UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages

Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee
School of Computer Science, Georgia Institute of Technology

ABSTRACT

The operating system kernel is the de facto trusted computing base for most computer systems. To secure the OS kernel, many security mechanisms, e.g., kASLR and StackGuard, have been increasingly deployed to defend against attacks (e.g., code reuse attack). However, the effectiveness of these protections has been proven to be inadequate—there are many information leak vulnerabilities in the kernel to leak the randomized pointer or canary, thus bypassing kASLR and StackGuard. Other sensitive data in the kernel, such as cryptographic keys and file caches, can also be leaked. According to our study, most kernel information leaks are caused by uninitialized data reads. Unfortunately, existing techniques like memory safety enforcements and dynamic access tracking tools are not adequate or efficient enough to mitigate this threat.

In this paper, we propose UniSan, a novel, compiler-based approach to eliminate all information leaks caused by uninitialized read in the OS kernel. UniSan achieves this goal using byte-level, flow-sensitive, context-sensitive, and field-sensitive initialization analysis and reachability analysis to check whether *an allocation has been fully initialized when it leaves kernel space; if not, it automatically instruments the kernel to initialize this allocation*. UniSan's analyses are conservative to avoid false negatives and are robust by preserving the semantics of the OS kernel. We have implemented UniSan as passes in LLVM and applied it to the latest Linux kernel (x86_64) and Android kernel (AArch64). Our evaluation showed that UniSan can successfully prevent 43 known and many new uninitialized data leak vulnerabilities. Further, 19 new vulnerabilities in the latest kernels have been confirmed by Linux and Google. Our extensive performance evaluation with LMBench, ApacheBench, Android benchmarks, and the SPEC benchmarks also showed that UniSan imposes a negligible performance overhead.

Keywords

kernel information leak; uninitialized read; reachability analysis; initialization analysis; memory initialization

CCS'16, October 24-28, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2976749.2978366

1. INTRODUCTION

As the de facto trusted computing base (TCB) of computer systems, the operating system (OS) kernel has always been a prime target for attackers. By compromising the kernel, attackers can escalate their privilege to steal sensitive data in the system and control the whole computer. There are three main approaches to launch privilege escalation attacks: 1) direct code injection attack; 2) `ret2usr` attacks [18]; and 3) code reuse attacks [37]. DEP (Data Execution Prevention) protection has been deployed to defeat traditional code injection attacks. Intel and ARM have recently introduced new hardware features (i.e., SMEP and PXN) to prevent `ret2usr` attacks [17]. As a result, code reuse attacks are becoming more prevalent. A general direction to defeat code reuse attacks is randomization, e.g., kernel address space layout randomization (kASLR) [9] and StackGuard [10] have been adopted by the latest kernels. kASLR aims to prevent attackers from knowing where the code gadgets are, and StackGuard [10] aims to prevent attackers from corrupting return addresses.

By design, the effectiveness of kASLR and StackGuard completely relies on the confidentiality of the randomness—leaking any randomized pointer or the stack canary will render these mechanisms useless. Unfortunately, information leak vulnerabilities are common in OS kernels. For example, Krause recently found 21 information leaks in the Linux kernel [19]. According to CVE Details [11], kernel information leak vulnerabilities have not only become more prevalent than buffer overflow vulnerabilities, but the number of kernel information leaks is also increasing with continuously introduced new features. Moreover, the leaked kernel memory may also contain other sensitive data, such as cryptographic keys and file caches. For these reasons, preventing kernel information leak is a pressing security problem.

There are four main causes of information leaks: uninitialized data read, buffer overread, use-after-free and logic errors (e.g., missing privilege check). Among these root causes, uninitialized read is the most common one. According to a Linux kernel vulnerabilities survey [6], 37 information leak vulnerabilities were reported from January 2010 to March 2011, 28 of which were caused by uninitialized data read. Similarly, we analyzed the causes of the kernel information leaks reported after 2013. Our study also revealed that about 60% kernel information leaks are caused by uninitialized data read (Figure 1). However, while many memory safety techniques [26–28, 35] have been proposed to prevent buffer overread and use-after-free, the prevention of uninitialized data leaks is still an open problem.

Preventing uninitialized data leaks is challenging for three reasons. First, since data write and read are frequent in programs, detecting uninitialized data reads by tracking these operations will always introduce unacceptable performance overhead. For example, Memo-

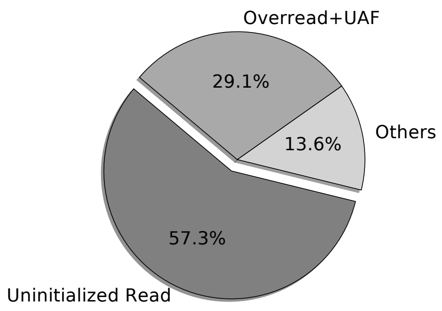

Figure 1: The root causes of kernel information leaks reported after 2013: uninitialized data read, spacial buffer overread + use-after-free, and others (e.g., missing permission check). Most leaks are caused by uninitialized data read.

rySanitizer [39] and kmemcheck [31] check every memory read and write to detect uninitialized data reads, thus incurring a performance overhead of more than three times. Second, uninitialized data reads are actually quite common in programs. Specifically, compilers often introduce padding bytes in data structures to improve performance. These padding bytes are usually uninitialized, but as long as the uninitialized data is not used (e.g., dereferenced as pointer or leaked), its access does not cause any problem. Since the padding is introduced by the compiler, developers are usually not aware of the potential data leaks—they need to be convinced such problems exist before they will fix the programs (see §2 for more details). Third, uninitialized data leaks often occur across multiple procedure boundaries—the uninitialized data is always passed to leaking functions (e.g., `copy_to_user`); hence intra-procedure analysis based detections (e.g., `-Wuninitialized` provided by compilers) cannot catch uninitialized data leaks.

In addition to tracking every data read and write, researchers have attempted an alternate approach—*force initializing*. For example, PaX's `STACKLEAK` plugin [40] clears the used kernel stack when the control is transferred back to user space, which effectively prevents data leaks between syscalls. However, `STACKLEAK` cannot prevent the leaking of uninitialized data generated during the same syscall. Also, this can introduce a significant performance overhead (see Table 4). Split kernel [20] instead clears stack frame whenever it is allocated (i.e., a function is called). Split kernel provides stronger security, but its performance overhead is even more significant. Peiró et al. [32] proposed using model checking to detect kernel stack allocations that have never been `memset` or assigned. However, this approach has obvious limitations. For example, since it neither tracks the propagation of uninitialized data nor handles partial initialization, it has high false negative and false positive rates. Moreover, none of these approaches can handle kernel heap leaks.

In this paper, we propose a novel mechanism, UniSan (Uninitialized Data Leak Sanitizer), to prevent kernel leaks caused by uninitialized data reads. Similar to `STACKLEAK`, UniSan is an automated compiler-based approach; that is, it does not require manual modifications to the source code and can transparently eliminate leaks caused by data structure padding or improper initialization. At the same time, UniSan also overcomes all the aforementioned limitations of previous force-initialization approaches. More specifically, UniSan leverages an inter-procedural static analysis to check *1) whether an allocation ever leaves kernel space, and 2) the allocation is fully initialized along all possible execution paths to the leak point*. The analysis is conservative—as long as there is one byte of an allocation that cannot be proved to have been initialized in any possible execution path before leaving the kernel space, it is considered *unsafe*; hence UniSan has no false negatives. It also handles both stack and heap allocations. At the same time, to improve the precision and thus minimize false positives, UniSan's analysis is fine-grained, which tracks each byte of an allocation in a flow-sensitive and context-sensitive manner. Once UniSan detects an unsafe allocation, it then instruments the kernel to zero the uninitialized portion of the allocation. In this way, UniSan can completely prevent kernel information leaks caused by uninitialized reads. Note that by being conservative, UniSan may still have false positives; however, initializing allocations that will never be leaked will not break the semantics of the kernel, but will just introduce unnecessary performance overhead. Because UniSan's instrumentation is semantic-preserving, robustness is guaranteed.

We have implemented UniSan based on the LLVM compiler [22]. UniSan consists of two components. The first is the *unsafe allocation detector*, which conservatively reports all potentially unsafe allocations. The second is the *unsafe allocation initializer*, which zeros the uninitialized memory by inserting zero-initialization, `memset`, or changing the allocation flags.

We have applied UniSan to the latest mainline Linux and Android kernels to evaluate the effectiveness and efficiency of UniSan in preventing kernel leaks. For effectiveness evaluation, we first tested UniSan over 43 recently discovered kernel information leak vulnerabilities resulting from uninitialized reads. UniSan successfully detected and prevented all these known vulnerabilities. Moreover, the unsafe allocation detector of UniSan has identified many new uninitialized data leak vulnerabilities in the latest Linux kernel and Android kernel, 19 of which have been confirmed by the Linux maintainers and Google.

We also measured UniSan's performance impacts on kernel operations, server programs, and user-space programs using multiple benchmarks, including LMbench [24], ApacheBench, Android benchmarks, and the SPEC CPU Benchmarks. The evaluation results showed that UniSan incurs a negligible performance overhead (less than 1% in most cases) and is thus much more efficient than existing solutions (e.g., `STACKLEAK`).

