Received March 2018; | Revised 17 July 2018; | Accepted 9 August 2018

DOI: xxx/xxxx

RESEARCH ARTICLE

Towards Linux Kernel Memory Safety

Elena Reshetova[1] | Hans Liljestrand[2] | Andrew Paverd[2] | N. Asokan[2]

[1]Intel OTC, Espoo, Finland
[2]Aalto University, Espoo, Finland

Correspondence
Elena Reshetova,
Email: elena.reshetova@intel.com
Hans Liljestrand,
Email: hans.liljestrand@aalto.fi
Andrew Paverd,
Email: andrew.paverd@ieee.org
N. Asokan,
Email: asokan@acm.org

Summary

The security of billions of devices worldwide depends on the security and robustness of the mainline Linux kernel. However, the increasing number of kernel-specific vulnerabilities, especially memory safety vulnerabilities, shows that the kernel is a popular and practically exploitable target. Two major causes of memory safety vulnerabilities are reference counter overflows (temporal memory errors) and lack of pointer bounds checking (spatial memory errors).

To succeed in practice, security mechanisms for critical systems like the Linux kernel must also consider performance and deployability as critical design objectives. We present and systematically analyze two such mechanisms for improving memory safety in the Linux kernel: (a) an overflow-resistant reference counter data structure designed to securely accommodate typical reference counter usage in kernel source code, and (b) runtime pointer bounds checking using Intel MPX in the kernel.

We have implemented both mechanisms and we analyze their security, performance, and deployability. We also reflect on our experience of engaging with Linux kernel developers and successfully integrating the new reference counter data structure into the mainline Linux kernel.

KEYWORDS:
Linux kernel, memory safety, Linux kernel development process

1 | INTRODUCTION

The Linux kernel forms the foundation of billions of different devices, ranging from servers and desktops to smartphones and embedded devices. There are many solutions for strengthening Linux application security, including access control frameworks (SELinux[1], AppArmor[2]), integrity protection systems (IMA/EVM[3], dm-verity[4]), encryption, key management, and auditing. However, these are all rendered ineffective if an attacker gains control of the kernel. Recent trends in Common Vulnerabilities and Exposures (CVEs) indicate a renewed interest in kernel vulnerabilities[5]. On average, it takes five years for a kernel bug to be found and fixed[6], and even when fixed, security updates might not be deployed to all vulnerable devices. Therefore we cannot rely solely on retroactive bug fixes, but require *proactive* measures to harden the kernel against potential vulnerabilities. This is the goal of the Kernel Self Protection Project (KSPP)[7], a large community of volunteers working on the mainline Linux kernel. In this paper, we describe our contributions, as part of the KSPP, to the development of two kernel memory safety mechanisms.

Depending on their severity, memory errors can allow an attacker to read, write, or execute memory, thus making them attractive targets. For example, *use-after-free* errors and *buffer overflows* feature prominently in recent Linux kernel CVEs[8,9]. Memory errors arise due to the lack of inherent memory safety in C, the main implementation language of the Linux kernel, and can be divided into two fundamental classes:

Temporal memory errors occur when pointers to freed/uninitialized memory are dereferenced. For example, a *use-after-free* error occurs when dereferencing a pointer that has been prematurely freed by another execution thread. The Linux kernel is vulnerable to temporal memory errors because it is written in C, and thus does not benefit from automated garbage collection. Instead, kernel object lifetimes are managed using *reference counters*[10]. Whenever a new reference to an object is taken, the object's reference counter is incremented, and whenever the object is released the counter is decremented. When the counter reaches zero, the object can be safely destroyed and its memory freed. Reference counters in the Linux kernel are typically implemented using the `atomic_t` type[11], which is in turn implemented as an `int` with a general purpose atomic API consisting of over 100 functions. This can give rise to temporal memory errors since `atomic_t` can overflow, as was the case in e.g, CVE-2014-2851, CVE-2016-4558, and CVE-2016-0728.

Spatial memory errors occur when pointers are used to access memory outside the bounds of their intended areas. For example, a buffer overflow occurs when the amount of data written exceeds the size of the target buffer. The Linux kernel is vulnerable to spatial memory errors as any other piece of C code due to the bugs introduced by its developers. Spatial memory errors in the mainline Linux kernel are pretty common and have appeared in e.g., CVE-2014-0196, CVE-2016-8440, CVE-2016-8459, and CVE-2017-7895.

Although memory safety has been scrutinized for decades (Section 3), much of the work has focused on user-space. These solutions are not readily transferable to kernel-space. For example, schemes like SoftBound[12] defend against spatial memory errors by storing pointer metadata in a disjoint data structure using a fast static addressing scheme (e.g., a large hash table). However, the range of memory addresses required for this data structure far exceeds the physical memory available on most systems. This is only possible for user-space applications because the kernel can handle page faults when a virtual address has not yet been mapped to physical memory. Since the kernel cannot handle its own page faults, this type of scheme cannot be used in kernel-space. Although there have been a small number of proposals for improving kernel memory safety (e.g., kCFI[13] and KENALI[14]), these have not considered the critical issue of deployability in the mainline Linux kernel. Other mechanisms, such as the widely used Kernel Address Sanitizer (KASAN)[15], are intended as debugging facilities, not runtime protection.

Contributions: In this paper, we present a solution for mitigating one of the major causes of temporal errors, and a solution for mitigating spatial memory errors in general. Specifically, we claim the following contributions:

- **Extended `refcount_t` API**: After performing a kernel-wide analysis of reference counters, we contributed to the design of `refcount_t`, a new reference counter data type that prevents reference counter overflows and significantly reduces the complexity of the previous reference counter design (Section 5.3).

- **Converting reference counters**: Locating reference counters in the kernel is non-trivial due to the diverse implementations and kernel-wide distribution. To achieve this, we developed a heuristic technique to identify instances of reference counters in the kernel source code (Section 5.1). Using this technique, we converted the core parts of the kernel source tree to use the new `refcount_t` API (Section 5.6). At the time of writing, 170 of our 233 `refcount_t` conversion patches have already been integrated.

- **MPXK**: We present a new spatial memory error prevention mechanism for the Linux kernel based on the recently released Intel Memory Protection Extensions (MPX) (Section 6). Applying MPX to the kernel is non-trivial due the strict performance and memory requirements. In particular, this required a complete re-design of how MPX memory is handled in the kernel, compared to the user-space MPX variant.

