, 38] *at runtime*. These solutions show promise in mitigating against temporal memory violations. However, none of them are sufficiently scalable to be used in modern OS kernels, as they introduce high performance and/or memory overhead or require invasive changes of the memory management systems.

In this paper, we propose ViK, a new access validation approach against exploits of temporal memory safety vulnerabilities in OS kernels and user-space applications. The core idea backing ViK is *Object ID inspection*: ViK assigns a random ID to every allocated object and stores it in the *unused bits* of the corresponding pointer value (virtual address). While the operating system or the user-space program is executing, ViK inspects the pointer value before each dereference and ensures that the pointer value references the original object for which it was created. ViK also inspects the pointer value before deallocating the object. This design endows

ViK with advantages in false positive/negative rates, memory usage, and runtime overhead. First, by storing metadata in pointer values, ViK maintains low memory overhead (averaging 9% on for SPEC CPU 2006, lower than any mitigation approaches). Second, ViK is thread-safe (and thus, can scale to OS kernels) because it does not manipulate shared data structures in memory. Third, by not relying on data-flow analysis or pointer tracking for identifying the location of pointers, ViK does not have false negatives that stem from type-unsafe pointers or pointer values temporarily stored in registers or on the stack, whereas existing solutions may introduce such errors and fail to mitigate exploits because of them [6, 12, 18, 30, 34].

Runtime overhead determines the practicality of temporal memory safety defenses. To minimize the runtime overhead caused by pointer inspections, ViK employs a sound static analysis to exclude safe pointer dereferences and only protects potentially unsafe ones. In the experiment on SPEC CPU 2006, ViK exhibits runtime overhead of about 10% on average, which is similar to a userspace-specific state-of-the-art UAF defense, MarkUs [6], but higher than another userspace-specific mitigation, FFmalloc [35]. The evaluation also shows that ViK-protected OS kernels have overall 20% system performance overhead on both Linux kernel and Android kernel. ViK is the first mitigation approach against temporal memory safety violations that scales to modern OS kernels.

Validating memory accesses and employing unused bits in pointers are not new ideas. They have been used in previous work, such as CETS and Baggy Bound [8, 25]. However, the access validation approach are commonly believed to introduce prohibitively high performance overhead, and thus, it is impossible to make a practical defense. We revisit the access validation approach in this paper. Through careful and heavy optimizations, we demonstrate the feasibility of applying access-validation-based UAF defense on real-world, complex software, such as OS kernels. The optimizations used in ViK will benefit other runtime defenses.

The pure-software version of ViK does not rely on hardware features, which makes it applicable to legacy hardware. Nonetheless, emerging hardware features can bring substantial benefit: By employing the Top Byte Ignore (TBI) feature of AArch64 processors, we implemented a hardware-assistant ViK variant, ViK$_{TBI}$, that achieves an average full-system runtime overhead of less than 2% when applied on Android kernels. As a result, we are currently deploying ViK$_{TBI}$ in OS kernels in smart-automotive consumer devices *in the real world*. To our knowledge, this makes ViK the first kernel-level temporal memory safety mitigation to be deployed in a real-world product.

Contributions. The paper makes the following contributions:

- We introduce ViK, a novel, lightweight, and widely applicable mitigation against temporal memory safety violations.
- We implement a prototype of ViK as LLVM passes and apply it as both the first OS kernel mitigation against temporal memory safety violations (Linux kernel on x86-64 and Android kernel on AArch64) and a mitigation for user-space applications.
- We thoroughly evaluate the effectiveness and overhead of ViK on two OS kernels and user-space applications.

2 BACKGROUND

Temporal memory safety violations occur if a dereferenced pointer no longer points to the original object. Such violations help attackers compromise computers with vulnerable operating systems and user-space programs, as they can be used for privilege escalation or arbitrary code execution. For ease of illustration, we will consider use-after-free (UAF) vulnerabilities as an example to show the implications of temporal memory violations and motivate the design of our approach.

2.1 Use-After-Free Exploits and Defenses

A UAF vulnerability exists when a pointer value can still be dereferenced after deallocation. To exploit a UAF vulnerability, an attacker must deallocate a *victim object* and create a *dangling pointer*. A *victim object* is a memory object that has been deallocated through a deallocation function (*e.g.*, `free()`), and a *dangling pointer* has a pointer value that points inside a victim object. The attacker will then re-allocate the dereferenced memory region to another object and use the dangling pointer to read from or write to this newly allocated-object, without the constraints that would normally be applied when using the original pointer.

Specifically, exploiting a UAF vulnerability requires the following three steps: (1) Creating a dangling pointer, (2) allocating an object to overlap with the deallocated victim object, to which the dangling pointer points, and (3) dereferencing the dangling pointer. Hence, to defend against UAF attacks (and any attack that exploits temporal memory safety violations), it suffices to stop the attack at any of these three steps. Previous work has proposed approaches that cover each of the steps. We classify previous work by the three following types based on the above three steps.

Pointer invalidation. Defenses designed to prevent the creation of a dangling pointer either invalidate pointers that point to deallocated memory regions or prevent the deallocation of an object if there are pointers that point to it [11, 18, 20, 30, 34, 38]. These defenses all maintain additional metadata to track the relationships between pointers and their corresponding objects (memory allocations). Compared to approaches based on safe memory allocation (which will be discussed later), pointer invalidation techniques impose lower memory overhead. However, pointer invalidation methods usually incur substantial runtime overhead because they have to monitor memory allocations and maintain relevant metadata. In multi-threaded programs, the use of joint metadata could also impose a high-performance overhead because the metadata must be updated in a thread-safe way [30]. Additionally, due to the difficulty (and infeasibility) of performing a sound and complete data flow analysis on programs in general, pointer invalidation defenses inevitably suffer from false negatives. For instance, these defenses do not track the propagation of pointer values through type-unsafe pointers. In addition, they cannot track and invalidate pointers which are stored in registers and on the stack, which can result in further false negatives [11, 18, 34, 38].

Safe memory allocation. For OS kernels, memory allocation techniques such as the SLUB allocator aim to increase the difficulty of exploiting UAF vulnerabilities. The allocator guarantees that a kernel object only overlaps with a deallocated object with the same

size. However, SLUB does not completely mitigate the exploitation of UAF vulnerabilities [37].

There are several similar mitigations designed to prevent the re-allocation of objects to the memory areas previously occupied by a victim object [7, 10, 12, 26, 31]. Although these mitigations make UAF exploits unfeasible by allocating each object in a unique virtual memory area, they tend to incur high memory overhead due to the use of new allocation policies. While, recently proposed user-space memory allocators, MarkUs [6] and FFmalloc [35], showed good performance overhead but still impose high memory overhead.

Access validation. Several approaches attempt to prevent UAF attacks by validating every memory access that involves a pointer dereference [23, 25, 32]. While these approaches provide security guarantees and acceptable memory overhead, they all incur substantial runtime overhead because they must check every pointer dereference or update metadata on a regular basis during runtime [25]. Recently, another access validation approach, PTAuth [23], was proposed. Despite the improvement, it still imposes high runtime overhead (26% for selected SPEC 2006 benchmarks on average). Also, PTAuth currently does not support multi-threaded programs [23].

2.2 Unused Bits in 64-Bit Virtual Addresses

64-bit architectures use 64-bit virtual addresses. However, most modern 64-bit architectures do not fully utilize the 64-bit virtual address space. For example, x86-64, AArch64, RISC-V, MIPS, and OpenSPARC only support virtual addresses up to (or less than) 48 bits, which correspond to a virtual address space of at most 256 TB[1] [9, 14, 15, 22, 33]. We observe that the most significant 16 bits in every pointer value are currently unused for *data pointers* on most processors.

These unused bits have not gone unnoticed. ARM announced pointer authentication instructions in the ARMv8.3-A instruction set, which uses the unused bits in 64-bit pointers to sign and authenticate virtual addresses [29]. To maintain the integrity of pointers in specific contexts, developers can generate a pointer authentication code (PAC) and store it in place of the unused bits. This approach is used for detecting (and verifying) changes to a pointer value (i.e., address) rather than validating the relationship between a pointer and a virtual address to which the pointer points. One typical example of pointer authentication is using it to protect the stack pointer. Unfortunately, pointer authentication instructions are *not* able to prevent UAF vulnerabilities directly because UAF can occur regardless of the pointer's authenticity [19]. Therefore, Farkhani et al. proposed PTAuth that prevents UAF exploits by using the PAC [23]. However, even with the aid of the PAC, PTAuth could not avoid high runtime overhead.

We note that ViK does not exclude the use of PAC but complements it by providing additional protection on relationships between data pointers and memory objects. It is even possible to use ViK and PAC together on the same pointer similar to PTAuth [23].