We believe that since UniSan is robust, effective, and efficient, it is ready to be adopted in practice to prevent uninitialized data leaks. In summary, we make the following contributions:

- **Survey of kernel information leaks**: We studied all the reported kernel information leaks vulnerabilities since 2013. We analyzed their root causes and corresponding defenses. We also discussed with kernel developers about how to prevent information leaks caused by data structure paddings.

- **Development of new protection mechanism**: We designed and implemented UniSan, an automated, compiler-based scheme to eliminate kernel information leaks caused by uninitialized data reads, which is the main cause of kernel leaks. UniSan has been successfully applied to the latest mainline Linux and Android kernels (yet not limited to kernels). UniSan is a practical, ready-to-use security protection scheme, and will be open-sourced for broader adoption.

- **Discoveries of new vulnerabilities**: During our evaluation, UniSan discovered many previously unknown information leak vulnerabilities in the latest Linux and Android kernels, 19 of which have been confirmed by Linux maintainers and Google.

In the rest of the paper, we introduce our kernel leak study (§2) and describe the overview (§3), design (§4) and implementation (§5) of UniSan. Then we evaluate UniSan in §6. Related work is summarized in §7. We discuss the limitations of UniSan in §8, and conclude in §9.

2. AN ANALYSIS OF KERNEL INFORMATION LEAKS

In this section, we provide a background on kernel information leaks, including their security implications and root causes. Then we discuss uninitialized data reads and how such vulnerabilities should be fixed from the developers' perspectives.

2.1 Kernel Information Leaks

Kernel information leak vulnerabilities can cause severe security consequences. First, as mentioned in §1, with the deployment of kASLR [9] and stack canary [10], a general prerequisite for many attacks (e.g., code reuse attack) is learning the randomized addresses and canary, which can be accomplished by exploiting kernel information leak vulnerabilities. Further, as the TCB of the whole system, the OS kernel also has access to many other types of sensitive information, such as encryption keys, file cache, and remaining data of terminated processes, etc. For performance reasons, memory pages allocated to store such information may not be cleared when they are released to the kernel. As a result, kernel information leak vulnerabilities also allow attackers to access such sensitive information. For example, [42] showed that an uninitialized data leak is used to leak the entropy source for `srandom`. In short, it is critically important to prevent kernel information leaks.

Kernel information leaks are also very common and have many causes. According to a previous study [6] of Linux kernel vulnerabilities discovered between January 2010 and March 2011, kernel information leak was ranked the second most common vulnerability, which is even more common than buffer overflow vulnerability. Specifically, of the total of 37 leak vulnerabilities, 28 were caused by uninitialized reads, 7 were caused by buffer overread, and two by other miscellaneous causes.

Since the aforementioned study [6] is already five-year old, we conducted another study of kernel information leak vulnerabilities reported after 2013 [11], which contained 103 leaks in total. The result is shown in Figure 1. The majority of kernel leaks are caused by uninitialized reads. Buffer overread is also a common cause, in which the size of reading is not properly checked. Use-after-free bugs may be exploited to leak the data of newly allocated objects or manipulate the size and address of the read. Other causes include missing permission check, race condition, or other logic errors. Since uninitialized read is the most common cause of kernel leaks, our work focuses on preventing uninitialized data leaks.

2.2 Uninitialized Data Leaks

An uninitialized data leak occurs when an allocated stack or heap object is not properly initialized when being copied to the outside world (e.g., to user space, network, or file systems). If the memory occupied by the object is used to store sensitive data (e.g., addresses), attackers can exploit this to leak such information. Note that *using* uninitialized memory is a type of memory safety error and can lead to undefined behavior. For this reason, modern compilers (e.g., the `-Wuninitialized` option in GCC) can generate warnings when variables are used without proper initialization. However, this compiler feature employs only an *intra-procedure* analysis and cannot handle many common cases (e.g., reading the uninitialized data through its pointer). Most uninitialized data leaks happen across multiple function boundaries (e.g., calling `copy_to_user` in Linux), so they can be identified only using *inter-procedural* analysis. Also, data can be propagated through various channels (e.g., network or file systems). In short, existing compiler checks are not effective in finding uninitialized data leaks in large-scale, sophisticated programs, like the Linux kernel. Moreover, uninitialized data reads may not be harmful if they are not dereferenced or leaked; reporting all of

```
   /* File: net/x25/af_x25.c */
   int x25_rx_call_request(struct sk_buff *skb,
                           struct x25_neigh *nb,
                           unsigned int lci) {
 ★   struct x25_dte_facilities dte_facilities;
     /* some fields of dte_facilities are not initialized */
 !   x25_negotiate_facilities(..., &dte_facilities);
     ...
     /* passed to the external */
     makex25->dte_facilities= dte_facilities;
     ...
   }
   static int x25_ioctl(struct socket *sock,
                        unsigned int cmd,
                        unsigned long arg) {
     ...
     /* leak uninitialized fields of dte_facilities */
 ⊙    copy_to_user(argp, &x25->dte_facilities,
                   sizeof(x25->dte_facilities));
     ...
   }
```

Figure 2: New kernel leak in the `x25` subsystem—6 fields of `dte_facilities` are not initialized and leaked in another function `x25_ioctl`. ⋆ denotes memory allocation, ! marks incorrect initialization, ⊙ notes a leaking point.

```
   /* File: net/wireless/nl80211.c */
   static int nl80211_dump_station(struct sk_buff *skb,
                                   struct netlink_callback *cb) {
 ★   u8 mac_addr[ETH_ALEN]; /* ETH_ALEN = 6 */
     ...
 !   err = rdev->ops->dump_station(\
             &rdev->wiphy,
             wdev->netdev, sta_idx, mac_addr, &sinfo);
     /* mac_addr is uninitialized but sent out via nla_put()
      * inside nl80211_send_station() */
 ⊙    if (nl80211_send_station(skb, NL80211_CMD_NEW_STATION,
                                NETLINK_CB(cb->skb).portid,
                                cb->nlh->nlmsg_seq, NLM_F_MULTI,
                                rdev, wdev->netdev, mac_addr,
                                &sinfo) < 0)
       goto out;
     ...
   }
```

Figure 3: New kernel leak in the `wireless` subsystem—the whole 6-bytes array `mac_addr` is not initialized but sent out. ⋆ denotes memory allocation, ! marks incorrect initialization, and ⊙ notes a leaking point.

them will burden or even annoy developers. The following examples demonstrate the common causes of uninitialized data leaks and why existing compiler features cannot detect them. Please note that all leakage examples listed in this paper are newly discovered by our system UniSan and fixed only in the mainstream repository; most deployed kernels may not be patched yet.

2.2.1 *Missing Element Initialization*

The simplest and most common case of an uninitialized data leak is when the developers fail to properly initialize all fields of an object or memory of a buffer. Figure 2 shows a real kernel leak vulnerability in the x25 module. Specifically, the object `dte_facilities` is supposed to be properly initialized in `x25_negotiate_facilities`. However, six fields are still not initialized. The object is then propagated to the external heap object `makex25` and finally leaked to userland in another function `x25_ioctl`. As we can see, detecting such leaks would require sophisticated inter-procedural data-flow analysis, which is not available during normal compilation.

Figure 3 shows another real kernel leak vulnerability caused by program design and developers. The 6-bytes buffer `mac_addr` is allocated on stack. It is supposed to be initialized in the `dump_station` function before it is sent out. However, since `dump_station` is an interface function that has different implementations for different protocols written by different programmers, it is hard to ensure

```
  /* File: drivers/usb/core/devio.c */
  struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
    /* 3-bytes padding inserted for alignment */
  };
  static int proc_connectinfo(struct usb_dev_state *ps,
                               void __user *arg) {
★   struct usbdevfs_connectinfo ci = {
      .devnum = ps->dev->devnum,
      .slow = ps->dev->speed == USB_SPEED_LOW
    };

    /* sizeof(ci) == 8, but only 5 bytes are initialized */
⊙   if (copy_to_user(arg, &ci, sizeof(ci)))
      return -EFAULT;
    return 0;
  }
```

Figure 4: New kernel leak caused by compiler padding. Object ci contains 3-bytes padding at the end, which is copied to userland. Note that ⋆ denotes memory allocation and partial initiation, and ⊙ notes a leakage point.

the object is consistently initialized. Specifically, this buffer is not initialized in the implementation of the `wilc1000` module. As this uninitialized data is actually never *used* by the kernel, it is simply copied outside the kernel boundary; from the perspective of compiler, these cases are not errors. Moreover, `dump_station` is a function pointer (i.e., dereferenced by an indirect call), tracking which requires a complete call graph that is not provided by current compiler features.

2.2.2 Data Structure Padding

Besides developer mistakes, a more interesting source of uninitialized data leaks is compiler-added data structure paddings. In particular, when modern processors access a memory address, they usually do this at the machine word granularity (e.g. 4-bytes on a 32-bits systems) or larger. So if the target address is not aligned, the processor may have to access the memory multiple times to perform shifting and calculating to complete the operation, which can significantly degrade the performance [33]. Moreover, different instructions or architectures could also have their own alignment requirements. For instance, SSE instructions on the x86 architecture require the operands to be 16-bytes aligned, and ARM processors always require memory operands to be 2-bytes or 4-bytes aligned. For these reasons, compilers usually add paddings within a data structure so fields are properly aligned.