- **Evaluation**: We present a systematic analysis of `refcount_t` and MPXK in terms of performance, security, and usability for kernel developers (Section 7). Such analysis is particularly challenging for non-functional features where micro-benchmarks are not representative of real-world impact and target-platforms are diverse. In the interest of reproducibility, we make source code and test setups available at https://github.com/ssg-kernel-memory-safety.

- **Experience**: One of our primary considerations was to develop memory protection techniques that can be deployed in the mainline Linux kernel. As demonstrated by our conversion of kernel reference counters, our efforts have been successful. In Section 8 we reflect on our experience of working with the Linux kernel maintainers, and offer suggestions for other researchers with the same objective.

2 | BACKGROUND

2.1 | Linux kernel reference counters

Temporal memory errors typically arise in systems that do not have automated mechanisms for object destruction and memory de-allocation (i.e., garbage collection). In simple non-concurrent C programs, objects typically have well defined and predictable allocation and release patterns, which make it trivial to free them manually. However, in complex systems like the Linux kernel, objects are extensively shared and reused, in order to minimize CPU and memory use. For example, in the Linux kernel everything from filesystem nodes to group-permission data structures are shared and reused. To enable this sharing sharing and reuse of kernel objects, the Linux kernel makes extensive use of reference counters[10]. However, reference counting schemes are historically error prone; a missed increment or decrement, often in an obscure code path, could imbalance the counter and cause a use-after-free error.

As explained in Section 1, reference counters in the Linux kernel are typically implemented using the `atomic_t` type[11], which is in turn implemented as an `int`, which can thus overflow. In other words, this type of reference counter can be reset to zero using only increments, which will inevitably result in the object being prematurely freed, leading to a use-after-free error. Overflow bugs are particularly hard to detect using static code analysis or fuzzing techniques because they require many consequent iterations to trigger the overflow[16]. A recent surge of exploitable errors, such as CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2017-7487 and CVE-2017-8925, specifically target reference counters. In addition, the general purpose API also provides ample room for subtly incorrect reference counting schemes motivated by performance or implementation shortcuts.

2.2 | Intel Memory Protection Extensions (MPX)

Intel Memory Protection Extensions (MPX)[17] is a recent technology to prevent spatial memory errors. It is supported on both Pentium core and Atom core micro-architectures from Skylake and Goldmount onwards, thus targeting a significant range of end devices from desktops to mobile and embedded processors. In order to use the MPX hardware, an application must be compiled with an MPX-enabled compiler, such as GCC or ICC, so that the resulting binary includes the new MPX instructions. In GCC, this support is built around the architecture-independent *Pointer Bounds Checker*. At present, these compilers only support MPX for user-space applications. Furthermore, the application must be run on an MPX-enabled operating system (OS), which manages the MPX metadata. MPX is source and binary compatible, meaning that no source code changes are required and that MPX-instrumented binaries can be linked with non-MPX binaries (e.g., system libraries). MPX-instrumented binaries can still be run on all systems, including legacy systems without MPX hardware. If MPX hardware is available, the instrumented binaries will automatically detect and configure this hardware when the application is initialized. However, the checks are performed by compiler instrumentation and can only cover compiler generated code, not, for instance, assembly.

MPX prevents spatial memory errors by checking pointer bounds before a pointer is dereferenced. Every pointer is associated with an upper and lower bound, which are determined by means of compile-time code instrumentation. For a given pointer, the bounds can either be set statically (e.g., based on static data structure sizes), or dynamically (e.g., using instrumented memory allocators). The MPX instrumentation uses the new `bndcl` and `bndcu` instructions to explicitly check the bounds before any use of a pointer, so the type of access performed by the pointer (read, write or execute) does not matter. For example, `malloc` is instrumented to set the bounds of newly allocated pointers. When pointers are derived from others — e.g., a pointer into a sub-structure is taken — the bounds are typically *narrowed* to that of the sub-structure. The narrowed bounds then behave as other bounds, i.e., they are checked upon pointer dereference. This narrowing is omitted in specific cases that indicate non-compatible use. For instance, a pointer to the first element of an array is often used as an iterator; hence its bounds cannot be narrowed. MPX does consider pointer arithmetic and will not modify bounds based on such modifications.

In order to perform a bounds check, the pointer's bounds must be loaded in one of the four new MPX bound (`bndx`) registers. MPX stores pointer bounds separately from the pointer itself, either in a special purpose data-structure, the stack or in static memory. In some cases the bounds to be stored cannot be determined at compile time, preventing the use of static or stack memory (e.g., a dynamically-sized array of pointers). In these cases, bounds are stored in memory in a new two-level metadata structure and accessed using the new bound load (`bndldx`) and store (`bndstx`) instructions. As shown in Figure 1 , these instructions use bits 20-47 of the pointer's address are used as an index into the process's Bound Directory (BD), which contains pointers to the relevant Bound Tables (BTs). Bits 3-19 of the pointer's address are then used as an index into the specific BT that contains the address and bounds for the pointer. On 64-bit systems the BD is 2 GB and each BT is 4 MB, which means that a

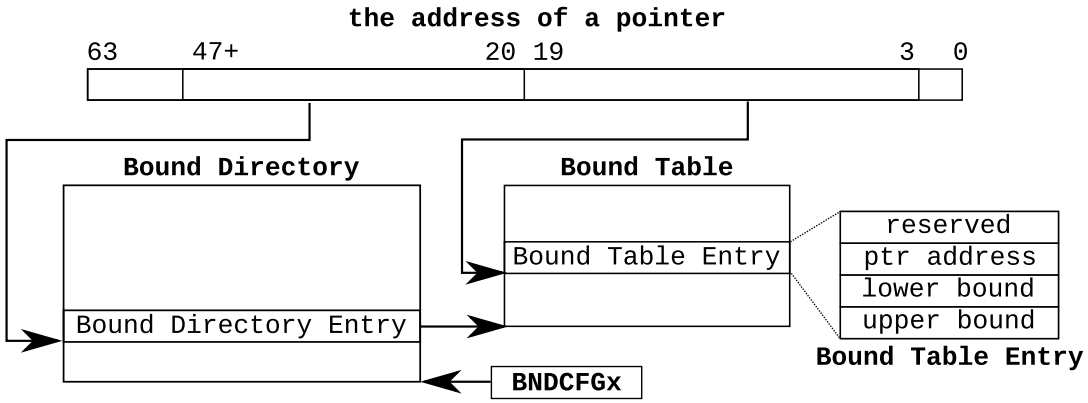


FIGURE 1 The MPX bounds addressing scheme uses the 2 GB Bound Directory to lookup 4 MB Bound Tables, which in turn contain the bounds information for individual pointers (sizes reported for 64 bit architectures). Using virtual memory and paging, the kernel only fills/allocates these data structures on demand, thus keeping actual memory use relatively low.