In addition, ARM introduced Memory Tagging Extension (MTE) in ARM v8.5 [2] and Application Data Integrity (ADI) is enabled in a number of SPARC processors (M7, M8, S7, T7, and T8) [27]. With MTE, a tag is assigned to each allocated memory region, and only a pointer that has the same tag can access the region. Similarly, ADI utilizes version numbers stored in the unused bits of application's memory pointers and the memory they point to. We expect that MTE and ADI can help prevent memory errors. However, the size of a tag in MTE and a version number in ADI is just 4 bits [5, 17], and thus, can only have 16 possible values. Also, how MTE and ADI can be automatically applied to OS kernels is an open research question. Currently, ADI can be used for user-space programs on Linux and Oracle Solaris [17, 27], but Linux kernel support for using MTE in user-space programs remains in development [13]. It, thus, is important to mitigate temporal memory safety violations for OS kernels and for the large set of processors, including the latest Intel chipsets, that do not offer hardware features for detecting temporal memory errors.

[1]On x86-64, the most significant 16 bits of a virtual address (from the 48th bit to the 63rd bit) must be the same as the 47th bit. Otherwise, the processor will raise an exception when accessing the address.

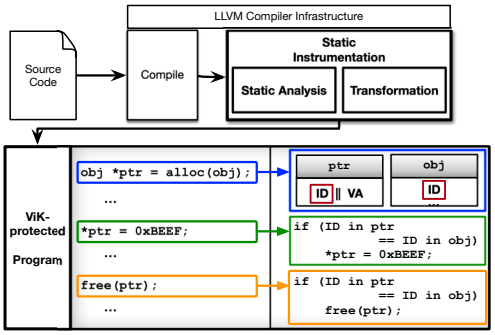

Figure 1: Overview of the static instrumentation for applying ViK and a ViK-protected program.

3 OVERVIEW

ViK identifies all allocation and pointer dereferencing sites and performs *static instrumentation* on the LLVM bitcode to insert object-ID-specific logic. Specifically, ViK replaces each memory allocator with a new one ($alloc_{vik}(x)$) and adds an object-ID check ($inspect(p)$) at (1) pointer dereferencing sites and (2) when an object is deallocated as defined in Section 5.

The core of ViK consists of the following three main steps:

I. ViK assigns a random object ID to an object when one is allocated in the heap memory.
II. ViK copies the assigned object ID to the unused bits of a pointer value and to a reserved field at the base of the newly allocated object. This way, ViK creates a correspondence between the pointer value and the object.
III. At runtime, when the pointer value is dereferenced, ViK inspects the object ID and allows the dereference only when the ID stored in the pointer value matches the one in the object. Also, ViK inspects the object ID when an object is freed to prevent the double-free.

Threat model. Similar to existing approaches [6, 12, 20, 30, 34], we especially focus on heap-related temporal memory safety vulnerabilities as stack-based UAF or double-free vulnerabilities can

be handled by use-after-scope or escape analysis [6, 12]. In addition, such stack-base UAF vulnerabilities are rare in reality. Kernel Address Sanitizer (KASan) stopped the support of detecting stack-related UAF errors because the detector is considered "almost entirely useless" [4].

Dynamic instrumentation in ViK. To maximize compatibility with legacy programs and systems, ViK is designed to be independent of hardware support on 64-bit architectures. By embedding object IDs directly into pointer values, ViK avoids two common dependencies, relied on by other defenses, that cause excessive runtime overhead, memory overhead, and concurrency challenges: the use of in-memory central metadata and the tracking of pointer propagation. Because it is stored in the pointer itself, object IDs used by ViK always move with the pointer value to which they belong whether the pointer value is loaded to a register, propagated into other pointers, or spilled onto the stack. However, a pointer value might not point to the base address of an object. To find the base address of any object pointed by a protected pointer value, ViK embeds a *base identifier* into each object ID, which will be discussed, together with the design of object IDs, in Section 4.

Static optimization in ViK. Our design allows it to scale to OS kernels. To accomplish this, ViK must be able to defend programs that have extreme numbers of pointer accesses (*e.g.*, there are about 2.3 million pointer dereferences in Linux kernel 4.12), and naïvely inspecting every memory access will incur an impractical (or, minimally, unnecessary) runtime overhead. Therefore, we aim to minimize the number of pointers to inspect by limiting the dynamic instrumentation only to those dereferencing sites that are potentially unsafe. To this end, ViK conducts an inter-procedural static data-flow analysis to identify memory accesses that are considered to be safe from UAF exploits and exclude them from the inspection.

One result of this optimization is that ViK omits object ID inspection on dereferences of pointers that are never stored in global regions or the heap, deeming them "UAF-safe." Because these temporal pointer values only exist on the stack, have a very short lifetime, and are generally accessed by a limited amount of code, their use as a dangling pointer in a UAF exploit is extremely unlikely.[2] Therefore, ViK does not inspect pointer values that are *only* stored on the stack and are never copied to the heap or into global variables. Prior work shares this trade-off: For example, DangNull only tracks pointers located on the heap [18]. Our protection model covers more dereference sites than DangNull [18] and the same amount of dereference sites as CRCount [30] and pSweeper [20]. Additionally, ViK covers pointer values in registers, weak-typed pointers, and pointers that are spilled onto the stack. We will extensively discuss this in Section 5.

Mis-detections. ViK's design guarantees an absence of false positives (mistaken UAF detection of UAF though it cannot take place), but false negatives (missed detections of UAF though it can take place) can occur if two objects are assigned the same object ID or if a pointer operation assumed to be UAF-safe is actually attackable. For the former case, *object ID collisions* occur with a very low probability. We believe that there is sufficient entropy in the object IDs to defeat an attacker's attempt of circumvention as we will discuss in Section 4.2 and Section 7.3, respectively. For the latter case, we show that ViK can catch the case, which we term *delayed mitigation* and will discuss in Section 7.3.

[2]Note that, if a heap-stored dangling pointer is used to attack the object pointed to these UAF-safe pointers, ViK will still catch the attack.

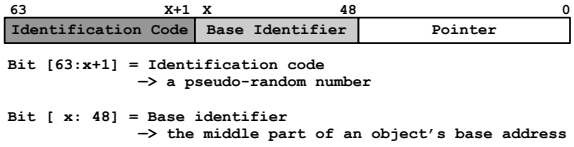

Figure 2: The object ID used in ViK.

4 OBJECT ID

We present how ViK generates object IDs (Section 4.1) and then further discuss object ID entropy (Section 4.2).

4.1 Generating Object IDs

As depicted in Figure 2, an object ID has 16-bits and is comprised of two parts of variable lengths.

Identification code. The identification code is a random number generated by the ViK allocator (defined in Section 5). ViK uses the identification code to identify each allocated object.

Base identifier. A pointer value does not necessarily point to the base address of an object; instead, it may point to any field inside the object. We need to be able to map a pointer value to the base address of the object to which it points so that ViK can find the object ID, which is stored as the first member of the object. Therefore, ViK introduces a *base identifier*, which is used to find the base address of any object.

ViK aligns allocated objects to a predefined alignment of 2^X bytes, which essentially creates *slots* of at least 2^X bytes. By aligning the addresses of objects, the least significant X bits of all objects' base addresses must be zero. An object may require one or more slots depending on its size. For example, if a slot is 16 bytes, a 24-byte object will use two slots.

ViK uses two predefined constants M and N to determine the alignment (which is also the size of each slot). 2^M is the maximum size (in bytes) of objects that can be covered by using slots of size 2^N (bytes) with a base identifier of $M-N$ bits. For example, suppose M and N are 12 and 6, respectively; the maximum size of any object is $2^{12} = 4096$ bytes; and the size of each slot is $2^6 = 64$ bytes. The base identifier will be $12 - 6 = 6$ bits long.

Once the constants are determined, the base identifier can be calculated from the start address of an allocated memory region as shown by Lines 1–3 of Listing 1. During runtime, ViK can recover the base address of an object from any pointer value, as in Lines 5–7 of Listing 1. ViK only uses bitwise operations to find base addresses of objects. It does not need memory accesses, which helps keep runtime overhead low.

Determining the constants. M and N must be configured before ViK's instrumentation. Since slots are the smallest allocation unit in ViK, using large slots can cause excessive memory overhead. ViK asks the user to specify M and N with the assistance of the knowledge of object sizes. ViK helps users to determine optimal choices of the two parameters by identifying sizes of all the involved objects in the target program, which is straightforward to do with

```
get_base_identifier(pointer, M, N):
  BI = (pointer & (2^M − 1)) >> (N);
  return BI;
get_base_address(pointer, M, N, BI):
  BA = (pointer & ~(2^M − 1)) | (BI << N);
  return BA;
```

Listing 1: Pseudo code for extracting the base identifier and recovering the base address of an object from a pointer value. Both operations only use bitwise instructions.

a compiler pass. Note that determining the optimal M and N is a one-time effort for each target program. We will demonstrate the process of determining M and N on Linux kernel in Section 6.3.