The problem is that as these paddings are not visible at the programming language level, and so they are not initialized even when all fields have been properly initialized in the code written by developers and hence may lead to information leaks. Figure 4 illustrates a kernel leak in the USB module caused by data structure paddings. In this case, although developers have explicitly initialized all fields of the stack object `ci`, because of its last 3-byte padding, information can still be leaked when `copy_to_user` is invoked to copy the whole object to user space. To detect such leaks, byte-level analysis is required; object-level or even field level (i.e., field-sensitive) analyses are still not fine-grained enough to catch such leaks.

To summarize, leaking padding bytes is a serious problem for three reasons. First, it is prevalent. Compilers frequently introduce padding for better performance. Padding is even more prevalent when porting programs from 32-bits to 64-bits platforms. Second, inter-procedural and byte-level analysis is required to detect such leaks. Third, it is often not visible to developers. Padding bytes can be leaked even developers have correctly initialized all fields.

After we submitted patches for these vulnerabilities, the compiler-injected padding issue was discussed extensively by the Linux community and became a major concern to the Linux kernel developers

```
★ struct usbdevfs_connectinfo ci = {
    .devnum = 1234,
    .slow = 1
  };
```

Figure 5: An example that fields in a object are initialized with constants in an initializer. LLVM will generate a `global initializer` for this object.

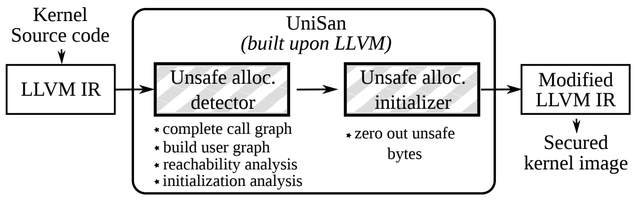

Figure 6: Overview of UniSan's workflow. UniSan takes as input the LLVM IR (i.e., bitcode files) and automatically generates secured IR as output. This is done in two phases: identifying all potentially unsafe allocations and instrumenting the kernel to initialize all detected unsafe allocations.

because, from the developers' perspective, they have properly initialized the data structures and this type of leak is hardly visible even to skilled programmers. On the other hand, from the compilers' perspective, they have the benefit of *not* proactively initializing such padding regions to achieve better performance, because this design decision can be independently made by each compiler according to the C/C++ specification. However, considering its severity, prevalence and more importantly, its non-trivial nature to developers, *we, as well as many kernel maintainers urge the incorporation of UniSan's approach to solve this problem at the compiler level, perhaps as an extra option to the compiler.*

Global initializer in LLVM. Initializers are often used in the kernel; Figure 5 shows an example of an initializer that initializes fields of the allocated object. When the fields are initialized with constants (not variables), LLVM will generate a *global initializer* that will zero the remaining bytes including the padding in the object. In this case, we should not report it as an unsafe allocation. On the other hand, GCC does not use the global initializer to zero the remaining bytes; hence it is still a leak when the whole object is sent out. This example demonstrates that compiler-based analysis is necessary to accurately detect unsafe allocations.

3. OVERVIEW

In this work, we focus on *preventing* kernel information leaks caused by uninitialized data reads. UniSan achieves this goal via a two-phase process (Figure 6). In the first phase, UniSan analyzes the kernel and conservatively identifies all potential unsafe objects that may leave the kernel space without having been fully initialized. Then in the second phase, UniSan instruments the kernel to automatically initialize (zero out) the detected unsafe allocations.

3.1 Problem Scope

The protection target of UniSan is the OS kernel. Since UniSan performs analysis over source code, we assume the source code of the kernel is available. We assume the attackers do not have the kernel privilege (e.g., through loading a malicious kernel driver), but they can be either local or remote. The goal of the attackers is to leak sensitive data (e.g., addresses, cryptographic keys, file content) in the kernel space to the external world (e.g., userland and network). After gathering the information, the attacker can launch more attacks, such as privilege escalation and phishing.

As discussed in §2, kernel information leaks have multiple root causes: uninitialized data read, buffer overread, use-after-free, data race, logical errors, etc. In this work, we focus on uninitialized data read because a majority of kernel leaks are caused by this

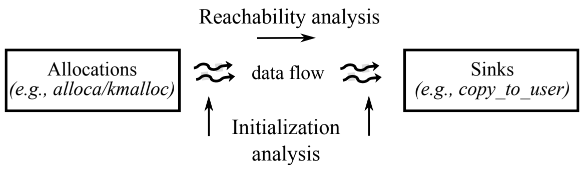

Figure 7: Unsafe allocation detector incorporates a reachability analysis and an initialization analysis to check whether any byte of the allocation is uninitialized when leaves the kernel space.

particular type of vulnerabilities and there is no practical solution yet; whereas other types of vulnerabilities can be addressed by existing techniques and are thus out-of-scope. For instance, buffer overread and use-after-free bugs can be prevented by memory safety techniques [26–28, 35]. Data race can be detected by [14]. Logical errors (e.g., missing permission check) are relatively rare and can be identified by semantic checking techniques [3, 25].

3.2 The UniSan Approach

There are multiple candidate solutions for preventing uninitialized data leaks. The first is zeroing the memory when it is deallocated. Since we cannot know when the deallocated memory will be allocated and used in the future, we have to conservatively zero all deallocated stack and heap objects, which can introduce a significant performance overhead. More importantly, deallocation is not always paired with allocation, such as the case of memory leak, thus introducing false negatives. The second is dynamically tracking the status of every byte in the object, so that we can know exactly if any uninitialized bytes are leaving the kernel space. MemorySanitizer [39] and kmemcheck [31] are based on this approach. However, while they are effective and able to detect general uninitialized data uses, their 3x performance overhead is too high to be used for runtime prevention. The third is selectively zeroing the allocated memory that is detected as unsafe by static analysis. After assessing the effectiveness and performance of these approaches, we chose the third approach—only initializing the unsafe allocations.

Figure 6 illustrates the approach of UniSan. Specifically, UniSan takes as input the LLVM IR (i.e., bitcode files) compiled from the kernel source code, upon which the analysis and instrumentation are performed. Given a stack or heap allocation, UniSan leverages static data-flow (taint) analysis to check whether this allocation can reach the pre-defined sinks, such as `copy_to_usr` and `sock_sendmsg`. Along the propagation path, UniSan also tracks the initialization status of each byte of the allocation. If any byte of the allocation reaches the sink without having been initialized in any possible execution path, UniSan considers it unsafe. After collecting all unsafe allocations, UniSan instruments the IR with initialization code right after them. For stack allocation, UniSan inserts a `memset` or zero-initiailzation to initialize it. For heap allocation, UniSan adds the `__GFP_ZERO` flag to the allocation functions (e.g., kmalloc). Finally, by assembling and linking the instrumented bitcode files, UniSan generates the secured kernel image.

4. DESIGN

In this section, we present the design of UniSan. We first describe the design of *unsafe allocation detector*, including how we generate the complete call graph, perform reachability analysis, and track the initialization status of allocations. Then we describe how *unsafe allocation initializer* instruments the kernel to generate secured kernel image.

4.1 Detecting Unsafe Allocations

Our unsafe allocation detection is essentially a static taint analysis that incorporates two orthogonal analyses: object initialization analysis and sink reachability analysis, as shown in Figure 7. The initialization analysis checks which bytes of the object will be initialized along the paths from allocation to sinks, i.e., which bytes will be assigned with other values. The sink reachability analysis checks which bytes of the object will leave the kernel space along the paths from allocation to sinks, i.e., being passed to sinks. The initialization analysis and reachability analysis are then integrated to detect unsafe bytes—a byte is *unsafe* if it is uninitialized when it leaves kernel space.

The workflow of the detection is as follows. Given the bitcode files of the target program, it first builds a complete and global call graph. Then it parallelly performs the reachability analysis and initialization analysis for each allocation to detect the unsafe bytes. Our analysis is flow-sensitive in that the order of the uses of the allocation is maintained with a dedicated *user-graph* (that will be elaborated in §4.1.4); context-sensitive in that function calls are followed with callsite-specific context; and field-sensitive in that it performs fine-grained tracking for each byte of the allocation. However, to avoid the path explosion problem and make our analysis scalable to the whole kernel, our analysis is path-insensitive.

4.1.1 Defining Sources and Sinks

Both the reachability analysis and initialization analysis track the "taint" status of the allocated bytes from the allocation site to the sink functions. As the first step, we need to pre-define the sources (i.e., allocations) and sinks (i.e., data-leaking functions).

Sources. For stack, all objects are allocated by the `AllocaInst` (i.e., an instruction to allocate memory on the stack) . By handling this instruction, we are able to find all tracking sources on stack. Heap objects can be allocated in many ways. In our current implementation, we include only the standard allocator from SLAB, namely, `kmalloc` and `kmem_cache_alloc`. These heap allocators accept a flag parameter. If the flag contains `__GFP_ZERO` bit, the allocated memory will be initialized with zero. In UniSan, we track only heap allocations without the `__GFP_ZERO` flag. Please note that although UniSan currently does not include custom allocators, it can be easily extended to support them once developers denote the function name and allocation flags (i.e., create the source signature).