BT can accommodate metadata for up to 2^{17} pointers. A pathological case with dedicated physical memory would thus require a base of 2 GB in addition to 4 MB for each pointer. To reduce this high memory overhead, the Linux kernel only allocates physical memory to the regions of the BD that are actually used, and only allocates individual BTs when they are accessed. When a bounds check fails, the CPU issues an exception that must be handled by the kernel. Current kernel implementations allow user-space processes to register their own error handlers which can then either log or abort on bound violations.

3 | RELATED WORK

Research on memory safety has long roots in both industry and academia, with many solutions proposed to detect, mitigate, or eliminate various types of memory errors. However, the vast majority of these have not treated deployability as a primary consideration, and so have not been deployed or used in real systems. Purify [18], Shadow Guarding [19,20], SoftBound [12], and other similar approaches [21,22,23,24,25] have unacceptably high run-time performance overhead. CCured [26] and Cyclone [27] require source code changes and are not backward compatible.

Moreover, these solutions have focused on user-space and usually cannot be applied in kernel-space. Some recent notable exceptions are kCFI [13] and KENALI [14], but from an implementation standpoint, neither of these have targeted integration with the mainline kernel. To our knowledge, none of these are used as run-time security mechanisms in production systems. A notable exception is the PaX/Grsecurity patches that have pioneered many in-use security mechanisms, such as Address Space Layout Randomization [28]. PaX/Grsecurity also includes a `PAX_REFCOUNT` [29] feature that prevents reference counter overflows, but requires extensive modification of the underlying atomic types. These extensive changes, and a potential race condition, made this feature unsuitable for mainline kernel adoption [29].

Other tools are widely used to debug memory errors during development. For example, Valgrind [30] offers a suite of six tools for debugging, profiling, and error detection, including the detection of memory errors. AddressSanitizer [31] is arguably the state-of-the-art in run-time memory error detection, but is unsuitable for production use due to performance overhead. KASAN [15] is integrated into the mainline Linux kernel and offers similar protections for the kernel, but again incurs performance overheads that are unacceptably high for most production use cases.

From a production perspective, much work has focused on preventing the exploitation of memory errors. Exploitability of buffer overflows, whether stack-based or heap-based, can be limited by preventing an attacker from misusing overflown data. One early mitigation technique is the non-executable (NX) bit for stack, heap and data pages [32,33]. However, this can be circumvented by using overflows for execution redirection into other legitimate code memory, such as the C library in so called return-to-libc attacks, or more generally to any executable process memory in return oriented programming (ROP) attacks [34]. Another mitigation technique is the use stack canaries to detect stack overflows, e.g., StackGuard [35] and StackShield [36]. However, these detection techniques can typically be circumvented using more selective overflows that avoid the canaries, or by exploiting other callback pointers such as exception handlers. Probabilistic mitigation techniques, such as memory randomization [28,37], are

commonly used, but have proven difficult to secure against indirect and direct memory leaks that divulge the randomization patterns, or techniques such as heap spraying[38].

In contrast to the development/debugging temporal safety tools and the run-time-friendly mitigation measures, MPXK is a run-time efficient system focused on the prevention of the underlying memory errors. MPXK shares some conceptual similarities with previous solutions[39,40,26,27,12]. Like these systems, MPXK does not use fat-pointers, which alter the implementation of pointers to store both the pointer and the bounds together, but instead preserves the original memory layout. Unlike purely software-based systems, such as SoftBound[12], MPXK has the advantage of hardware registers for propagating bounds and hardware instructions for enforcing bounds. HardBound[41] employs hardware support similar to MPXK, but has a worst-case memory overhead of almost 200%. However, HardBound has only been simulated on the micro-operation level and lacks any existing hardware support. Unlike MPXK, none of these previous schemes have been designed for use in the kernel.

4 | PROBLEM STATEMENT

As explained above, two major causes of memory errors are: i) temporal memory errors caused by reference counter overflows; and ii) spatial memory errors arising from out-of-bounds memory accesses. Our objective is to develop solutions that *proactively* protect the Linux kernel against these two causes of memory errors. Specifically, we define the following requirements:

- **Reference counter overflows**: we require a reference counter type and associated API in which the counter is *guaranteed never to overflow*.
- **Out-of-bounds memory accesses**: we require an access control scheme for kernel memory that *prevents out-of-bounds accesses*.

In addition to the above requirements, the following are mandatory design considerations in order for the above solutions to be used in practice:

- **Performance:** Some kernel subsystems, such as networking and filesystem, have strict performance requirements. Any security mechanisms that are perceived to add unacceptable performance penalty risk being rejected by the maintainers of these subsystems. However, there is no kernel-wide definition of what constitutes an unacceptable performance penalty, as this differs per subsystem. Furthermore, the Linux kernel runs on a vast range of devices, including closed fixed-function devices like routers, where software attacks are not a threat but performance requirements are stringent. Thus, the balance between security and performance must be configurable, in order to meet the constraints of different usage scenarios.
- **Deployability:** In order to be integrated into the mainline Linux kernel, a solution should follow the Linux kernel design guidelines, minimize the number and extent of kernel-wide changes, and be sufficiently easy to use and maintain. *Usability for kernel developers* is particularly crucial for new features that may be adopted at the discretion of subsystem developers.

5 | REFERENCE COUNTER OVERFLOWS

A prerequisite for the analysis and conversion of reference counters is to identify all uses of reference counters in the kernel. This is non-trivial because i) reference counters are often implemented using a general-purpose atomic integer type, which is also used for other purposes; and ii) not all reference counters follow conventional practices. Unconventional reference counter implementations cannot be ignored because these are usually more likely to be error-prone, and would thus benefit the most from our security mechanism.