4.2 Object ID Collisions

The effective entropy of an object ID is equal to the length of identification code. The base identifier does not add any security: As long as the attacker allocates an object at the exact same address where the victim object was placed, this newly allocated object will have the same base identifier as in the victim object. This entropy that ViK provides may seem low, but it is sufficient to stop attacks. In our evaluation, we used 10-bit identification code (which has a collision rate of about 0.09%), and ViK successfully defeated all attacks that use known UAF vulnerabilities in Linux kernel without any collision of object IDs. 10-bit entropy is equal or higher than KASLR implementations in OS kernels [16]. Further, for a successful UAF attack, the attacker must re-allocate an object at the location of the victim object and ensure that it has the same random object ID. In kernel UAF attacks, the attacker has only one chance: The kernel will panic upon failed attacks due to an invalid memory access via a pointer value returned from the `inspect()` function (Definition 5.2). Although the 0.09% collision rate may not seem very low, bypassing ViK will still be unlikely in practice.

5 INSTRUMENTATION

In this section, we discuss ViK's core logic—the instrumentation process. While ViK has a straightforward core idea, its true power lies in its scalability. We identify pointers that are immune from UAF vulnerabilities (coined *UAF-safe pointers* in Section 5.1) through performing a static data-flow analysis on the target program (Section 5.2), and exclude these pointers from ViK's protection. This optimization allows ViK to only protect a small subset of pointers, which significantly reduces runtime overhead and enables ViK to protect real-world, complex software such as OS kernels. Then ViK instruments the LLVM bytecode of the target program to insert pointer-inspection logic (Section 5.3).

Common Terms. We define terms that we will use throughout this paper: the memory allocator and the inspect function.

Definition 5.1. ViK's memory allocator ($alloc_{vik}()$) allocates in heap memory a chunk of a specific size (x) and returns its start address (p) along with an object ID (id). The object ID is also stored in the allocated chunk.

$$alloc_{vik}(x) \rightarrow p_{id},\ id \in [p...(p+x)] \in \{mem_{heap}\}$$

p_{id} is the combined representation of both p and id, where id is stored in unused bits of 64-bit pointers.

Definition 5.2. The inspect function ($inspect(p)$) returns p if and only if the id in p matches the id in a memory object where p points to.

$$inspect(p_{id}) \rightarrow p \Leftrightarrow id \equiv p \in \{mem_{heap}\} \wedge id \in [p...(p+x)]$$

Therefore, in ViK-protected programs, UAF attacks are mitigated because one of the following two cases will hold for dangling pointers: (1) it will have an object ID that is different from the ID of the object to which it points; or (2) it will not point to a valid memory region on the heap.

5.1 UAF-Safe Pointers

The runtime overhead of ViK is proportional to the number of pointer inspection sites that are inserted for validating memory accesses. Therefore, the goal of our optimization is to minimize the number of calls to the inspect function in the protected program. Our following insight allows us to exclude calls to the inspect function for a large portion of pointers without negatively impacting the quality of defense.

Insight. Our insight is that pointer values that are only stored on the stack and never copied to heap or global regions are immune from being used in UAF attacks. We regard these pointer values, and all memory accesses using these pointer values, *UAF-safe*, and exclude them from ViK's protection. ViK performs a static data-flow analysis to find UAF-safe pointers, which we detail in Section 5.2. The following text presents our reasoning.

Pointer values that are stored *only* on the stack are usually not exploitable, since these pointer values often have a short life time, i.e., the window between when these pointers are freed (thus, they become dangling pointers) and when these pointers are dereferenced again (if that ever happens) is extremely small. This is because the life time of stack variables are bounded to functions, and when a function returns, its stack-located pointer values will no longer be used. Such a small window makes it extremely difficult for attackers to exploit these stack-located pointers by allocating objects to overlap with the freed victim object. In comparison, pointer values that are stored in heap or global regions tend to have much longer life times, which make it easier for attackers to purposefully control memory allocation and build UAF exploits. State-of-the-art UAF defenses, such as DangNULL [18], makes the exact same assumption. Therefore, from a security aspect, our insight is reasonable.

Exception. The only possible UAF exploitation case that is currently known, with a strictly stack-located pointer value, is leveraging a double-free bug that can cause a UAF error. This is illustrated in Figure 3. ViK always inspects the pointer and checks its object ID when an object is deallocated. Hence, ViK successfully detects this exploitation attempt at the time when double-free happens.

UAF-safety. Memory accesses using UAF-safe pointer values are considered UAF-safe *unless* these pointer values are copied to global variables or the heap. Any pointer value that is copied from the heap or global variables is considered *UAF-unsafe*. We formally define how to determine UAF-safety as follows.

Definition 5.3. Any pointer value that points to a global or a stack variable is UAF-safe. Also, a pointer value that points to the heap is UAF-safe if and only if the pointer value has never been stored in the heap or a global variable.

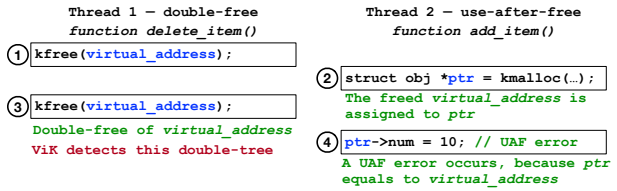

Figure 3: In Thread 2, a UAF occurs with a pointer value stored only on the stack by the double-free bug at ③ in Thread 1. This type of UAF error can be exploited if an attacker can allocate a new object between ③ and ④. ViK inspects the object ID when an object is deallocated at ③ and stops the attack.

This states that only globally known pointer values pointing to the heap are relevant to *exploiting* UAF vulnerabilities.

Definition 5.4. Assuming that a function takes a pointer value as one of its arguments, this pointer value is UAF-safe if and only if the pointer value is UAF-safe in the caller function.

When only UAF-safe pointer values are used as arguments of function calls, these arguments are UAF-safe in the callee function since they are still used as stack variables and may not be accessed by other threads unless they are copied to a global variable or the heap.

Definition 5.5. Assuming that a function returns a pointer value, the left-hand-side (lhs) pointer value at the call site in the caller function is UAF-safe if and only if the pointer value is UAF-safe in the callee function

If the returned pointer value is UAF-safe at the return site of the callee function, the *lhs* pointer value at the call site will be a local variable in the caller function before it is copied to a global variable or the heap. Hence, this pointer value is still UAF-safe before the copying happens. However, if we do not know whether or not the returned pointer value is UAF-safe in the callee function, for soundness, we must make an under-approximation by assuming that the pointer value in the caller function is *UAF-unsafe* from when the pointer value is returned. This can happen when our analysis does not consider the callee function or cannot determine the UAF-safety of a pointer value returned by the callee function. Likewise, any pointer values that are not UAF-safe are considered to be UAF-unsafe for soundness.

5.2 Determining UAF-Safety

ViK employs static data-flow analysis to determine if a pointer value is UAF-safe at a site where it is dereferenced. For accuracy, ViK's static analysis is both flow-sensitive and path-sensitive. In intra-procedural analysis, since the UAF-safety of a pointer value may change depending on its execution path, simply backtracking data flows is insufficient to determine the UAF-safety of a pointer value at arbitrary pointer operations. To consider all possible cases that affect the UAF-safety of a pointer value, our static analyzer contains a Reaching Definition Analyzer (RDA) that works on LLVM bitcode instructions. The RDA recovers all possible data flows that can reach a pointer operation. We regard a pointer value as UAF-safe if and only if all its uses are UAF-safe according to Definitions 5.3, 5.4, and 5.5.

Step 1: Intra-procedural Analysis. We conduct an intra-procedural analysis on each function, to find UAF-safe pointer values, by analyzing all pointer operations in a function. To this end, we execute RDA for every pointer operation. According to Definition 5.3, a pointer value is UAF-safe if it points to a stack variable or a global variable. In such cases, ViK does not place *inspect*() functions for that pointer value. Pointer values copied from the heap or global variables must be inspected by ViK because they are considered UAF-unsafe.

Next, we mark a pointer value as UAF-unsafe if the pointer variable may hold a return value from another function. As the first step is intra-procedural, we do not know if the returned pointer value is UAF-safe or not. However, we mark pointer values with return values returned from basic allocators (*e.g.*, *malloc*() in libc or *kmalloc*() in the Linux kernel) as UAF-safe, since the pointer values returned by basic allocators are obviously UAF-safe. Lastly, if a function argument is a pointer value, we deem this pointer value UAF-unsafe. The UAF-safety of these pointers can be updated after finding UAF-safe arguments and return values in future steps.