Sinks. Under the threat model of UniSan, any function that may send kernel data to userland, network, or file is classified as a sink function. In UniSan, we use two policies to generally define the sinks. We first empirically define a list of known sink functions, based on our study of previous kernel leaks. For example, `copy_to_user` copies data to userland; `sock_sendmsg` sends data to network, and `vfs_write` writes data to files. Although there are various implementations for file writing (for different file systems) and message sending (for different protocols), `vfs_write` and `sock_sendmsg` are the uniformed interfaces, so we can generally catch the sink functions by annotating these functions.

Clearly, there are more sink functions that are not covered by the first step. To eliminate false negatives introduced by an incomplete sink list, we utilized three conservative rules to generally cover additional sinks. These rules are defined based on the fact that under our recursive tracking algorithm (Figure 8), *for any data to leave kernel space, it will always be stored to a non-kernel-stack location (or non-`AllocaInst` to be specific)*, so once we cannot determine that the destination of an store operation is on kernel stack, we treat it as a sink.

- **Rule 1**: A `StoreInst` (i.e., an instruction for storing to memory) is a sink if the destination is not allocated by an `AllocaInst` in kernel;
- **Rule 2**: A `CallInst` (i.e., an instruction for calling a function)

```
/* Unsafe allocation detection algorithm */
UnsafeAllocDetection(Module) {
    for (Alloc in Module) {
        /* user relationships are maintained */
        MergedUnsafeBytes = Array();
        UserGraph = BuildUserGraph(Alloc);
        NextUsers = GetFirstUser(UserGraph);
        RecursiveTrackUsers(Alloc,NextUsers,MergedUnsafeBytes);
        if (IsNotEmpty(MergedUnsafeBytes))
            Report(Alloc, MergedUnsafeBytes);
    }
}
RecursiveTrackUsers(Alloc, NextUsers, MergedUnsafeBytes) {
    /* terminate if all bytes have been inited or sinked */
    if (AllInitied(Alloc) || AllSunk(Alloc))
        return;
    UnsafeBytes = Array();
    for (User in NextUsers) {
        /* Next users of User are recursively tracked */
        if (IsLoadInst(User))
            RecursiveTrackLoad(User, UnsafeBytes);
        else if (IsStoreInst(User))
            RecursiveTrackStore(User, UnsafeBytes);
        else if (IsCallInst(User))
            RecursiveTrackCall(User, UnsafeBytes);
        else if (IsGetElementPtr(User))
            RecursiveTrackGEP(User, UnsafeBytes);
        ...
        /* Unrecognized cases */
        else
            /* assume remaining uninitialized bytes unsafe */
            UnsafeBytes += GetUninitializedBytes(Alloc);
        /* Conservatively merge (union) all unsafe bytes */
        MergedUnsafeBytes += UnsafeBytes;
    }
}
```

Figure 8: The pseudo-code of the recursive tracking algorithm for the unsafe allocation detection.

is a sink if the called value is inline assembly that is not in the whitelist (§4.1.6);

- **Rule 3**: A `CallInst` is a sink if the called function's body is empty (i.e., not compiled into LLVM IR).

4.1.2 Building Global Call-Graph

Since UniSan's analysis is inter-procedural, global call graph is required. To eliminate false negatives, UniSan must conservatively identify all potential targets of indirect calls. To this end, we first collect the address-taken functions, and use the type-analysis-based approach [30, 41] to find the targets of indirect calls. That is, as long as the type of the arguments of an address-taken function matches with the callsite of the indirect call, we assume it is a valid target. Note that we also assume universal pointers (e.g., `char *`, `void *`) and an 8-bytes integer can match with any type.

4.1.3 Recursive Detection Algorithm

With the global call-graph, we conduct the unsafe allocation detection. The algorithm of the detection is shown in Figure 8. In short, given an allocation in a module, we first build its user-graph (that will be elaborated in §4.1.4). After that, we recursively keep track of which bytes of the allocated object have been initialized and which bytes have reached sink functions, by traversing the user-graph. Different users are handled properly. If there are any corner cases that are not recognized before, we conservatively assume the bytes being tracked are unsafe. All unsafe bytes in different paths are concatenated together. That is, a byte is assumed unsafe as long as it is unsafe in any possible path.

4.1.4 Building User-Graph

In LLVM IR, a *user* of a value is an instruction or expression that takes the value as an operand. The unsafe allocation detection

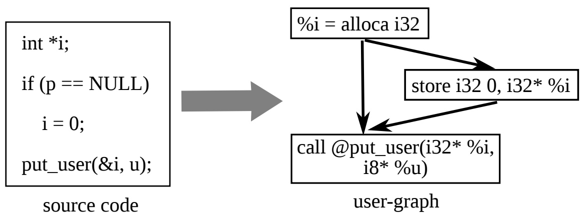

Figure 9: A simple user-graph example.

is designed to be flow-sensitive, which requires considering the order of users. Given the being tracked value (e.g., the allocated value), LLVM framework only tells us all the users but not the relationships (e.g., sequential and parallel relationships) among them. To maintain the relationships of the users, we build the user-graph for the tracked value. Figure 9 shows a simple user-graph example. Instructions that do not use the tracked value will not show up in the graph. More specifically, we first put all the users in the corresponding basic blocks. Users in the same basic block are always in a sequential order. Then we use the `DominatorTree` pass of LLVM to understand the relationships among the involved basic blocks. With this information, we chain all the users together into a user graph. When an alias of the tracked value is generated (e.g., by the `CastInst` and `GetElementPtr` instructions), all the users of the alias will be seamlessly merged into the user-graph as well.

4.1.5 Fine-Grained Status Tracking

UniSan performs unsafe allocation detection in a fine-grained manner (i.e., byte-level analysis). There are several advantages of performing the analysis at byte granularity. First, due to the compiler's padding, initializing all fields of an object does not guarantee that all bytes are initialized (Figure 4). Therefore, byte-level analysis is necessary to detect the uninitialized padding bytes. Second, `union` is widely used in kernel data structures and byte-level tracking can help resolve the issue introduced by field alias. Finally, byte-level detection can precisely filter out safe bytes, so that the instrumentation module can selectively initialize only the unsafe ones, thus further reducing the runtime performance overhead.

To perform the byte-level analysis, we use a buffer to record the initialization and sinking status of every byte in a tracked allocation. Whenever an object is allocated, a buffer of the same size is created. Currently, we only use two bits of the corresponding byte in the buffer to represent the initialization and sinking status of each byte in the original object. To keep track of which bytes are being accessed, at every `GEPOperator` (an instruction or expression for type-safe pointer arithmetic to access elements of arrays and structs) node in LLVM IR, we calculate the offset (into the base of the object) and the size of the obtained element. In our current design, we do not perform range analysis, so if any of the indices of the `GEPOperator` node are not constants, we cannot statically calculate the resulting element. In this case, we conservatively treat all bytes of the allocation as uninitialized and pause the initialization analysis—as long as the reachability analysis determines that the allocation can leave the kernel space, we assume it is unsafe. Since non-constant indices are not common in `GEPOperator` node, our byte-level analysis works well in most cases.

4.1.6 Eliminating False Negatives

As a prevention tool, UniSan aims to eliminate potential false negatives. Clearly, aggressively initializing all stack and heap allocations can guarantee no false negative; however, such a naive approach will introduce unnecessary performance overhead, since most allocations will either never leave kernel space or be properly initialized. To make UniSan more efficient, our principle is to eliminate as many false positives as possible while ensuring no false negative; and whenever we encounter an undecidable problem or an unhandled corner case, we always sacrifice the detection accuracy and assume the tracked allocation is unsafe. In this subsection, we summarize cases that may introduce false negatives and describe how we handle them.

Complete call graph. LLVM's built-in call graph pass does not find callees of indirect calls. As described in §4.1.2, we adopt a conservative type-analysis to find indirect call targets to build a complete and global call graph.

Conservative path merging. There are often many paths from the allocation site to the sink points. And in different paths, the allocated object can be initialized differently. In LLVM IR, the most common cases that can introduce multiple paths include: 1) load and store instructions. These instructions copy the tracked value to somewhere else, creating a new data-flow; 2) indirect call instructions; 3) return instructions; 4) branch instructions. To ensure that no leaks will be missed, UniSan always tracks each path independently and merges the tracking results (when tracking of a path returns) by calculating the union of all unsafe bytes; in other words, a byte is deemed unsafe as long as it is unsafe in one path.

Propagation to alias. Our reachability and initialization analyses are performed in a forward manner (i.e., from source to sink). Whenever an alias of the tracked value is created (e.g., by `CastInst` or `GetElementPtr` instructions), we further track the alias by merging its users into the user-graph of the current tracked value. Therefore, we do not need alias analysis for this case. However, when the tracked value is stored to another value, we need a backward slicing analysis to find the possible aliases of the store-to value, which is difficult for some cases (e.g., global variable). To handle this, we employ the simple basic alias analysis [21]. Additionally, we enforce two conservative policies to eliminate potential false negatives: 1) if we find that the aliases are pointing to a non-stack object (e.g., global variable or heap object), whose data-flow is hard to follow, we assume the tracked value is unsafe; 2) if we find the aliases is a returned value of a function call or a parameter of current function, we also assume the tracked value is unsafe.