Recently, a new `refcount_t` type and a corresponding minimal API were introduced by one of the Linux maintainers, Peter Zijlstra. This prevents incrementing a reference counter from zero or overflowing the counter. However, our kernel-wide analysis showed that the `refcount_t` API was too restrictive to be used in certain kernel subsystems. To overcome this, we proposed several additions to the initial `refcount_t` API, making it widely usable as a replacement for existing reference counters. Using this improved API, we have developed a set of patches to *convert all conventional reference counters in the kernel* to use `refcount_t`.

5.1 | Analyzing Linux Kernel reference counters

We use Coccinelle[42], a static analyzer integrated into the Linux kernel build system (KBuild), to systematically analyze the kernel source code and locate all reference counters based on the `atomic_t` type. Coccinelle takes code patterns as input and finds (or replaces) their occurrences in the given code base. We defined three such code patterns to identify reference counters based on their behavior (see Listing 3 in the Appendix for the full patterns):

1. Using `atomic_dec_and_test` (or one of its variants) to decrement an atomic variable, testing if the resulting value is zero, and if so, freeing a referenced object. This is the archetypical reference counter use case.
2. Using `atomic_add_return` to decrement a variable (by adding -1), and comparing its updated value against zero. This is a variation of the basic `dec_and_test` case using a different function.
3. Using `atomic_add_unless` to decrement a counter only when its value is not one. This case is less common.

These patterns are strong indicators that the identified object employs an `atomic_t` reference counting scheme. So far, this approach has detected all occurrences of reference counters. Some false positives were reported, particularly in implementations that make use of `atomic_t` variables for purposes other than reference counting. For example, under one condition an object might be freed when the counter reaches zero (like a reference counter), but under a different condition the object might instead be recycled when the counter reaches zero (see Section 5.5 for examples). Of the 250 `atomic_t` variables reported on an unmodified v4.10 kernel we have manually confirmed 233 variables as reference counters.

5.2 | refcount_t API

The initial `refcount_t` API, introduced by Peter Zijlstra[1] (Listing 1), was designed around strict, semantically correct, reference counter use. This meant that beyond `set` and `read` calls, the API provided only two incrementing functions and three decrementing functions.

Both incrementing functions refuse to increment a counter when its value is zero, in order to avoid potential use-after-free errors as illustrated in Figure 2 . The `refcount_inc_not_zero` function returns a `true` value if the counter was non-zero, indicating that the referenced object is safe to use. Alternatively, `refcount_inc` can be used to avoid redundant checks in situations where the resulting value will definitely be positive.

The three decrementing functions return a value of `true` if the counter value reaches zero, and include compile-time warnings to ensure that this return value is checked. When a reference counter reaches zero, the object should always be released to avoid a memory leak. The `refcount_dec_and_mutex_lock` and `refcount_dec_and_lock` atomically combine the counter decrement with acquiring a mutex or lock.

Listing 1: Initial bare `refcount_t` API.

```
void refcount_set(refcount_t *r, unsigned int n);
unsigned int refcount_read(const refcount_t *r);
bool refcount_inc_not_zero(refcount_t *r);
void refcount_inc(refcount_t *r);
bool refcount_dec_and_test(refcount_t *r);
bool refcount_dec_and_mutex_lock(refcount_t *r, struct mutex *lock);
bool refcount_dec_and_lock(refcount_t *r, spinlock_t *lock);
```

The main challenge in designing the `refcount_t` API was how to deal with an event that would otherwise cause the counter to overflow. One approach would be to simply ignore the event, such that the counter remains at its maximum value. However, this means that the number of references to the object will be greater than the maximum counter value. If the counter is subsequently decremented, it will reach zero and the object will be freed before all references have been released, leading to a use-after-free error. To overcome this challenge, the `refcount_t` API instead *saturates* the counter, such that it remains at its maximum value, even if there are subsequent increment *or decrement* events (Algorithms 1 and 2). A saturated counter would, therefore, result in a memory leak since the object will never be freed. However, a cleanly logged memory leak is a small price to pay for avoiding the potential security vulnerabilities of a reference counter overflow. This approach is similar to that previously used by the PaX/Grsecurity patches[29].

[1] http://lwn.net/Articles/713645/

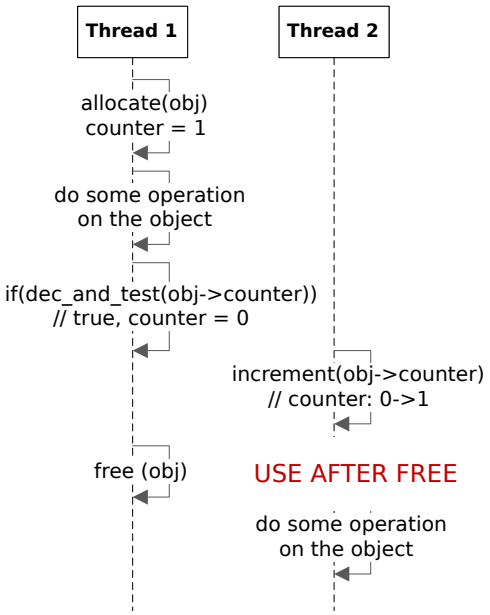


FIGURE 2 Potential use-after-free when incrementing a reference counter from zero.

Algorithm 1 `High-level refcount_inc_and_test` saturation behavior. Instead of allowing overflow, the counter is set to its maximum value, thus preventing any further changes.

1: **if** $value == 0$ **then**
2: **return** false
3: **end if**
4: **if** $value + 1 < value$ **then**
5: $value \leftarrow$ `MAX_VALUE` ▷ Saturate value on overflow
6: **else**
7: $value \leftarrow value + 1$
8: **end if**
9: **return** true

5.3 | Our extensions to the `refcount_t` API

We conducted a systematic analysis of existing reference counters in the Linux kernel source code. This revealed several variations of the strict archetypical reference counting schemes that are incompatible with the initial `refcount_t` API. As a result,

Algorithm 2 `High-level refcount_dec_and_test` saturation behavior. A return value of true indicates that the value is non-zero and that the referenced object is usable, not whether the decrement actually took place or not. Note that underflows are prevented and treated exactly as a successful non-zero decrement.

```
1: if value == MAX_VALUE then
2:     return false                                   ▷ Value is saturated
3: end if
4: if value − 1 > value then
5:     return false                                   ▷ Underflow
6: end if
7: value ← value − 1
8: return value == 0                                  ▷ Return value is true only when reaching zero
```

we designed several API additions that have subsequently been integrated into the Linux kernel.[2] Our new API calls are shown in Listing 2.