Step 2: Analyzing UAF-safe heap addresses. Once the initial intra-procedural analysis is complete, we build a call graph for each module (a source file or an object file after compiling) of the target program. The call graph is used to determine the UAF-safety of pointer values in our inter-procedural analysis.

First, we analyze pointer values that hold the pointer values returned from basic allocators. Note that immediately after a basic allocator returns, the pointer value (which is the address of the newly allocated object) is unknown to other threads. If there is an instruction that copies a pointer value to a global variable or the heap, the pointer value must become UAF-unsafe *after* the execution of the `store` instruction. Tracking data flows related to these pointer values is straightforward since they are local variables located within the stack frame.

Step 3: Finding UAF-safe function arguments. To find UAF-safe arguments for every function call, we visit functions from the dominator node based on the call graph. We check every call instruction and its arguments. If a pointer is used as an argument, we decide whether or not the argument is UAF-safe. We add an attribute to the argument to mark it, only if an argument is UAF-safe for every case before the call instruction. We repeat this process until we have visited all functions in the call graph. Because we limit the range of our static analysis to a single module, we omit to check arguments of functions that escape the analysis scope.

If an argument of a function was marked as UAF-safe, we visit the function to run RDA again to change the UAF-safety of pointers in each pointer operation that we visited before.

Step 4: Finding UAF-safe function return values. We find UAF-safe return values from the post-dominator nodes in the call graph. Similar to the process of finding safe arguments, we visit functions that return a pointer value and analyze their return values. In a function, it is possible to have multiple return instructions depending on execution paths. Therefore, we perform a path- and flow-sensitive analysis for each return instruction and decide their UAF-safety. Only if all possible return values are UAF-safe, do we mark the pointer value as a UAF-safe. As in Step 3, it is necessary

to execute the RDA again for every function after finding UAF-safe return values. The UAF-safety of pointer values can be changed depending on the UAF-safe return values.

Step 5: Finding the first memory access in a function. Through the previous analyses, the UAF-safety of all pointer values has been decided. In the last step, we optimize the number of *inspect*() functions for each function. The idea of this step is to inspect only the very first pointer operation of a UAF-unsafe pointer value in a function, and thus, UAF-unsafe pointer values are inspected once in every function. We carefully take this step using RDA to detect changes of pointer values along all possible execution paths. We thus are able to find the first memory access among pointer operations using the same pointer value. This optimization significantly decreases the runtime overhead, but it may lead to false negatives. We discuss its security implications further in Section 7.3.

5.3 Transformation

In the transformation phase, we first insert the *inspect*() function before pointer operations that ViK must inspect during runtime. In the *inspect*() function, ViK does not store the restored virtual address back to memory, but stores it only in a register temporarily and accesses the heap by referencing the register. We thus need to insert a *restore*() function before all the other pointer operations using ViK-protected pointer values. The *restore*() function is used to temporarily recover the canonical form of a virtual address through only a bitwise operation. Consequently, ViK-protected pointer values must go through either the *inspect*() or *restore*() function before a pointer operation. When deallocating an object, only the *inspect*() function is used. ViK inlines these functions at each inspection site. Compared to inserting a *call* instruction that invokes the inspection function, inlining increases the size of prgrams but it is critical to lowering the runtime overhead.

Next, ViK replaces function calls to basic allocators and deallocators with call instructions that will invoke the wrapper functions. When a new object is allocated, the wrapper functions will generate an object ID, store the object ID at the base address of the object, and return it to the caller as part of the pointer value. In wrappers for deallcaotors, ViK inspects the object ID before an object is deallocated.

Pointer arithmetic. Since ViK only uses the unused bits of a pointer value, all legal pointer arithmetic operations ("+", "++", "-", and "−") can be used on ViK-protected pointers without restoring them first. In rare cases, pointer values may be used in a comparison (e.g., `ptr1 == ptr2`), where the object IDs in the two pointer values are different if they are not derived from the same base pointer. In such cases, ViK will restore the pointer values before comparing pointers.

6 IMPLEMENTATION

We implement ViK in both OS kernels (Linux kernel 4.12 on x86-64 and Android kernel 4.14 on AArch64) and user-space applications. Also, we implement ViK$_{TBI}$ using the Top Byte Ignore (TBI) feature of recent ARM processors.

```
1 void inspect(pointer_value) {
2   PTR_ID = pointer_value >> 48;
3   BI = PTR_ID & 0x003f;
4   BA = (pointer_value & ~(2^M − 1))|(BI << N)|0xffff000000000000;
5   OBJ_ID = *BA;
6   pointer_value = pointer_value & (~(PTR_ID ^ OBJ_ID) << 48); /*If
        two object IDs do not match, dereferencing this new
      pointer_value will raise an exception.*/}
```

Listing 2: The pseudocode of the *inspect*() function which only consists of bitwise operations and a load operation for minimal execution overhead. Protected pointer values are restored to their canonical forms if object IDs are matched.

6.1 Kernel Implementation

ViK is mostly architecture-agnostic for user-space and kernel protection, as long as unused pointer bits are available on the target architecture. The only step to adapt ViK to OS kernels is to patch a small amount of inline assembly code, such as inline functions implemented in `atomic.h`, because LLVM IR passes cannot analyze, modify, or rewrite assembly code. We added or modified 446 and 363 lines of code for Linux and Android kernels, respectively. The Linux kernel has many basic memory allocators, and our implementation handles all allocators of the `kmalloc` and `kmem_cache_alloc` family. Note that we excluded source code related to the booting process from instrumentation because these functions will no longer execute after booting is complete.

Inspection logic. Since the *inspect*() function checks every pointer dereference, its implementation is critical to minimizing ViK's runtime overhead. Therefore, we implemented the *inspect*() function in a conditional-instruction-free manner. We only use bitwise instructions to inspect pointer values, however, the *inspect*() function must still raise an exception when a pointer value and an object have different object IDs. The key idea is outsourcing the job of raising exceptions upon unmatched object IDs to the CPU.

Listing 2 shows the pseudocode of *inspect*() function that consists of bitwise instructions and one memory access for loading the object ID from the heap. First, the function extracts the object ID from a pointer value by bit shifting (Line 3). Second, on Lines 4 and 5, it obtains the base identifier and the base address of the object. Third, it loads the object ID from the actual object stored in memory. Then, the *inspect*() function performs a bitwise `XOR` operation using object IDs stored in the pointer value and the object. The result of this operation is used for a bitwise `AND` operation with the pointer value. If the two object IDs are identical, the pointer value will be of the canonical form (for the kernel, all the unused bits will be 1), and thus, the pointer value will be properly dereferenced and the corresponding object will be accessed. Otherwise, the processor will raise an exception.

Enforcing memory alignment. In ViK-protected programs, all memory objects must be located at aligned memory addresses that are derived from the constant N, which is used for generating the base identifier. However, the basic allocators of the Linux kernel do not guarantee this memory alignment requirement. Therefore, we enforce memory alignments by wrapping basic allocators in custom wrapper functions in which additional bytes are added to enforce alignment.

Table 1: The sizes of structures dynamically allocated in Linux kernel 4.12. Roughly 98% of structures is smaller than 4 KB.

Allocation size (byte)			M	N	$M-N$	Alignment	Percentage
	x	< 256	8	4	4	16	76.73%
$256 \leq$	x	< 4096	12	6	6	64	21.31%

The wrappers execute the following operations: (1) When an object is allocated, they allocate (2^N + 8) bytes more than the size of the object, where 2^N bytes is the size of a basic alignment unit. The additional 8 bytes are used for storing an object ID. (2) The wrappers then determine a base address that is aligned to 2^N within the allocated memory region. Because the wrapper allocated an additional 2^N bytes, there must be an address that is aligned by 2^N bytes. (3) The wrappers store the object ID at the base address. (4) The wrappers return a pointer value with a value of the base address plus 8, after storing the object ID into the pointer value's unused bits. If the aligned memory address is $X + 2^N$ (where X is a virtual address returned from the basic allocator), the wrappers store the object ID at the address $X + 2^N$ and the object will use memory from address $X + 2^N + 8$. Because the wrappers allocate an additional $2^N + 8$ bytes, the object can be stored from the virtual address ($X + 2^N + 8$).