Inline assembly. To improve performance, kernel developers commonly write inline assembly. Since inline assembly is not compiled into LLVM IR, our detection cannot be directly applied. To handle inline assembly, we manually whitelisted some safe inline assembly that will not leak the tracked value or store the tracked value to other places. All other inline assembly functions are conservatively treated as sinks.

4.2 Instrumenting Unsafe Allocations

After identifying all unsafe allocations, the initializer module of UniSan further instruments the kernel to initialize the identified unsafe allocations by zeroing the unsafe bytes. In particular, the initialization code is inserted right after the allocation (e.g., the stack `Alloca` and `kmalloc`). Since the detection module reports unsafe allocation at the byte-level, in many cases, we do not have to initialize the whole allocated object but only the unsafe bytes. In particular, for stack allocation, we use `StoreInst` to zero the unsafe bytes if they have a continuous size of less than or equal to 8; otherwise, `memset` is used for zeroing the bytes. Heap allocation is also initialized in a similar way except that the `__GFP_ZERO` flag is passed to the heap allocator to initialize the memory if all bytes are unsafe. With the initialization, all possible uninitialized leaks cannot disclose any meaningful entropy from the kernel space, and thus can be prevented.

5. IMPLEMENTATION

In this section, we present the implementation details that readers may be interested in. UniSan is built on the LLVM compiler infrastructure with version `3.7.1`. The unsafe allocation detector is implemented as two analysis module passes. One is for building the complete call graph iteratively and maintaining all necessary global context information (e.g., defined functions); the other performs the initialization and reachability analyses based on the built call graph. The unsafe allocation initializer is implemented as a transformation function pass, invoked after the detection pass. Both passes are inserted after all optimization passes. To compile the kernel into LLVM IR, we leverage the LLVMLinux project [23]. Because `llvm-link` has a symbol renaming problem, instead of merging all bitcode files into a single module, we adopt the iterative algorithm from KINT [43] to process individual bitcode files. So the input of the analysis phase is just a list of bitcode files. UniSan is easy to use:

```
$ CC=unisan-cc make
$ unisan @bitcode.list
```

5.1 Bookkeeping of the Analysis

For each tracked allocation, we use a dedicated data structure to record its tracking results along the propagation paths. This data structure mainly includes the initialization and sinking information of each byte. The tracking history is also included to avoid repeatedly tracking a user—we do not need to track a user multiple times if the status of the tracked value is not changed.

As mentioned in §4.1.4, UniSan maintains the user-graph for the tracked value. The users in the graph may access different elements of the tracked value. To know which part of the tracked value is being initialized or sunk during the tracking, element information is also kept in each node of the user-graph.

Moreover, we maintain *reference hierarchy* for pointers. The *reference hierarchy* is to understand if a pointer is directly or indirectly (recursively) pointing to the tracked allocation. To better understand it, let us see this example: `store i8* %A, i8** %B` stores value `A` to the memory pointed to by `B` but not `B` itself. To differentiate whether the storing operation is targeting the tracked allocation or its reference, the referencing relationship between them is required, and thus we maintain the *reference hierarchy*. Our *reference hierarchy* is straightforward: the "indirectness" is decreased by one by LoadInst; but increased by one by StoreInst. To tackle the alias problem, if StoreInst stores to non-stack memory, we assume it is sinking and stop tracking. Certainly, point-to analyses can achieve the same goal, but they are heavyweight—analyzing the kernel may take many hours.

The initialization status is updated when the tracked value is assigned another value by `StoreInst`, while the sinking status is updated when the tracked value is stored to a non-stack value in `StoreInst` or it is passed to sink functions in `CallInst`.

5.2 Tracking Different Users

Analyses are carried out by traversing the user-graph. UniSan handles different kinds of users accordingly. After handling a user, the next users of the current user will be further tracked. In this section, we detail how the handling of each type of user is implemented.

LoadInst. `LoadInst` loads the data *pointed to* by the tracked value to the target value. We independently track both values and merge (i.e., the union operation) the unsafe bytes when both tracking processes return. Since `LoadInst` is essentially dereferencing the tracked value that is a pointer, we also increase the reference hierarchy by one in the tracking of the target value.

StoreInst. `StoreInst` stores the tracked value to the memory *pointed to* by the target value. Let us see `store i8* %A, i8** %B` again. When the tracked value is `A` (i.e., it is the value operand), we first use the conservative basic alias analysis (§4.1.6) to find the aliases of `B`. If not all aliases can be found or some aliases are from non-stack memory, we assume `A` is sunk and update its sinking status; otherwise, we further track aliases independently and merge all unsafe bytes. Since the target value is a reference of the stored value, the reference hierarchy is decreased by one in the tracking of aliases. When the tracked value is `B`, we first consult the reference hierarchy (§5.1) to see if `B` is the tracked allocation (but not its reference); if yes, we record that the corresponding bytes of the tracked allocation are initialized.

CallInst. When the tracked value is passed to callees via argument, we recursively track the arguments in callees independently and merge all unsafe bytes. Inline assembly is conservatively handled as shown in §4.1.6. If the called function is a sink, we record that the corresponding bytes are sunk.

GEPOperator. `GEPOperator` statements get the pointer of an element of the tracked value, which essentially creates an alias of the tracked value. The offset of the target element into the base of the tracked value and its size are calculated and maintained in the user-graph. The users of the target element are merged into the user-graph of the tracked value. The element information is the key to implement the byte-level analyses; however, when the indices of `GEPOperator` are not constants, we will not be able to obtain the element information. In this case, we stop the initialization analysis and only continue the reachability analysis, since we cannot statically decide which bytes will be initialized.

ReturnInst. `ReturnInst` is an instruction that returns a value from a function. We first use the global call-graph to find all `CallInsts` that call the current function containing the `ReturnInst`. Then these `CallInsts` are independently tracked, and unsafe bytes are merged.

CastInst, SelectInst, and PHINode. The definitions of these statements can be found at [1]. These cases are generating alias of the tracked values. We find their users and merge the users to the user-graph of the tracked value.

CmpInst, SwitchInst, BranchInst. The definitions of these statements can be found at [1]. We skip the handling for these cases, since they do not operate the tracked value.

Others. Any other cases are conservatively treated as sinks, so all uninitialized bytes in the tracked value are assumed to be unsafe.

5.3 Modeling Basic Functions

Similar to traditional static program analyses, we also modeled some basic functions. Specifically, we modeled string-related functions that typically use loops to process strings, to improve the efficiency of tracking. Moreover, we modeled some frequently used LLVM intrinsic functions (e.g., `llvm.memset`) that do not have function bodies in LLVM IR. In total, we modeled 62 simple functions by summarizing how they propagate and initialize the arguments.

5.4 Dynamic Allocations

Dynamic allocations create objects with a dynamic size. They are common (about 40%, according to our study) in heap. In general, we do not perform initialization analysis for dynamic allocations, because we cannot determine which bytes are being accessed at compiling time. As a result, we conservatively consider the whole allocation as uninitialized and only perform reachability analysis. With one exception, during the initialization analysis we will record the size value (in LLVM IR) of the dynamic allocation; if the allocation is later initialized by a basic function (e.g., `memset`) using the same size value as recorded, we consider that it is fully initialized and safe. In the instrumentation module, extra instructions are inserted to compute the size at runtime for dynamic allocations on stack, which is then passed to `memset` to initialize the memory. For dynamic allocation on heap, we utilize the `__GFP_ZERO` flag for initialization.

6. EVALUATION

We systematically evaluated UniSan to answer the following questions:

- The **accuracy** of UniSan, i.e., to what extent can UniSan filter out safe allocations?
- The **effectiveness** of UniSan, i.e., whether it can prevent known and detect previously unknown uninitialized data leaks?
- The **efficiency** of UniSan, i.e., what is the performance overhead of the secured kernel?

Experimental setup. We applied UniSan to both x86_64 and AArch64 kernels. For x86_64, we used the latest mainline Linux (with version `4.6.0-Blurry Fish Butt`) with patches from the LLVMLinux projects [23]. x86_64 kernels were tested in a desktop machine equipped with an Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz processor and 32GB RAM. The OS is the 64-bits Ubuntu 14.04. For AArch64, we used the latest Android kernel (with version `tegra-flounder-3.10-n-preview-2`) from the Android Open Source Project, with patches from [38]. The Android kernels were tested in the Nexus 9 device that has a duo-core ARMv8 processor and 2GB RAM. We used the default configurations for both kernels.

6.1 Accuracy of Unsafe Allocation Detector

We first conducted the statistical analysis on the accuracy of the unsafe allocation detector—how many allocations are reported as unsafe. The results are shown in Table 1. In particular, there are around 2k modules (i.e., bitcode files) enabled by the default kernel configuration. It is worth noting that LLVM's optimizations aggressively inline functions and significantly opt-out most allocations—there were 156,065 functions and 413,546 static stack allocations before the optimizations. UniSan is accurate in detecting unsafe stack allocations: only 8.4% and 9.5% stack allocations are detected as unsafe for x86_64 and AArch64, respectively. Since dynamic allocation is common for a heap object, UniSan's detection rate is higher for heap allocations—13.2% for x86_64 and 14.9% for AArch64. UniSan performs byte-level detection, so we also report the total number of statically allocated bytes and the detected unsafe bytes. These statistic results show that UniSan can filter out most safe allocations to avoid unnecessary initializations.