Listing 2: Our additions to the `refcount_t` API.

```
void refcount_add(unsigned int i, refcount_t *r);
bool refcount_add_not_zero(unsigned int i, refcount_t *r);
void refcount_dec(refcount_t *r);
bool refcount_dec_if_one(refcount_t *r);
bool refcount_dec_not_one(refcount_t *r);
void refcount_sub(refcount_t *r);
bool refcount_sub_and_test(unsigned int i, refcount_t *r);
```

The functions allowing arbitrary additions and subtractions i.e., `refcount_add` and `refcount_sub`, are necessary for situations in which larger value changes occur. For example, the `sk_wmem_alloc`[3] variable in the networking subsystem serves as a reference counter but also tracks the transfer queue, and is thus subject to arbitrary additions and subtractions. The return values of `refcount_sub_and_test` and `refcount_add_not_zero` indicate whether the counter has reached zero. We introduced `refcount_dec` to accommodate situations in which the counter will definitely be non-zero and a forced return value check would incur needless overhead. For instance, some functions in the `btrfs` filesystem[4] handle nodes that are guaranteed to be cached, i.e., there will always be at least one reference held by the cache. Finally, `refcount_dec_if_one` and `refcount_dec_not_one` enable schemes that require specific operations before or instead of releasing objects. For instance, the networking subsystem extensively uses patterns where a reference counter value of one indicates that an object is invalid but can be recycled.

5.4 | Implementation considerations

To avoid costly lock or mutex use, the generic implementation of the `refcount_t` API—i.e., non architecture-specific implementation—uses the *compare-and-swap* pattern, which is built around the atomic `atomic_cmpxchg` function shown in Algorithm 3. On x86, `atomic_cmpxchg` is implemented as a single atomic CPU instruction (`cmpxchg`), but the implementation is guaranteed to be atomic regardless of architecture. The function always returns the prior value but exchanges it only if it was equal to the given condition value *comp*. Compare-and-swap works by indefinitely looping until a `cmpxchg` succeeds. This avoids costly locks and allows all but the `cmpxchg` to be non-atomic. Note that in the typical case, without concurrent modifications, the loop runs only once, thus being much more efficient than synchronization mechanisms (e.g., a lock or mutex).

As an example of the extended `refcount_t` API implementation, consider the `refcount_add_not_zero` function shown in Algorithm 4. It is used to increase `refcount_t` when acquiring a reference to the associated object, and a return value of `true` indicates that the counter was non-zero and thus the associated object is safe to use. The actual value from a user perspective is irrelevant and `refcount_add_not_zero` guarantees only that the return value is true if, and only if, the value of `refcount_t` at the time of the call was non-zero. Internally the function further guarantees that the increment takes place only when the prior

[2] http://www.openwall.com/lists/kernel-hardening/2016/11/28/4
[3] http://elixir.free-electrons.com/linux/v4.10/source/include/net/sock.h}L389
[4] E.g., http://elixir.free-electrons.com/linux/v4.10/source/fs/btrfs/raid56.c}L789.

Algorithm 3 `cmpxchg`(*atomic*, *comp*, *new*) sets the value of *atomic* to *new* if, and only if, the prior value of *atomic* was equal to *comp*. Irrespective of whether the value was changed, the function always returns the prior value of *atomic*. In practice this function is implemented as a single inline CPU instruction.

```
1: old ← atomic.value
2: if old = comp then
3:     atomic.value ← new
4: end if
5: return old
```

value was in the open interval (0, `UINT_MAX`), thus preventing use-after-free due to either increment from zero or overflow. This case results in the default return statement at line 20 of Algorithm 4. Attempted increment from zero or `UINT_MAX` results in returns at lines 3 and 6 respectively. Finally, an addition that would overflow the counter instead saturates it by setting its value to `UINT_MAX` on line 10.

Algorithm 4 `refcount_add_not_zero` (*refcount*, *summand*) attempts to add a *summand* to the value of *refcount*, and returns true if, and only if, the prior value was non-zero. Note that the function is not locked and only the `cmpxchg` (line 13) is atomic, i.e., the value of *refcount.value* can change at any point of execution. The loop ensures that the `cmpxchg` eventually succeeds. Overflow protection is provided by checking if the value is already saturated (line 6) and by saturating instead of overflowing on line 10.

```
1: val ← refcount.value                                   ▷ use local copy
2: while true do
3:     if val = 0 then
4:         return false                                   ▷ counter not incremented from zero
5:     end if
6:     if val = UINT_MAX then
7:         return true                                    ▷ counter is saturated, thus not zero
8:     end if
9:     new ← val + summand                                ▷ calculate new value
10:    if new < val then
11:        new ← UINT_MAX                                 ▷ Saturate instead of overflow
12:    end if
13:    old ← atomic_cmpxchg(refcount.value, val, new)
14:    if refcount.value was unchanged then
15:        old = val
16:        break                                          ▷ value was updated by cmpxchg
17:    end if
18:    val ← old                                          ▷ update val for next iteration
19: end while
20: return true                                           ▷ value incremented
```

5.5 | Challenges

Despite the additions to the `refcount_t` API, several reference counting instances nonetheless required careful analysis, and in some cases modifications to the underlying logic. These challenges were the main reason for not automating the conversion of reference counting schemes using our Coccinelle patterns. The most common challenges are *object pool patterns* and *non-conventional* reference counters.

For example, some implementations of the object pool pattern use negative values of the reference counter to distinguish objects that should be recycled from those that should be freed. Our additions to the `refcount_t` API support this pattern

without resorting to negative reference counter values, by using the value of one to indicate that the object should be recycled. These implementations therefore often necessitated non-trivial changes to ensure that neither increment on zero operations nor negative values are expected. Overall we encountered 6 particularly challenging recycling schemes. For example, the `inode refcount_t` conversion spanned a total of 10 patches.[5]

In some cases, reference counters were used in non-conventional ways, such as to govern other behavior or track other statistics in addition to the strict reference count itself (e.g., the network socket `sk_wmem_alloc` variable described above). Straightforward conversion to use the `refcount_t` API can be surprising or outright erroneous for such unconventional uses; it might for instance be expected that such variables can be incremented from zero or potentially reach negative values. In our conversion efforts we encountered 21 distinct reference counters in this category.