6.2 ViK$_{TBI}$ for AArch64 on Android Kernel

ViK$_{TBI}$ using the TBI feature of recent ARM processors achieves much lower performance overhead. With TBI, software can use the most significant 8 bits of the virtual address to hold additional information about an address. By employing this hardware feature, ViK can utilize the 8 bits without handling it in software. However, because only 8 bits are available, we do not use the base identifier in ViK$_{TBI}$ in order to have 8-bit entropy for object IDs. This implies that only pointer values that point to the base addresses of objects can be inspected. Also, when an object is created, we insert padding bytes and store an object ID right before the base address of an object so that the ID can be accessed via the base address. Albeit ViK$_{TBI}$ cannot provide as strong of security as ViK can, its high efficiency makes it very practical. We evaluate the security effectiveness and performance overhead in Section 7.

6.3 Determining the Constants

As discussed in Section 4.1, our proof-of-concept implementation of the instrumentation pass has a functionality to provide the sizes of dynamically allocated memory objects so that we can analyze them and decide the two parameters. Table 1 shows our results on the size of objects dynamically allocated in Linux kernel 4.12. This object memory size analysis helps with ViK's effectiveness and is done one-time for the kernel. Based on this analysis, we found that over 98% of the kernel memory objects are smaller than 4 KB. In the Android kernel, about 98% of memory objects are smaller than 4 KB as well. Therefore, for the security evaluation, we used 6-bit base identifiers with the parameters M=12 and N=6 to have 10-bit identification codes for all objects, and we did not assign an object ID for the objects which are larger than 4 KB. We set the constants according to the M and N shown in Table 1 for evaluating the memory overhead of ViK. It is worth noting that different sets of the constants (M and N) can be used for optimal memory overhead on each system. The prototype of ViK only supports these sets of parameters, and we leave this implementation improvement as future work (see Section 8).

7 EVALUATION

We evaluate the effectiveness and performance of ViK in protecting OS kernels and user-space programs.

7.1 Experiment Setup

For OS kernels, we built Linux kernel 4.12 on x86-64 and Android kernel 4.14 on AArch64. All the experiments were conducted on a workstation with an Intel i7-6700 CPU with Ubuntu 18.04 x86-64 and on a development board featuring an ARM Cortex-A76 for the Android kernel.

Optimization modes. We evaluate ViK-protected OS kernels in the following optimization modes.

ViK$_S$: An *inspect*() function is inserted for every dereference of possibly UAF-unsafe pointers.

ViK$_O$: All optimization methods presented in Section 5.2 are enabled. In this mode, ViK only inspects the first object access of each UAF-unsafe pointer in every function.

ViK$_{TBI}$: ViK is implemented using the Top Byte Ignore (TBI) feature of AArch64, where ViK inspects only pointer values that point to the base address of objects.

7.2 Kernel Instrumentation Results

We measured the number of inserted inspection functions (*inspect*()) and the increase of image sizes after deploying each of the three variants of ViK on Ubuntu and Android kernels in different architectures. Our results, in Table 2, show that in both kernels, ViK$_S$ inserted inspection functions for around 17% of pointer operations, which means the static analysis regarded 17% of all pointer operations as potentially operating on UAF-unsafe pointers. The other 83% of pointer operations were UAF-safe and do not need any protection. Moreover, in ViK$_O$ mode, our results show that only 4% of pointer operations must be inspected by ViK: ViK$_O$ decreases runtime overhead by omitting protection for over three quarters of all UAF-unsafe pointer operations. ViK$_{TBI}$ further reduces runtime overhead by only instrumenting less than 8% of pointer operations that are protected by ViK$_O$.

7.3 Security Effectiveness

ViK mitigates UAF exploits with *no* false positives (i.e., incorrectly blocking pointer accesses that are not UAF). Nonetheless, ViK may have false negatives and allows a UAF exploit: If an object that has been re-allocated to the freed region has the same object ID as the victim object because of an object ID collision, ViK cannot mitigate the UAF exploit. Fortunately, as discussed in Section 4.2, the probability of an object ID collision is small enough to make ViK practical.

Another case of false negatives occurs only in ViK$_O$ where ViK does not mitigate the exploit immediately but instead exhibits a *delayed* mitigation. Figure 4 shows a case where a UAF exploit occurs due to a race condition. If `dealloc()` is executed at any

Table 2: Statistics of ViK-protected Linux kernel 4.12 (x86-64) and Android kernel 4.14 (AArch64). About 17% of all pointer operations involve UAF-unsafe pointers. ViK$_O$ and ViK$_{TBI}$ instrument much fewer pointer operations than ViK$_S$ does.

Kernel & Architecture	Mode	Image size (MB) Original	Image size (MB) ViK (delta %)	Build time Original	Build time ViK (delta)	# of pointer operations	# of *inspect*() functions (%)
Linux kernel 4.12 x86-64	ViK$_S$	36.26	63.38 (+27.12%)	19m 11s	26m 31s (+7m 20s)	2,401,337	421,406 (17.54%)
	ViK$_O$		48.33 (+12.07%)		22m 10s (+2m 59s)		91,134 (3.79%)
Android kernel 4.14 AArch64	ViK$_S$	189.09	208.20 (+19.11%)	23m 3s	27m 32s (+4m 29s)	2,012,421	333,020 (16.54%)
	ViK$_O$		200.94 (+11.85%)		25m 38s (+2m 35s)		78,782 (3.91%)
	ViK$_{TBI}$		200.93 (+11.84%)		25m 30s (+2m 27s)		25,969 (1.29%)

```
1 void race() {
2     global_ptr->num = 1;
3     ...
4     global_ptr->num = 0;
5     ...
```

```
6 void dealloc() {
7     free(global_ptr);
8 }
9
10
```

Figure 4: Example code snippets that can cause a UAF error via a race condition.

time before the last dereference of `global_ptr`, this variable will become a dangling pointer and will cause a UAF. ViK$_S$ inspects every UAF-unsafe pointer operation (at both Line 2 and Line 4) and mitigates the UAF exploit. However, ViK$_O$ only inserts inspection functions at Line 2 of `race()`. It is then possible for a UAF to occur if the object that `global_ptr` points to is deallocated between executing Line 2 and Line 4. In the worst-case scenario, the attacker frees the victim object between Line 2 and Line 4 and re-allocates a new object to the dereferenced memory region, which will evade our protection. ViK$_O$ will still mitigate the exploit if the pointer is dereferenced again in other functions later, as we have observed in CVE-2019-2215.

Mitigating real-world kernel exploits. To evaluate the effectiveness of ViK, we selected five known UAF vulnerabilities with public exploits on Linux kernel and tested them against a ViK-protected Linux kernel 4.12. All these vulnerabilities are related to race conditions. For Android kernel, we picked four UAF vulnerabilities, three out of which are caused by race conditions. All the exploits are collected from the Exploit Database and another research project FUZE [3, 36]. Five of these vulnerabilities (CVE-2017-17053, -15649, CVE-2019-2215, -2025, and -2000) can be exploited directly on Linux kernel 4.12 and Android kernel 4.14 (without ViK), while the other four vulnerabilities (CVE-2017-11176, -7533, -2636, CVE-2016-8655, and -4817) do not exist on our versions of Linux and Android kernels. We manually ported them onto Linux kernel 4.12 and Android Kernel 4.14 by reverting the related patches. Details of the selected vulnerabilities and the results of the security evaluation are shown in Table 3. As expected, ViK-protected kernels, including ViK$_O$, detected UAFs caused by these vulnerabilities.

TBI optimization. We evaluated the TBI-featured variant of ViK, ViK$_{TBI}$, on Android kernel 4.14 and list the results in Table 3. ViK$_{TBI}$ did not stop the exploit for CVE-2019-2215 because the exploit uses a pointer that points to the middle of an object while ViK$_{TBI}$ only inspects pointers that point to the base address of an object. Also, it is worth noting that a delayed mitigation happened with CVE-2019-2000: Since the dangling pointer used in the exploit points to the middle of an object, ViK$_{TBI}$ did not detect the UAF exploit when this pointer is first dereferenced and the victim object is updated.

Table 3: Experimental results of ViK against known UAF exploits in OS kernels.

CVE	Race Condition	Linux kernel 4.12 ViK$_S$	ViK$_O$	(ViK$_{TBI}$)
CVE-2017-17053	Yes	✓	✓	(✓)
CVE-2017-15649	Yes	✓	✓	(✓)
CVE-2017-11176	Yes	✓	✓	(✓*)
CVE-2017-2636	Yes	✓	✓	(✓)
CVE-2016-8655	Yes	✓	✓	(✓)
CVE-2016-4557	Yes	✓	✓	(✓)

CVE	Race Condition	Android kernel 4.14 ViK$_S$	ViK$_O$	ViK$_{TBI}$
CVE-2019-2215	No	✓	✓	✗
CVE-2019-2025	Yes	✓	✓	✓
CVE-2019-2000	Yes	✓	✓	✓*
CVE-2017-7533	Yes	✓	✓	✓

*: ViK$_{TBI}$ did not immediately stop the exploit when UAF happened, but it stopped the attack through a delayed mitigation.