To understand how initialization analysis and reachability analysis individually help in filtering out safe allocations, we further counted the number of unsafe allocations when we disable one of them. Specifically, in x86_64, if we disable the initialization analysis, there are 3,380 unsafe stack allocations. If we disable the reachability analysis (i.e., assuming all function calls as sinks), there are 14,094 unsafe stack allocations. In AArch64, the numbers are 2,961 and 11,209, respectively.

6.2 Effectiveness of Preventing Leaks

Preventing known leaks. Conservative policies (§4.1.6) have been enforced to eliminate potential false negatives of UniSan. To confirm that UniSan does not miss uninitialized data leaks in practice, we selected 43 recent kernel uninitialized data leaks reported after 2013. All these leaks have been assigned with CVE identifiers.

Arch	Module	Function	Static Alloca	Dyn. Alloca	Static Malloc	Dyn. Malloc	Unsafe Alloca	Unsafe Malloc	Static Bytes	Unsafe Bytes
x86_64	2,152	92,954	17,854	24	1,768	1,161	1,493	386	3,588,095	863,455
AArch64	2,030	93,067	15,596	32	1,790	1,233	1,485	451	11,525,808	3,351,181

Table 1: Detection accuracy of UniSan, measured by how many allocations are detected as unsafe. `Alloca` represents stack allocations, while `Malloc` represents different heap allocations. Please note that only the code (about 10% of the whole kernel code) enabled by the default kernel configuration is included.

CVE	Mem.	Sink	Leak Bytes	Cause	UniSan
CVE-2015-5697	heap	user	<4096	M	✓
CVE-2015-7884	stack	user	4	M	✓
CVE-2015-7885	stack	user	< 28	M	✓
CVE-2014-1444	stack	user	2	P	✓
CVE-2014-1445	stack	user	2	P	✓
CVE-2014-1446	stack	user	4	M	✓
CVE-2014-1690	stack	sock	< 16	M	✓
CVE-2014-1739	stack	user	192	M	✓
CVE-2013-4515	stack	user	10	M	✓
CVE-2013-3235	stack	user	12	M	✓
CVE-2013-3234	stack	user	12	P+M	✓
CVE-2013-3233	stack	user	12	M	✓
CVE-2013-3232	stack	user	6	P+M	✓
CVE-2013-3230	stack	user	4	M	✓
CVE-2013-3223	stack	user	1	P	✓
CVE-2013-2636	stack	sock	< 20	M	✓
CVE-2013-2636	stack	sock	4	P+M	✓
CVE-2013-2636	stack	sock	3	P	✓
CVE-2013-2636	stack	sock	18	M	✓
CVE-2013-2635	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	34	M	✓
CVE-2013-2634	stack	sock	8	M	✓
CVE-2013-2634	stack	sock	20	P+M	✓
CVE-2013-2634	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	18	M	✓
CVE-2013-2547	stack	sock	< 64	M	✓
CVE-2013-2547	stack	sock	8	M	✓
CVE-2013-2237	heap	sock	1	M	✓
CVE-2013-2234	heap	sock	2	M	✓
CVE-2013-2148	stack	user	1	M	✓
CVE-2013-2141	stack	user	4	M	✓
CVE-2012-6549	stack	user	2	M	✓
CVE-2012-6548	stack	user	2	M	✓
CVE-2012-6547	stack	sock	4	M	✓
CVE-2012-6546	stack	user	2	P	✓
CVE-2012-6545	stack	sock	1	P	✓
CVE-2012-6544	stack	sock	2	M	✓
CVE-2012-6543	stack	sock	2	M	✓
CVE-2012-6541	stack	user	4	P	✓
CVE-2012-6540	stack	user	12	M	✓
CVE-2012-6539	stack	user	4	P	✓
CVE-2012-6537	stack	user	4	P	✓

Table 2: Tested known uninitialized data leaks. UniSan can successfully prevent all of them. In the cause column: P-compiler padding; M-missing element initialization.

Please note that some leaks are not included because the corresponding code is not enabled by the default kernel configuration or a very similar leak has already been included. The patches of these vulnerabilities are temporarily reverted for testing. The results of the experiment are shown in Table 2. UniSan successfully detected and prevented all these leaks without any *false negative*; hence the effectiveness in preventing existing leaks is confirmed.

Detecting previously unknown vulnerabilities. UniSan is designed as a prevention tool that can automatically detect and fix all uninitialized data leaks at the LLVM IR level; no manual effort is required. However, to confirm that UniSan can truly prevent new leaks and estimate the false positives due to our conservative policies, we manually review the unsafe allocations reported by the unsafe allocation detector. As shown in Table 1, UniSan detected about 1,500 possible unsafe stack allocations and 300 possible unsafe heap allocations. Due to limited time and labor, we randomly chose 300 detected unsafe stack allocations and 50 unsafe heap allocations. In summary, we have verified `19` new uninitialized data leaks in the latest Linux kernel and Android kernel. All of these vulnerabilities are from stack, more details can be found in Table 3. All of them have been confirmed by the Linux kernel team and Android security team. Since UniSan has not been adopted by them yet, we have to manually write the corresponding patches in source code.

6.3 Efficiency of the Secured Kernels

UniSan carries out flow-sensitive, context-sensitive, and field-sensitive analyses to accurately detect the unsafe allocations, so that the performance overhead is controlled by minimizing the number of initializations. To quantify this benefit, we conducted a series of extensive performance evaluations. In particular, we first used the LMBench [24] micro benchmark to evaluate the performance on core system operations (e.g., syscalls latency). We then used Android Benchmarks and the SPEC Benchmarks as the macro benchmarks to evaluate the performance impacts on user space programs for the protected Android kernel and Linux kernel, respectively. To measure the performance impacts on I/O intensive server programs, we further used ApacheBench to test the performance of Apache web server. All these evaluations consist of three groups: 1) native mode, in which UniSan is not applied; 2) blind mode, in which all stack allocations and heap allocations without the `__GFP_ZERO` flag are initialized without checking whether or not they are safe; and 3) UniSan mode, in which UniSan is applied. In the three groups of evaluations, the kernel was replaced with the corresponding one. Note that, we did not further break down the performance overhead introduced by stack and heap, because the overall overhead is already negligible.

6.3.1 System Operations

In order to measure how UniSan affects the performance of core operating system services, we used LMBench [24] as the micro benchmark. Specifically, we focus on the latency of syscalls (e.g., `null`, `write`, `open/close`, `sigaction`, etc.) and impact on bandwidth (e.g., `pipe`). We ran each experiment 10 times, and the results are shown in Table 4. In the blind mode, the performance overhead could be up to 22% (the signal handle case). There are also some cases (e.g., select and pipe) where its overhead is more than 10%. Such a performance overhead is significant for the OS kernel—the foundation of computer systems. In contrast, UniSan has a much lower performance overhead than the blind mode; its maximum overhead is 7.1% in the protection fault case. We also notice that UniSan's performance overhead is negligible (< 1%) in many cases. On average, the performance overhead of UniSan is less than 1.5% for both the Linux and Android kernels.

As a comparison, we further measured STACKLEAK's performance overhead in the same settings as UniSan except using GCC to compile it. As shown in Table 4, STACKLEAK imposes an average of more than 40% performance overhead in system operations, which is much higher than UniSan.

#	Sub-System	Module	Object	Mem.	Sink	Leak Bytes	Cause	Kernel	Patch	CVE
1	net/core	rtnetlink.c	map	stack	sock	4	P	A&L	✓	CVE-2016-4486
2	usb	devio.c	ci	stack	user	3	P	A&L	✓	CVE-2016-4482
3	net/wireless	nl80211.c	mac_addr	stack	sock	6	M	A&L	✓	AndroidID-28620350
4	nec/llc	af_llc.c	info	stack	user	1	P	A&L	✓	CVE-2016-4485
5	sound	timer.c	tread	stack	user	8	P	A&L	✓	CVE-2016-4569
6	sound	timer.c	r1*	stack	user	8	P	A&L	✓	CVE-2016-4578
7	sound	timer.c	r1*	stack	user	8	P	A&L	✓	CVE-2016-4578
8	net/x25	af_x25.c	dte_facilities	stack	user	8	M	A&L	✓	CVE-2016-4569
9	net/tipc	netlink_compat.c	link_info	stack	sock	<60	M	A&L	✓	CVE-2016-5243
10	net/rds	recv.c	minfo	stack	sock	1	M	A&L	✓	CVE-2016-5244
11	net/mac80211	mlme.c	deauth_buf	stack	sock	26	M	A	S	AndroidID-28620568
12	net/wireless	wl_cfg80211.c	sinfo	stack	sock	<116	M&P	A	S	AndroidID-28619338
13	net/wireless	util.c	hdr	stack	sock	1	M	A	S	AndroidID-28620324
14	net/netfilter	...queue_core.c	phw	stack	sock	2	M	A	S	AndroidID-28673002
15	net/netfilter	nfnetlink_log.c	phw	stack	sock	2	M	A	S	AndroidID-28672819
16	net/netfilter	nfnetlink_log.c	pmsg	stack	sock	1	M	A	S	AndroidID-28616963
17	media	media-device.c	u_ent	stack	user	<192	M&P	A	S	AndroidID-28616963
18	media	media-device.c	pad	stack	user	10	M&P	A	S	AndroidID-28616963
19	media	media-device.c	link	stack	user	28	M&P	A	S	AndroidID-28616963

Table 3: List of new kernel uninitialized data leak vulnerabilities discovered by UniSan. In `Patch` column, `S` represents it has been submitted, and ✓represents it has been applied. In the `Cause` column, `P` is compiler padding and `M` is missing element initialization. In the `Kernel` column, `A` represents Android and `L` represents Linux. *These two "`r1`" are in different functions. AndroidID is internally maintained by Android security team at Google.