5.6 | Deployment of refcount_t

We developed 233 distinct kernel patches, each converting one distinct variable, spanning all the kernel subsystems. During our work, the `refcount_t` API, with our additions, was also finalized in the mainline Linux kernel. The next stage of our work consisted of submitting all the patches to the respective kernel subsystem maintainers and adapting them based on their feedback. Based on discussions with maintainers, some patches were permanently dropped, either because they would require extensive changes in affected subsystems or would incur unacceptable performance penalty without any realistic risk of actually overflowing the particular counter. As explained in Section 4, some performance-sensitive systems proved challenging to convert due to performance concerns. As a result, a new `CONFIG_REFCOUNT_FULL` kernel configuration option was added to allow switching the `refcount_t` protections off and thus use the new API without any performance overhead. This can be utilized by devices that have high performance requirements but are less concerned about security based on their nature (e.g., closed devices such as routers that do not allow installation of untrusted software). These patching efforts, discussions, and patch reviews also uncovered prior reference counting bugs that were fixed in subsequent kernel patches.[6]

6 | OUT-OF-BOUNDS MEMORY ACCESS

To prevent spatial memory errors in the Linux kernel, we have adapted Intel MPX for in-kernel use. The resulting solution is called *MPX for Kernel (MPXK)*. Although MPX is currently only available in Intel processors, it covers a very large share of existing devices (e.g., more than 99% of all servers have Intel processors[43]). Moreover, we hope that if this technology proves to be successful in preventing spatial memory errors, other CPU vendors would introduce similar technologies. Since the MPX instrumentation (see Section 2.2) utilizes the architecture-independent Pointer Bounds Checker, it should be straight-forward to adapt the software components to support similar features from other vendors. The current MPX hardware can be used in both user-space and kernel-space because it has two distinct sets configuration registers. However, as explained in Section 2.2, current support for MPX is only available for user-space applications, and requires the assistance of the kernel[7]. Prior to our work, MPX has not been used to protect kernel-space.

6.1 | Challenges

The following challenges arise when attempting to use MPX in kernel-space:

Memory use: The MPX Bound Directory (BD) and Bound Tables (BT) incur a high memory overhead. As explained in Section 2.2, user-space MPX attempts to reduce this overhead by allocating this memory only when needed. However, this requires the kernel to step in at arbitrary points to handle page faults or Bound Faults caused by BD dereferences or unallocated BTs. This approach cannot be used in the kernel because the kernel cannot handle page faults at arbitrary points *within its own execution*. It is also not feasible to pre-allocate the BD and BTs, as this would increase base memory usage by over 500%[8] and require extensive modifications to accommodate certain classes of pointers (e.g., pointers originating from user-space). An

[5] http://lkml.org/lkml/2017/2/24/599
[6] http://lkml.org/lkml/2017/6/27/567, http://lkml.org/lkml/2017/3/28/383 etc.
[7] Available since the Linux 3.19 kernel release.
[8] Each potential pointer, i.e., any used memory, would require a pre-allocated BD entry and BT entry, which is the size of four pointers.

alternative approach would be to substitute the hardware-backed BD and BTs with our own metadata, similar to SoftBound[12] or KASAN[15]. However, this would still incur the same memory and performance overheads as those systems.

Kernel support code: MPX is supported in user-space by GCC library implementations of various support functionality, such as initialization and function wrappers. The user-space instrumentation initializes MPX during process startup by allocating the BD in virtual memory and initializing the MPX hardware. However, this existing initialization code cannot be used directly in the Linux kernel because kernel-space MPX must be configured during the kernel boot process.

In user-space, the compiler also provides instrumented wrapper functions for all memory manipulation functions, such as `memmove`, `strcpy`, and `malloc`. These user-space wrappers check incoming pointer bounds and ensure that the correct bounds are associated with the returned pointers. They are also responsible for updating the BD and BTs (e.g., `memcopy` must duplicate any BD and BT entries associated with the copied memory). However, these user-space wrappers also cannot be used in the kernel because the kernel implements its own standard library.

Binary compatibility (Mixed code): User-space MPX is binary compatible and can therefore be used in mixed environments with both MPX-enabled code and legacy (non-instrumented) code. A fundamental problem for any binary-compatible bounds checking scheme is that the instrumentation *cannot track pointer manipulation performed by legacy code* and therefore cannot make any assumptions about the pointer bounds after the execution flow returns from the legacy code. MPX offers a partial solution by storing the pointer's value together with its bounds (using the `bndstx` instruction) before entering legacy code. When the legacy code returns, MPX uses `bndldx` to load the bounds again, but since the pointer has been modified, MPX will reset the bounds, essentially making them infinite. This means that pointers changed by legacy code (legitimately or otherwise) can no longer be tracked by MPX.

6.2 | Design of MPXK

In this section we describe the design of our solution, MPXK, and explain how we solved the above challenges.

At runtime, MPXK prevents spatial memory errors in the same way as user-space MPX, i.e., by checking pointer bounds before a pointer is dereferenced. On its base MPXK determines pointer bounds the same way as MPX: from metadata set by the MPX compile-time instrumentation. However, when the pointer bounds cannot be propagated using the MPX instrumentation (via MPX registers), MPXK does not use the Bound Directory (BD) and Bound Tables (BTs) to store and fetch them, but uses its own method, as explained in the *memory use* paragraph below. Both user-space MPX and MPXK perform narrowing of bounds in exactly the same way, as outlined in Section 2.2.

Memory use: Instead of using BD and BTs for storing and retrieving pointer bounds, MPXK determines them by using *existing kernel metadata*. Specifically, we re-use the kernel memory management metadata created by `kmalloc`-based allocators that already contains the information about the sizes of allocations. Thus, MPXK does not need to have any additional dynamic storage for the bounds information. We define a new function, `mpxk_load_bounds`, that queries this existing kernel metadata using a kernel-provided interface to determine the bounds for a pointer allocated by `kmalloc`, and loads these into the MPX registers. In case a pointer has not been allocated using `kmalloc`-based allocator, the function returns empty bounds. A side-effect of using existing kernel metadata is that retrieved bounds are rounded up to the nearest allocator cache size (i.e., may be slightly larger than the requested allocation size). However, this has no security implications because the allocator will not allocate any other objects in the round-up memory area[44].