However, ViK$_{TBI}$ detected the UAF when the original pointer (which points to the base address of the object) is later used before returning from the kernel to user space, which illustrates the effectiveness of ViK even when applying all aforementioned optimizations. Finally, since current x86-64 CPUs do not implement TBI, we examined every Linux kernel vulnerability in our dataset, manually analyzed if ViK$_{TBI}$ will defend against each UAF exploit, and present the results in the ViK$_{TBI}$ column of Linux kernel in Table 3.

Sensitivity analysis of object IDs. We performed a sensitivity analysis of object IDs by using Linux kernel UAF exploits in Table 3. For the analysis, we executed each exploit 2,000 times on a ViK-protected Linux kernel. Eventually, ViK detected all UAF errors. As the result demonstrates, creating a successful exploit is very difficult under ViK because an attacker must re-allocate an object that has the same base identifier as the victim object and the newly allocated object must have the same object ID which is randomly selected (the random space is not decreased by allocating new objects).

7.4 Performance: OS Kernels

For performance evaluation, we used two renowned benchmark tools: LMbench and UnixBench. We then evaluated the memory overhead of each kernel when protected by ViK.

Micro benchmarks on kernels. We first present the benchmark results using LMbench which measures latency and basic costs of key operations of UNIX/POSIX systems [21]. Table 4 shows the results for each kernel. In ViK$_O$, the average percentages of increased latency are 20.71% and 19.86% on the Ubuntu and Android kernel, respectively. Because ViK inserts fewer *inspect*() functions into the Android kernel, its performance overhead is lower than ViK on the Linux kernel 4.12. As expected, ViK$_O$ has substantially

Table 4: The runtime overhead measured by LMbench.

Benchmark	Linux kernel 4.12 ViK$_S$	Linux kernel 4.12 ViK$_O$	Android kernel 4.14 ViK$_S$	Android kernel 4.14 ViK$_O$
Latency (percentage of increase)				
Simple syscall	16.88%	10.82%	15.60%	7.16%
Simple fstat	96.74%	67.41%	68.86%	47.15%
Simple open/close	140.40%	77.01%	74.88%	38.62%
Select on fd's	23.19%	15.42%	35.52%	28.47%
Sig. handler installation	6.36%	4.09%	19.24%	6.37%
Sig. handler overhead	41.19%	4.34%	113.83%	46.86%
Protection fault	0%	0%	5.52%	0%
Pipe	40.91%	26.48%	60.80%	15.45%
AF_UNIX sock stream	26.91%	8.35%	77.91%	23.80%
Process fork+exit	85.90%	68.01%	35.13%	16.40%
Process fork+/bin/sh -c	96.45%	62.66%	32.21%	14.31%
GeoMean	**40.77%**	**20.71%**	**37.13%**	**19.86%**

Table 5: The performance overhead measured by UnixBench.

Benchmark	Linux kernel 4.12 ViK$_S$	Linux kernel 4.12 ViK$_O$	Android kernel 4.14 ViK$_S$	Android kernel 4.14 ViK$_O$
Dhrystone 2	0%	0%	0%	0%
DP Whetstone	0.83%	0.21%	0%	0%
Execl Throughput	77.95%	48.18%	50.32%	28.62%
File Copy 1024 bufsize	100.30%	56.43%	123.00%	61.13%
File Copy 256 bufsize	99.33%	54.45%	148.91%	77.51%
File Copy 4096 bufsize	70.71%	41.89%	71.42%	34.01%
Pipe Throughput	110.90%	74.66%	60.77%	41.55%
Pipe-based Ctxt. Switching	126.70%	80.78%	50.09%	0.39%
Process Creation	85.05%	57.22%	42.53%	22.58%
Shell Scripts (1 concurrent)	58.47%	36.16%	34.88%	22.13%
Shell Scripts (8 concurrent)	55.96%	35.71%	27.24%	16.02%
System call overhead	8.89%	1.11%	30.18%	15.45%
GeoMean	**45.14%**	**22.20%**	**54.80%**	**19.80%**

better performance. Compared with ViK$_S$, on both kernels, ViK$_O$ exhibits about 20% lower performance overhead.

We also evaluated ViK-protected kernels using UnixBench, which includes benchmarks that test the performance of a UNIX-like system [1]. It generates a system index score as an overall indicator of the performance. As shown in Table 5, the results are similar to the average percentages of increased latency in Table 4. In summary, micro benchmark results show that the ViK-protected OS kernels incur around 22% and 20% runtime overhead on Android and Linux kernels, respectively.

Memory overhead. To measure the memory overhead of ViK-protected kernels, we checked the total amount of memory used by each kernel in `/proc/meminfo`. We measured the memory usage (1) after the system finished booting, and (2) after running LMbench and report the results in Table 6. When ViK aligned memory addresses by 64 bytes, the overall memory overhead was around 42%. ViK achieved much lower memory overhead when it employed the alignment strategy as described in Table 1 where 16-byte alignment is used for objects smaller than 256 bytes and 64-byte alignment is used for other objects. There is no difference in memory usage between ViK$_S$ and ViK$_O$ mode because they allocate the same number of objects. The major source of memory overhead is the amount of memory added to structs to guarantee the alignment. In our implementation of ViK, we used the constants shown in Table 1. For lower memory overhead, ViK will need various sets of M and N that are optimally calculated for different sizes of kernel objects, which requires a more complex implementation that we leave as future work.

Table 6: Memory overhead imposed by ViK on each kernel.

Memory alignment	After Reboot (%) Ubuntu	After Reboot (%) Android	After Bench (%) Ubuntu	After Bench (%) Android
Table 1	13.08%	16.01%	25.03%	28.30%
64 bytes	42.42%	43.98%	41.69%	43.89%

Table 7: The performance and memory overhead on ViK$_{TBI}$–protected Android kernel.

Android kernel 4.14 — ViK$_{TBI}$ UnixBench benchmarks	Overhead	LMbench benchmarks	Overhead
Dhrystone 2	0%	Simple syscall	0%
DP Whetstone	0%	Simple fstat	0%
Execl Throughput	0%	Simple open/close	0.9%
File Copy 1024 bufsize	1.0%	Select on fd's	0.2%
File Copy 256 bufsize	6.3%	Sig. handler installation	0%
File Copy 4096 bufsize	0%	Sig. handler overhead	0%
Pipe Throughput	0%	Protection fault	0%
Pipe-based Ctxt. Switching	0%	Pipe	0%
Process Creation	1.1%	AF_UNIX sock stream	2.1%
Shell Scripts (1 concurrent)	0%	Process fork+exit	0%
Shell Scripts (8 concurrent)	0%	Process fork+/bin/sh -c	0%
System Call Overhead	0%		
GeoMean	**1.91%**	**GeoMean**	**0.72%**
Memory overhead			
After Reboot	7.80%	After Bench	17.50%

Performance of ViK$_{TBI}$. The use of TBI and the reduced number of inspection functions make the runtime overhead of ViK$_{TBI}$ negligible (<2%) as shown in Table 7. The memory overhead of ViK$_{TBI}$ is low: 8% after booting and 17% after running benchmarks. We believe the performance of ViK$_{TBI}$ is sufficiently low to be deployed on customer-facing devices.

8 LIMITATIONS

Protection scope. As in Section 3, currently ViK does not handle stack pointers, i.e., those pointing to any part of stack frames, because such pointers have a short lifetime bounded by a function, and thus, attackers have a limited time to access such pointers, which makes stack pointers hard to exploit. However, we believe that ViK can be extended for preventing stack-based temporal safety violations by using a similar mechanism for heap objects. Also, inspecting only the very first pointer operation of a UAF-unsafe pointer value in a function might cause false negatives at the time when the pointer operation executes, but, we showed that ViK and ViK$_{TBI}$ could catch this case as discussed in Section 7.3.

Static analysis. ViK finds pointer operations using UAF-unsafe pointers in a flow- and path-sensitive manner. We bypass common challenges of static analysis (*e.g.*, scalability) by limiting the range of static analysis to individual modules. However, this design decision limits the potential of ViK's optimizations. We expect ViK to have even lower runtime overhead without sacrificing the security guarantees if we can apply inter-procedural and inter-modular optimizations.

Arbitrary memory read and write. Arbitrary memory read and write vulnerabilities, that are not based on temporal meory safety violations, may allow attackers to tamper the internal state of ViK, which enables Object ID forging. We argue that such vulnerabilities

are rare in real world. These vulnerabilities can be addressed by other defenses.