	Linux (x86_64)					**Android (AArch64)**					**STACKLEAK (x86_64)**		
	Native	**Blind**	(%)	**UniSan**	(%)	**Native**	**Blind**	(%)	**UniSan**	(%)	**W/O**	**With**	(%)
null syscall	0.04	0.04	(0.0%)	0.04	(0.0%)	0.42	0.42	(0.0%)	0.42	(0.0%)	0.04	0.12	(200.0%)
stat	0.37	0.40	(8.1%)	0.38	(2.7%)	1.33	1.43	(7.5%)	1.37	(3.0%)	0.45	0.70	(55.6%)
open/close	1.20	1.22	(1.7%)	1.15	(-4.2%)	6.09	6.27	(3.0%)	6.07	(-0.3%)	1.17	1.74	(48.7%)
select TCP	2.44	2.48	(1.6%)	2.44	(0.0%)	9.67	10.67	(10.3%)	9.80	(1.3%)	2.13	2.37	(11.3%)
signal install	0.11	0.11	(0.0%)	0.11	(0.0%)	0.58	0.58	(0.0%)	0.57	(-1.7%)	0.11	0.24	(118.2%)
signal handle	0.58	0.71	(22.4%)	0.60	(3.4%)	1.79	1.98	(10.6%)	1.85	(3.4%)	0.63	0.71	(12.7%)
fork+exit	156	157	(0.6%)	156	(0.0%)	851	892	(4.8%)	861	(1.2%)	156	167	(7.1%)
fork+exec	452	464	(2.7%)	455	(0.7%)	2687	2737	(1.9%)	2742	(2.0%)	455	467	(2.6%)
prot fault	0.295	0.317	(7.5%)	0.316	(7.1%)	1.37	1.47	(7.3%)	1.35	(-1.5%)	0.315	0.462	(46.7%)
pipe(latency)	9.673	10.21	(5.6%)	9.909	(2.4%)	8.850	8.898	(0.5%)	8.882	(0.4%)	9.141	10.2	(11.6%)
TCP(latency)	91.8	99.5	(8.4%)	97.3	(6.0%)	–	–	–	–	–	92.3	99.8	(8.1%)
pipe(bandw)	3,321	3,250	(2.1%)	3,315	(0.2%)	838	728	(13.1%)	804	(4.1%)	987	960	(2.7%)
TCP(bandw)	2,331	2,264	(2.9%)	2,333	(-0.1%)	–	–	–	–	–	1514	1183	(21.9%)

Table 4: LMBench results. Time is in microseconds. The last two rows are measuring bandwidth, which is the bigger the better. We do not have numbers for TCP case in Android kernel, because we were not able to establish the TCP socket to our Nexus 9. STACKLEAK is compiled with GCC.

6.3.2 User Space Programs

For x86_64, we used the standard SPEC CPU 2006 benchmark suite to test the performance impacts of UniSan on user space programs. The benchmark programs were compiled with the common options (`-pie -fPIC -O2`) and each benchmark was run 10 times. The results are shown in Table 5. Specifically, the performance overhead introduced by UniSan is only 0.54%, which is negligible. The performance overhead of the blind mode is higher than UniSan, which is 1.92%. Note that the moderate blind-mode overhead over SPEC benchmarks is when the user-space is not protected; when we also applied the blind mode to the user-space code, the overhead becomes 10% (7% from stack and 3% from heap).

For AArch64, we instead used the dedicated benchmarks for Android: AnTuTu (including 3DBench) and Vellamo. They are typical Android benchmarks to test a variety of cases (e.g., video streaming, web browsing, photo editing, etc.). Each testing was repeated five times for deriving the average numbers. The evaluation results are shown in Table 6. Again, UniSan's performance overhead is negligible (<1%) and is lower than blind mode.

6.3.3 Server Programs

We used ApacheBench to test the performance impacts of UniSan

Programs	**Native**	**Blind**	(%)	**UniSan**	(%)
perlbench	3.61	3.61	(0.0%)	3.59	(-0.6%)
bzip2	4.74	4.78	(0.8%)	4.78	(0.8%)
gcc	0.968	0.970	(0.2%)	0.966	(-0.2%)
mcf	2.73	2.76	(1.1%)	2.74	(0.4%)
gobmk	14.0	14.0	(0.0%)	14.0	(0.0%)
hmmer	2.07	2.03	(-1.9%)	2.04	(-1.4%)
sjeng	3.27	3.30	(0.9%)	3.33	(1.8%)
libquantum	0.0387	0.0397	(2.6%)	0.0395	(2.1%)
h264ref	9.26	9.32	(0.6%)	9.28	(0.2%)
omnetpp	0.362	0.364	(0.6%)	0.362	(0.0%)
astar	7.81	7.92	(1.4%)	7.92	(1.4%)
xalancbmk	0.0758	0.0779	(2.8%)	0.0738	(-2.6%)
milc	4.57	4.76	(4.2%)	4.76	(4.2%)
namd	8.85	8.86	(0.1%)	8.83	(-0.2%)
dealII	10.5	10.6	(1.0%)	10.6	(1.0%)
soplex	0.0166	0.0186	(12.0%)	0.0170	(2.4%)
povray	0.427	0.439	(2.8%)	0.426	(-0.2%)
lbm	1.68	1.69	(0.6%)	1.69	(0.6%)
sphinx	1.19	1.28	(7.6%)	1.20	(0.8%)
Geo-mean	(seconds)		1.92%		**0.54%**

Table 5: User space (x86_64) performance evaluation results with the SPEC benchmarks. Time is in second, the smaller the better.

Benchmark	Native	Blind	(%)	UniSan	(%)
AnTuTu	94,998	92,900	(2.2%)	94,735	(0.3%)
Vellamo-Browser	4,776	4,711	(1.4%)	4,729	(1.0%)
Vellamo-Metal	2,766	2,777	(-0.4%)	2,776	(-0.4%)
Vellomo-Multicore	2,610	2,499	(4.3%)	2,600	(0.4%)

Table 6: User space (AArch64) performance evaluation results with multiple android benchmarks. The scores are the higher the better.

on the latency and bandwidth of the server program—Apache web server. The ApacheBench ran in the laptop connected to our desktop machine with an one Gigabit cable. We used Apache with its default configurations. We first used a 1KB file to evaluate the latency when the concurrency (i.e., simultaneous connections) was set to be 1, 50, 100, 150, 200, and 250. Then we used files with different sizes (100B, 1KB, 10KB, 100KB, and 1MB) to measure the bandwidth (i.e., throughput) of the web server. In the bandwidth evaluation, the concurrency was set to be 32; the network I/O was not saturated except when the file size was 1MB. The request for each experiment was repeated 10,000 times. The evaluation results show that the blind mode incurs an average slowdown of 0.7% in the concurrency experiment and 0.9% in the bandwidth experiment. In contrast, UniSan imposes an unobservable slowdown (<0.1%) in both experiments. These results confirm that UniSan incurs almost no overhead to the server programs that are I/O intensive.

6.4 Miscellaneous

Scalability. UniSan is scalable, which can analyze complex programs like the OS kernel. To better understand its scalability, we also measure the time for it to protect the kernels. In our physical machine, UniSan finished the protection of kernels within 117 seconds for x86_64 and 170 seconds for AArch64.

Binary size. UniSan inserts small initializers (e.g., zero-initialization) for unsafe allocations. To confirm that it does not significantly increase the binary size, we measured the size of the installed boot images. The results showed that UniSan increases the size of Linux boot image from 7,417KB to 7,445KB (0.38%) and the one of Android from 6.348KB to 6.366KB (0.27%). Both are negligible.

7. RELATED WORK

Kernel leak detection and prevention. The most related works are Peiró's model checking based kernel stack leak detection [32], PaX STACKLEAK plugin [40], and Split kernel [20].

Peiró's model checking is essentially a simple taint tracking from stack allocation to `copy_to_user`. If there is no assignment or `memset` between the allocation and `copy_to_user`, a leak is reported. Such an approach has some limitations: it does not track targets of indirect calls, different sinks, the propagation of uninitialized data, or handle partial initialization, so many leak cases will be missed. In UniSan, we accurately track each byte of an allocation and eliminate false negatives thoroughly.

`STACKLEAK` simply clears the used kernel stack when the control is transferred back to the user space. Such an approach cannot prevent leaks that disclose data generated in the current syscall. Split kernel [20] instead clears the stack frame whenever a function is called. Both approaches impose significant performance overhead (see Table 4) because they need to frequently zero-out blocks of memory. Compared with UniSan, these approaches provide a broader security, as they also prevent other uninitialized data uses (e.g., uninitialized pointer dereferencing).