Kernel support code: Since we cannot use the existing MPX user-space memory management wrappers, we have developed similar kernel-specific wrapper functions that are implemented as normal in-kernel library functions. These MPXK wrappers are significantly less complex (and thus easier to audit for security) than their user-space counterparts because the MPXK wrappers do not need to include logic for updating the BD or BTs. In addition to the memory management wrappers, we have also developed new code to initialize the MPX hardware during the kernel boot process. This addresses the second challenge described in Section 6.1.

Binary compatibility: Since in both MPX and MPXK, function arguments rely on the caller to supply bounds, neither scheme can determine bounds for arguments originating from non-instrumented code, and thus these bounds cannot be checked. As future work, we are investigating how to determine such bounds using the `mpxk_load_bounds` function. However since MPXK does not use the `bndstx` and `bndldx` instructions but instead attempts to load such bounds using `mpxk_load_bounds`, it can continue tracking pointers that are modified by legacy code, provided the pointer is supported by `mpxk_load_bounds`.

Table 1 summarizes the main differences between user-space MPX and MPXK.

TABLE 1 Summary of the main differences between MPX and MPXK.

	MPX	MPXK
Hardware initialization	At process start	At kernel boot
Dynamic bounds storage	Uses BD and BTs	Re-uses kernel metadata[9]
Memory management function wrappers	Compiler-provided user-space wrappers	Kernel's own lightweight wrappers
Pointers modified by legacy code	Cannot be tracked	Can be tracked if supported by `mpxk_load_bounds`

Kernel instrumentation details

Our MPXK instrumentation is based on the existing MPX support in GCC, but uses our new GCC plugin to adapt this for use in the kernel. Specifically, our plugin instruments the kernel code with the new MPXK bound loading function, MPXK initialization code, and kernel-space wrapper functions described above. We use the GCC plugin system that has been incorporated into Kbuild, the Linux build system since Linux v4.8, to ensure that MPXK is seamlessly integrated with the regular kernel build workflow. To include MPXK instrumentation, a developer simply needs to add predefined MPXK flags to any Makefile entries. The plugin itself is implemented in four compiler passes, of which the first three operate on the high-level intermediate representation, GIMPLE, and the last on the lower-level Register Transfer Language (RTL), as follows:

1. *mpxk_pass_wrappers* replaces specific memory-altering function calls with their corresponding MPXK wrapper functions, e.g., replacing `kmalloc` calls with `mpxk_wrapper_kmalloc` calls.
2. *mpxk_pass_cfun_args* inserts MPXK bound loads for function arguments where bounds are not passed via the four `bndx` registers. This naturally happens when more than four bounds are passed, or due to implementation specifics for any argument beyond the sixth.
3. *mpxk_pass_bnd_store* replaces `bndldx` calls with MPXK bound loads, and removes `bndstx` calls. This covers all high-level (GIMPLE) loads and saves, including return values to legacy function calls.
4. *mpxk_pass_sweeper* is a final low-level pass that removes any remaining `bndldx` and `bndstx` instructions. This pass is required to remove instructions that are inserted during the expansion from GIMPLE to RTL.

The source-code for our MPXK-plugin is available at https://github.com/ssg-kernel-memory-safety.

7 | EVALUATION

We evaluate our proposed solutions against the requirements defined in Section 4.

7.1 | Security guarantees

We analyze the security guarantees of both `refcount_t` and MPXK, first through a principled theoretical analysis, and second by considering the mitigation of real-world vulnerabilities.

Reference counter overflows

With the exception of `refcount_set` and `refcount_read`, all functions that modify `refcount_t` can be grouped into *increasing* and *decreasing* functions. All increasing functions maintain the following invariants:

I1: the resulting value will not be smaller than the original value;

I2: a value of zero will not be increased;

The decreasing functions maintain the corresponding invariants:

D1: the resulting value will not be larger than the original value;

[9] With the exception of the case, described in Section 7.1 *Indirect pointers* paragraph.

D2: a value of UINT_MAX will not be decreased;

For example, the increasing function refcount_add_not_zero (Algorithm 4) maintains the invariants as follows:

- **input** = 0: Lines 3-4 prevent the counter being increased (I2).
- **input** = UINT_MAX: Lines 6-7 ensure the counter will never overflow (I1).
- **input** $\in (0, \texttt{UINT\_MAX})$: Lines 10-11 ensure that the counter value cannot overflow as a result of addition (I1).

Line 13 ensures that the addition is performed atomically, thus preventing unintended effects if interleaved threads update the counter concurrently. Regardless of how the algorithm exits, the invariant is maintained. The same exhaustive case-by-case enumeration can be used to demonstrate that all other refcount_t functions maintain the above invariants.

An attacker could still attempt to cause a use-after-free error by finding and invoking an *extra decrement* (i.e., decrement without a corresponding increment). This is a fundamental issue inherent in all reference counting schemes. However, the errors caused by the extra decrement would almost certainly be detected early in development or testing. In contrast, *missing decrements* are very hard to detect through testing as they may require millions of increments to a single counter before resulting in observable errors. Thanks to the new refcount_t, missing decrements can no longer cause reference counter overflows.

In terms of real-world impact, refcount_t would have prevented several past exploits, including CVE-2014-2851, CVE-2016-0728, and CVE-2016-4558. Although it is hard to quantify the current (and future) security impact on the kernel, our observations during the conversion process support the intuition that the strict refcount_t API discourages unsafe implementations. For example, at least two new reference counting bugs[10] were detected and fixed due to their incompatibility with the new API.

As a practical test for the protection provided by the refcount_t type and API, we have tested an attack[11] for the kernel CVE-2016-0728. The exploit abuses a bug inside kernel keyring facility: forgotten decrement of the reference counter when substituting the session keyring with the same keyring. We have made a test on Ubuntu 16.04 with 4.4 mainline kernel with the bug fix commit reverted and converting the corresponding refcounter to use the new refcount_t type. The exploit was successfully stopped (it fails to get the root credentials) and the run-time dmesg log was showing an overflow detected by the refcount_t interface. The corresponding conversion to refcount_t type was one of the first to be merged to the mainline Linux kernel.

Out-of-bounds memory access

The objective of MPXK is to prevent spatial memory errors by performing pointer bounds checking. Specifically, for objects with known bounds, MPXK will ensure that pointers to those objects cannot be dereferenced outside the object's bounds (e.g., as would be the case in a classic buffer overflow). A fundamental challenge of bounds checking schemes is thus how to determine the correct bounds for a specific pointer. MPXK combines compile- and runtime mechanisms to load bounds, which means that its ability to load bounds depends on the specific situation.