Implementation details. Although ViK allows users to determine the values of two parameters (M and N) by analyzing sizes of all involved objects in the target program, manual effort is required to make optimal decisions. Using large slots may cause significant memory fragmentation, which may negatively impact runtime performance. Therefore, beyond identifying sizes of memory objects, automatically suggesting the optimal constants would be helpful to prevent unnecessary memory overhead. Also, ViK currently does not support objects larger than 4 KB. We leave this improvement as future work. In addition, ViK does not support using different sets of constants at the same time, which we deem as pure engineering effort. During instrumentation, base identifier calculation functions with different constants can be injected for each pointer operation, depending on the size of each object and predefined constants. We can further reduce memory overhead and cover more memory objects that are larger than 4 KB by implementing support of multiple sets of constant values.

Shifting pointers. The current implementation of ViK does not specially handle shift operations for pointers. Therefore, shifting a pointer (e.g., ptr = ptr » 4) and simply using it without a proper restoration process of the pointer will make a system panic. Even though we have not met such cases yet, ViK's instrumentation pass may need to handle them provided shifting operations remove/-modify object IDs in pointers. We leave this limitation as future work.

57-bit memory address. For CPUs using the 57-bit linear-address space with 5-level paging [28], we can only use the most significant 7 bits of the virtual address. Hence, we have to use 7-bit object IDs and inspect pointer operations accessing the base address of memory objects similar to ViK$_{TBI}$.

9 RELATED WORK

Access validation. Prior mitigation work validates memory accesses in ways that are similar to ViK [23, 25, 32], but they all introduce significant runtime overhead and false positives. ViK achieves better performance than any other access validation approaches. Moreover, these defenses have compatibility issues that ViK does not have: CETS does not support multi-threading [25], and MemSafe requires performing a full-program data-flow analysis [32]. Also, PTAuth suffers from the search time for the base address from interior pointers pointing to the middle of objects [23]. While ViK can search the base address in constant time, regardless of the size of objects, searching time of PTAuth linearly increases upon the size of objects (for a 1024-byte object, PTAuth has to run a PAC instruction 64 times in the worst case). In Linux kernel, it is very common to use such interior pointers, and thus, PTAuth would experience high performance overhead for searching base addresses of memory objects.

Pointer invalidation. Many systems attempt to detect the creation of dangling pointers by tracking reference relationships between pointers and objects [11, 18, 30, 34, 38]. Their designs differ regarding the format of the metadata and the manner in which the metadata is managed. CRCount uses a pointer bitmap to represent locations of heap pointers for reference counting [30]. pSweeper invalidates dangling pointers through another thread that runs in the background, managing a live pointer list and sweeping dangling pointers [20]. DangSan employs an append-only per-thread pointer logger for each memory object [34]. DANGNULL records the relationship between objects in a hierarchical structure called shadowObjTree [18]. However, common to all approaches is the existence of the joint metadata, which imposes high runtime and memory overhead, especially for multi-threaded programs. Additionally, these approaches suffer from propagations of type-unsafe pointers and non-pointer type variables that have pointer values, because of the difficulty of achieving complete data flow analysis [18, 30, 34, 38].

Safe memory allocation. Another type of UAF mitigation is to prevent reusing unsafe address spaces when a new object is allocated [7, 10, 12, 26, 31]. These approaches typically suffer from high memory overhead caused by their object allocation or memory management policies. FFmalloc [35] and Markus [6] showed good memory and performance overhead in user-space programs. However, ViK has much lower memory overhead in allocation-intensive programs (2.42%) than FFmalloc (about 53%) and Markus (about 40%). Also, these approaches are not intended to be used by OS kernels.

Hardware-based approach. WatchdogLite proposes a new instruction set (Intel's Instruction Set Architecture extension) for preventing out-of-bound accesses and UAF errors through compiler support [24]. BOGO utilizes bounds metadata managed by the Intel MPX for providing temporal memory safety [39]: When memory is deallocated, BOGO checks the bound metadata and invalidates dangling pointers. Although both approaches heavily rely on hardware features, they all impose significant runtime overhead.

10 CONCLUSION

Temporal memory safety violations are critical, and it is challenging to enforce a temporal memory safety in an efficient, strong, and flexible (widely applicable) manner. In this paper, we propose a novel approach, ViK, that detects UAF exploits with no false positives. Also, as our evaluation indicates, ViK imposes low overhead, and is a practical mitigation.

ACKNOWLEDGMENTS

Many thanks to the anonymous referees for their thoughtful reviews. We would also like to thank our shepherd, Changhee Jung.

This material is based upon work supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A4A1029650), the Defense Advanced Research Projects Agency (DARPA) HR001118C0060 and FA875019C0003, the Office of Naval Research (ONR) KK1847, and Samsung Research, Samsung Electronics.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

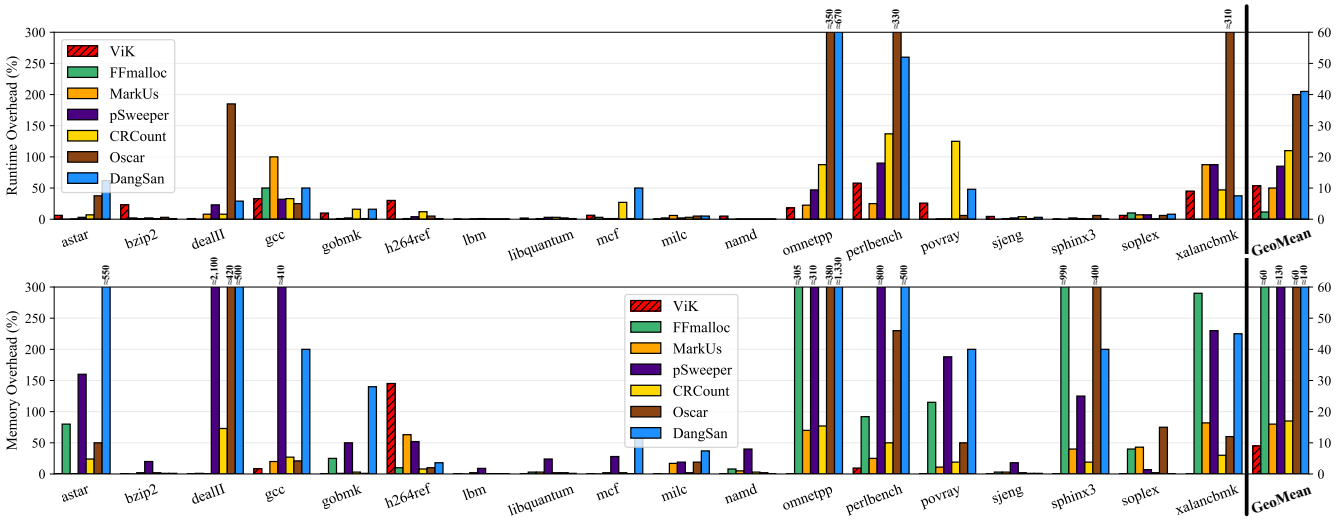

Figure 5: Runtime and memory overhead comparison of the user-space implementation of ViK with FFmalloc, MarkUs, pSweeper, CRCount, Oscar, and DangSan on the SPEC CPU 2006 benchmark programs.

A APPENDIX

A.1 Running Example of the Static Analysis

Listing 3 shows an example of our static analysis result. In the `ptr_ops` function, `unsafe_ptr` has a UAF-unsafe return value from the `get_obj` function, and thus, is unsafe even though the pointer is a stack variable. On the other hand, `safe_ptr` is a safe pointer value (because the pointer value has not been stored in the heap or a global variable) and its operations are not inspected until the point where it may turn into an unsafe pointer value (from Line 23). However, the result of the function call at Line 23 does not affect the UAF-safety of the `safe_ptr`'s pointer operation at Line 26 since the code is under the `else` condition. Hence, ViK does not inspect the pointer operation on Line 26; however, ViK must inspect the pointer operation on Line 30. ViK omits the inspection of `unsafe_ptr`'s pointer operation on Line 31 because the unsafe pointer value has already been inspected by ViK *and* it does not contain a new unsafe pointer value copied from somewhere. If a new unsafe pointer value might be in `unsafe_ptr`, ViK will inspect the pointer operation.

Our static analysis can confirm that an argument of the `add` function is a safe pointer value whenever the function is called; thus, ViK does not inspect the pointer operation using the argument on Line 4. On the contrary, `unsafe_ptr` is used as an argument of the `sub` function. Hence, the pointer operation inside the `sub` function must be inspected, even though the pointer value has been inspected before the function is called.

As shown in the example, our static analysis is flow-sensitive and path-sensitive, which not only significantly reduces the performance overhead, but also helps provide robust security guarantees against UAF attacks.