None of above approaches can protect heap allocations. To the best of our knowledge, UniSan is the first approach that prevents both stack and heap uninitialized data leaks efficiently.

Detecting uninitialized memory accesses. Traditional uninitialized data read detections and undefined behavior [42] detections can also detect uninitialized data leaks.

Both LLVM and gcc provide the `-Wuninitialized` option to detect uninitialized data use. First, they only perform intra-procedure analysis; however, all the uninitialized data leaks propagate data across function boundary (e.g., calling `copy_to_user`); hence they cannot detect such leaks. Second, their analysis does not handle many common cases. For example, reading the uninitialized data through its pointer cannot be caught.

Some dynamic tracking techniques [5, 31, 36] have been proposed to detect uninitialized data reads. These techniques rely on dynamic instrumentation platforms (e.g., Valgrind [29] and DynamoRIO [5]) to instrument the binary and analyze memory access patterns dynamically. These tools can report uninitialized data use without false positives; however, their performance overhead (>10x) is too high for them to be adopted as prevention tools.

Differently, MemorySanitizer [39] relies on compile time instrumentation and shadow memory to detect uninitialized data use at run-time. Its performance is much better than dynamic instrumentation based tools; however, it still imposes a 3-4x performance overhead. Usher [44] proposes value-flow analysis to reduce the number of tracked allocations to reduce the performance overhead of MemorySanitizer to 2-2.5x.

Memory safety and model checking. Many memory safety techniques [26–28, 35] have been proposed to prevent spacial memory errors (e.g., out-of-bound read) and use-after-free bugs, and model checking techniques [3, 25] can detect semantic errors caused by developers. These tools are effective to detect or prevent kernel leaks caused by spacial memory errors, use-after-free, or semantic errors, and thus are orthogonal to UniSan. DieHard [4] probabilistically detects uninitialized memory uses for heap but not stack. Cling [2] constrains memory allocation to allow address space reuse only among objects of the same type, which can mitigate exploits of temporal memory errors (e.g., use-after-free and uninitialized uses). StackArmor [7] is a sophisticated stack protection that also prevents uninitialized reads in binaries. Since StackArmor uses stack layout randomization to prevent inter-procedural uninitialized reads, such a protection is probabilistic. Type systems [12, 15] can prevent dangling pointer and uninitialized pointer dereferences, and out-of-bound accesses; however uninitialized data leaks are not covered.

Protections using zero-initialization. Zero-initialization has been leveraged to achieve protections in previous works. Secure deallocation [8] zero-initializes the deallocated memory to reduce the lifetime of data, thus reducing the risk of data exposure. Lacuna [13] allows users to run programs in "private sessions". After a session is over, all memory of its execution is erased. In general, zeroing-upon-deallocation has two issues: 1) deallocations are not always available (e.g., deallocation is missing in the case of memory leak); 2) it is hard to selectively zero deallocations for efficiency, as discussed in §3.2. StackArmor [7] only zero-initializes intra-procedural allocations (i.e., not be passed to other functions) that cannot be proven to be secure against uninitialized reads.

8. DISCUSSION AND FUTURE WORK

Custom heap allocator. In the current implementation, UniSan only tracks typical heap allocators (`kmalloc` and `kmen_cache_alloc`). To handle custom heap allocations (e.g., `alloc_skb`), UniSan can use the specifications of allocators provided by developers. In fact, for user space programs, LLVM already provides an API `isMallocLikeFn` to test whether a value is a call to a function that allocates uninitialized memory based on heuristics. We plan to also use heuristics to infer "malloc-like" functions in the kernel.

Source code requirement. Source code is required. In case that some kernel drivers are close-sourced, we have to carefully identify all possible calls that target these drivers and assume all such calls as sinks. Failures in identifying such calls will result in incompleteness of call-graph and thus false negatives.

Security impacts of zero-initialization. Some systems use uninitialized memory as the source of randomness. For example, the `SSLeay` implementation of `OpenSSL` uses uninitialized buffer as entropy source to generate random number (see `ssleay_rand_bytes()`). Zero-initialization will clearly reduce the entropy of such a source of randomness; however, we argue that using such a source of randomness is insecure and should be avoided—reading uninitialized data is classified as memory error.

False positives. In many cases, UniSan eliminates false negatives by sacrificing accuracy (i.e., increasing the false positive rate). There is still room to reduce the false positives. For example, point-to analysis [16] can help find indirect call targets, and dynamic taint analysis [34] can help handle cases like inline assembly.

Considering that false positives do not affect program semantics but just introduce more performance overhead and that UniSan is already efficient, we leave these optimizations for future work.

More kernel modules. In our experiments, we included only modules enabled by the default kernel configurations. Since the current Linux kernel has around 20,000 modules total, a majority of them have not been included in our evaluation. Unfortunately, due to some GCC-specific features, some modules are still not compilable by LLVM and require extra engineering effort to patch. Since UniSan is a proof-of-concept research project, we believe supporting these additional modules is out-of-scope. We may be able to rely on the open source community to provide the required patches (e.g., the LLVMLinux project [23]) or port UniSan to GCC.

Beyond kernels. UniSan's detection and instrumentation work on the LLVM IR level, and thus it can be naturally extended to protect user space programs. Specifically, to support user space program, IR of libraries should also be included, and sources (for heap) and sinks should be re-defined. As a future work, we will use UniSan to detect and prevent information leaks in security- and privacy-sensitive programs (e.g., OpenSSL).

9. CONCLUSION

Information leaks in kernel pose a major security threat because they render security protection mechanisms (e.g., kASLR and StackGuard) ineffective and leak security-sensitive data (e.g., cryptographic keys and file caches). In particular, uninitialized data read is the most critical vulnerability because it is the cause of most information leaks in kernel. Furthermore, none of the existing defenses can completely and efficiently prevent uninitialized data leaks.

The key idea behind UniSan is to use byte-level, flow-sensitive, and context-sensitive reachability analysis and initialization analysis to identify any allocation that leaves kernel space without having been fully initialized, and to automatically instrument the kernel to initialize this allocation. UniSan has no false negatives. That is, it prevents all possible uninitialized data leaks in kernel. We have applied UniSan to the latest Linux kernel and Android kernel and found that UniSan can successfully prevent 43 known uninitialized data leaks, as well as many new ones. In particular, 19 of the new data leak vulnerabilities in the latest kernels have been confirmed by the Linux community and Google. Extensive evaluation has also shown that UniSan is robust and imposes only a negligible performance overhead.

10. ACKNOWLEDGMENT

We thank Byoungyoung Lee, Herbert Bos, Anil Kurmus and the anonymous reviewers for their valuable feedback, as well as our operations staff for their proofreading efforts. This research was supported in part by the NSF award CNS-1017265, CNS-0831300, CNS-1149051, CNS-1563848 and DGE-1500084, by the ONR under grant N000140911042 and N000141512162, by the DHS under contract N66001-12-C-0133, by the United States Air Force under contract FA8650-10-C-7025, by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP-006, by the ETRI MSIP/IITP[B0101-15-0644]. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ONR, DHS, United States Air Force, DARPA or MSIP.

References

[1] LLVM Classes Definition, 2016. http://llvm.org/docs/doxygen/html/annotated.html.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, 2006.

[4] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[5] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, Mar. 2011.

[6] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, Shanghai, China, July 2011.

[7] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[8] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005.

[9] K. Cook. Kernel address space layout randomization, 2013. http://outflux.net/slides/2013/lss/kaslr.pdf.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, Jan. 1998.

[11] C. Details. Vulnerabilities By Type, 2016. http://www.cvedetails.com/vulnerabilities-by-types.php.

[12] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[13] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

[16] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[17] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. Ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[18] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[19] M. Krause. CVE-2013-1825: various info leaks in Linux kernel, 2013. http://www.openwall.com/lists/oss-security/2013/03/07/2.

[20] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2014.

[21] LLVM. LLVM Alias Analysis Infrastructure, 2016. http://llvm.org/docs/AliasAnalysis.html.

[22] LLVM. The LLVM Compiler Infrastructure, 2016. http://llvm.org/.

[23] LLVMLinux. The LLVMLinux Project, 2016. http://llvm.linuxfoundation.org/index.php/Main_Page.

[24] L. W. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[25] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[26] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, Orlando, FL, Feb. 2014.

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[28] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[29] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[30] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[31] V. Nossum. Getting Started With kmemcheck, 2015. https://www.kernel.org/doc/Documentation/kmemcheck.txt.

[32] S. Peiró, M. M. noz, M. Masmano, and A. Crespo. Detecting stack based kernel information leaks. In *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, 2014.

[33] J. Rentzsch. Data alignment: Straighten up and fly right – Align your data for speed and correctness, 2005. https://www.ibm.com/developerworks/library/pa-dalign/pa-dalign-pdf.pdf.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[36] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, June–July 2005.

[37] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007.

[38] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[39] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, Feb. 2015.

[40] P. Team. PaX - gcc plugins galore, 2013. https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf.

[41] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium*, 2014.

[42] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, July 2012.

[43] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[44] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, Orlando, FL, Feb. 2014.