A.2 User-Space Implementation

We also implement ViK for C and C++ user-space programs. The user-space version of ViK is the same as the kernel-space ViK except for the following aspects:

- User-space programs use different allocators than the kernel, so the instrumentation pass creates appropriate wrappers for memory allocations such as malloc and calloc.
- In user-space, valid pointer values have the first 16 bits as 0, instead of 1 in the kernel. The *inspect*() function is changed in user-space ViK to account for this difference.
- Programs may use shared libraries that are ViK-unaware. ViK can be used in programs with ViK-unawre libraries. However, similar to the other compiler-based approaches [18, 30, 34], if a library is not instrumented by the static instrumentation pass, pointer values that come from the ViK-unaware shared library cannot be inspected in ViK-protected programs.

A.3 Performance: User-Space Programs

We evaluated the user-space ViK implementation by measuring the performance of ViK-protected C and C++ programs of SPEC CPU 2006. To evaluate the runtime and memory overhead of ViK$_O$ (16-byte aligned), We converted the LLVM instrumentation pass into a Link-Time Optimization (LTO) module and compiled the programs with the LTO module enabled. The experimental results are shown in Figure 5. For clarity, we also include in the figure the overhead numbers of applying six state-of-the-art user-space runtime UAF protections (FFmalloc, MarkUs, pSweeper, CRCount, Oscar, and DangSan) on the same programs.[3]

Runtime overhead. ViK has average runtime overhead of 10.6%, higher than FFmalloc (2.3%) and same as MarkUs within rounding error. PTAuth showed 26% of the runtime overhead on average for

[3]The performance numbers of the previous work are extracted from the original papers or provided by their authors.

```
struct obj *global_ptr = NULL;

void add(struct obj *ptr) {
  *ptr += 5; /* safe */
}
void sub(struct obj *ptr) {
  *ptr -= 5; /* unsafe -> inspect() */
}
void make_global(struct obj *ptr) {
  global_ptr = ptr;
}
void ptr_ops(int arg) {
  struct obj *safe_ptr = malloc(4); /* safe */
  struct obj *unsafe_ptr = get_obj(); /* unsafe */

  *safe_ptr = 10; /* safe */
  *unsafe_ptr = 10; /* unsafe -> inspect() */

  add(safe_ptr);
  sub(unsafe_ptr);

  if( arg == 0 ) {
    make_global(safe_ptr); /* safe -> unsafe */
  }
  else {
    *safe_ptr = 10; /* safe */
    global_ptr = malloc(4);
  }

  *safe_ptr = 0; /* unsafe -> inspect() */
  *unsafe_ptr = 0; /* unsafe -> restore() */
  ...
}
```

Listing 3: An example of the static analysis result. Our flow-sensitive and path-sensitive static analysis helps ViK reduce the performance overhead significantly as well as provide robust security guarantees against UAF attacks.

several SPEC 2006 benchmarks (i.e., `bzip2`, `mcf`, `milc`, `gobmk`, `sjeng`, `libquantum`, `h264ref`, `lbm`, and `sphinx3`), ViK imposes around 1% runtime overhead on average for the same benchmarks. ViK performs better than the other five defenses except for FFmalloc when we compare the average overhead on the most pointer-intensive 8 benchmarks in terms of the number of memory allocations and pointer operations (`perlbench`, `omnetpp`, `mcf`, `gcc`, `povray`, `milc`, `xalancbmk`, `astar`, `soplex`, and `gobmk`)—ViK incurs average runtime overhead of about 20%, while it is 25% for MarkUs, 27% for pSweeper, 48% for CRCount, 107% for Oscar, and 128% for DangSan. ViK performs better than FFmalloc for gcc (33.03% and 53.7%, respectively) that uses the largest amount of memory among the benchmarks [35]. This indicates that ViK would have better runtime performance than FFmalloc for programs which consumes a large amount of memory.

Compared to other defenses, ViK shows better or similar runtime overhead on all but two programs, which are `bzip2` and `h264ref`. This is because in ViK, pointer dereferences have a larger impact on the runtime overhead than memory allocations or deallocations. These two programs have relatively low numbers of memory allocations and deallocations but possess high numbers of pointer dereferences, which are unideal for ViK. For example, during an execution of `bzip2`, the `malloc` function executed 8 times at the beginning of its compression routine and 6 times at the beginning of its decompression routine, which are much lower than the numbers for other programs.

Memory overhead. We measured the memory overhead of ViK-protected user-space programs by taking the maximum resident set sizes (RSS). ViK incurs average memory overhead of about 9%, compared with 61% for FFmalloc, 16% for MarkUs, 130% for pSweeper, 17% for CRCount, 60% for Oscar, 140% for DangSan. Overall, ViK achieves similar or lower memory overhead than the other solutions on all tested programs except for `h264ref`. We found that the majority of memory allocations in `h264ref` are small-sized, which severely penalizes ViK due to its memory alignment enforcement. This is supported by ViK's performance on the most four allocation-intensive benchmarks `perlbench`, `xalancbmk`, `omnetpp`, and `dealII`—ViK incurs much less memory overhead (2.42%) than the others (about 53% for FFmalloc, 40% for MarkUs, and 50% for CRCount).

REFERENCES

[1] 2018. *Byte-unixbench: A Unix benchmark suite.* https://github.com/kdlucas/byte-unixbench.

[2] 2019. *Arm A-Profile Architecture Developments: Armv8.5-A.* https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[3] 2019. *Exploit Database.* https://www.exploit-db.com.

[4] 2019. *KASAN: remove use after scope bugs detection.* https://kernel.googlesource.com/pub/scm/linux/kernel/git/torvalds/linux/+/7771bdbbfd3d6f204631b6fd9e1bbc30cd15918e.

[5] 2019. *White Paper: ARM v8.5-A Memory Tagging Extension.* ARM.

[6] Sam Ainsworth and Timothy M Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland).* San Francisco, CA.

[7] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium (Security).* Washington, DC, 177–192.

[8] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium (Security).* Montreal, Canada.

[9] ARM. [n.d.]. *Address spaces in Armv8-A.* https://developer.arm.com/architectures/learn-the-architecture/memory-management/address-spaces-in-armv8-a.

[10] Emery D Berger and Benjamin G Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Ottawa, Canada, 158–168.

[11] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA).* Minneapolis, MN, 133–143.

[12] Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *Proceedings of the 26th USENIX Security Symposium (Security).* Vancouver, BC, Canada, 815–832.

[13] Vincenzo Frascino. 2019. ARM v8.5 Memory Tagging Extension. In *Linux Plumbers Conference 2019.* Lisbon, Portugal.

[14] Intel. [n.d.]. *5-Level Paging and 5-Level EPT.* https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[15] RISC-V International. 2019. *RISC-V Instruction Set Manual.* https://github.com/riscv/riscv-isa-manual.

[16] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS).* Vienna, Austria, 380–392.

[17] The kernel development community. 2019. *The Linux Kernel 5.9.0-rc3 documentation: Application Data Integrity (ADI).* https://www.kernel.org/doc/html/latest/sparc/adi.html.

[18] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS).* San Diego, CA.

[19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, J Ekberg, and N Asokan. 2019. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security).* Santa Clara, CA, 781–797.

[20] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*

(CCS). Toronto, Canada, 1635–1648.

[21] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*. San Diego, CA, 279–294.

[22] MIPS. 2015. *MIPS Architecture For Programmers Volume III: MIPS64/microMIPS64TM Privileged Resource Architecture*. https://www.mips.com/?do-download=the-mips64-and-micromips64-privileged-resource-architecture-v6-03.

[23] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Vancouver, Canada.

[24] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2014. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*. Orlando, FL.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices* 45, 8 (2010), 31–40.

[26] Gene Novark and Emery D Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL, 573–584.

[27] Oracle. 2019. *Oracle Solaris 11.3 Programming Interfaces Guide: Using Application Data Integrity (ADI)*. https://docs.oracle.com/cd/E53394_01/html/E54815/gqajs.html.

[28] Lenovo Press. 2021. *Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux*. https://lenovopress.com/lp1468.pdf.

[29] Qualcomm. 2017. *Pointer Authentication on ARMv8.3*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[30] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C+. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[31] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, 2389–2403.

[32] Matthew S Simpson and Rajeev K Barua. 2013. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience* 43, 1 (2013), 93–128.

[33] Inc. Sun Microsystems. 2007. *OpenSPARC T2 Core Microarchitecture Specification*. https://www.oracle.com/technetwork/systems/opensparc/t2-06-opensparct2-core-microarch-1537749.html.

[34] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, Serbia, 405–419.

[35] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Vancouver, Canada.

[36] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, 781–797.

[37] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, 414–425.

[38] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[39] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Providence, RI, 631–644.