Static pointers: In the case of local or global pointers to non-dynamic memory, all checks and pointer bound handling can be determined and instrumented during compile time. At runtime such bounds are stored along with the pointers themselves (e.g., stack based pointers have stack based bounds) and updated by static instrumentation. The compile-time instrumentation also propagates bounds into and back from function calls. MPXK can therefore comprehensively perform bounds checking on all static pointers.

Dynamically allocated pointers: Pointers stored in dynamic memory cannot rely on compile-time information and thus require runtime support. The MPXK mpxk_load_bounds function uses available in-kernel memory management metadata to determine the allocated memory area for a specific memory address. Our current implementation can determine bounds for all objects allocated by kmalloc-based allocators, and support for other dynamic memory allocators could be similarly added.

Indirect pointers: Indirect pointers, e.g., pointers contained by another data-structure, are problematic because both their origin and size may be unknown at compile-time. If the pointer also points to dynamically allocated memory mpxk_load_bounds can obtain the bounds as described above. However, pointers to static memory do not have equivalent runtime metadata and therefore cannot be loaded in this way. Nevertheless, in many cases these pointers can be covered by the existing FORTIFY_SOURCE directive which inserts bounds checks based on compile-time type information.

[10] http://lkml.org/lkml/2017/6/27/409, http://lkml.org/lkml/2017/3/28/383
[11] https://www.exploit-db.com/exploits/39277/

TABLE 2 CPU load measurements (in cycles) on different CPUs.

Function	Skylake i3-6100U (stddev)	Kaby Lake i7-7500U (stddev)
atomic_inc()	15.1 (0.01)	14.7 (0.07)
refcount_inc()	38.7 (0.03)	49.1 (0.06)

Pointers in legacy code: Legacy code, i.e., code compiled without MPXK support, is by definition not protected by MPXK. Therefore when pointers originating from legacy code are passed to MPX-enabled code, the bounds are typically not known. However, unlike user-space MPX, MPXK can still use `mpxk_load_bounds` to obtain the bounds if the pointers point to dynamically allocated memory.

Pointer manipulation: If the attacker can corrupt a pointer's value to point to a different object *without dereferencing* the pointer, this can be used to subvert bounds checking schemes. For example, object-centric schemes such as KASAN enforce bounds based on the pointer's value. If this value is changed to point within another object's bounds, the checks will be made (incorrectly) against the latter object's bounds. In theory, pointer-centric schemes such as user-space MPX should not be vulnerable to this type of attack, since they do not derive bounds based on the pointer's value. However, for compatibility reasons, if MPX detects that a pointer's value has changed, it *resets* the pointer's bounds (i.e., allows it to access the full memory space). Like KASAN, MPXK will use the corrupted value to infer an incorrect set of bounds. MPXK is therefore no worse than MPX or other object-centric schemes in this regard. However, this type of attack requires the attacker to have a *prior exploit* to corrupt the pointer, which should have been thwarted by MPXK in the first place.

As a practical demonstration of MPXK's real-world effectiveness, we have tested it against an exploit built around the recent CVE-2017-7184. This vulnerability in the IP packet transformation framework `xfrm` is a classic buffer overflow caused by omission of an input size verification. We first confirmed that we can successfully gain root privileges on a current Ubuntu 16.10 installation running a custom-built v4.8 kernel using the default Ubuntu kernel configuration. We then recompiled the kernel applying MPXK on the `xfrm` subsystem, which caused the exploit to fail with a bound violation reported by MPXK.

7.2 | Performance

Reference counter overflows

Although the `refcount_t` functions consist mainly of low-overhead operations (e.g., additions and subtractions), they are often used in performance-sensitive contexts. We performed various micro-benchmarks of the individual functions during the `refcount_t` development[12]. The measurements were taken by running the corresponding operation 100,000 times and calculating the average. As shown in Table 2 , `refcount_inc` introduces an average overhead of 29 CPU cycles, compared to `atomic_inc`, and the exact number of cycles varies between different tested CPU platforms. The overhead comes from additional overflow and increment-from-zero checks that `refcount_inc` performs similar to `refcount_inc_and_test` explained in detail in Algorithm 1.

However micro-benchmarks cannot be considered in isolation when evaluating the overall performance impact. To gauge the overall performance impact of `refcount_t` we conducted extensive measurements on the networking subsystem using the Netperf[45] performance measurement suite. We chose this subsystem because i) it is known to be performance-sensitive; ii) it has a standardized performance measurement test suite; and iii) we encountered severe resistance from mainline Linux kernel maintainers on performance grounds when proposing to convert this subsystem to use `refcount_t`. The concerns are well-founded due to the extensive use of reference counters (e.g., when sharing networking sockets and data) under potentially substantial network loads. The main challenge when evaluating performance impact on the networking subsystem is that there are no standardized workloads, and no agreed criteria as to what constitutes "acceptable" performance overhead. We used the 0-day test service[13] to run the tests on Haswell-EP6 processors and the mainline `v4.11-rc8` kernel. We measured the real-world performance impact of converting all 78 reference counters in the networking subsystem and networking drivers from `atomic_t` to `refcount_t`. Each individual test used a 300 second runtime and was executed three times.

As shown in Table 3 , our Netperf measurements include CPU utilization for UDP and TCP streams, and TCP throughput in MB/s and transactions per second (tps).First the results indicate that for both TCP and UDP, the average processing overhead

[12] http://lwn.net/Articles/718280/
[13] https://01.org/lkp/documentation/0-day-test-service

TABLE 3 Netperf refcount usage measurements

Netperf Test type	base (stddev)	refcount (stddev)	change (stddev)
UDP CPU use (%)	0.53 (0.17)	0.75 (0.06)	+42.1% (34%)
TCP CPU use (%)	1.13 (0.03)	1.28 (0)	+13.3% (3%)
TCP throughput (MB/s)	9358 (0)	9305 (0)	-0.6% (0%)
TCP throughput (tps)	14909 (0)	14761 (0)	-1.0% (0%)

TABLE 4 MPXK and KASAN CPU overhead comparison.

	baseline	KASAN		MPXK	
	time in *ns* (stddev)	diff in *ns* (stddev)	% diff	diff in *ns* (stdev)	% diff
No bound load.					
memcpy, 256 B	45 (0.9)	+85 (1.0)	+190%	+11 (1.2)	+30%
memcpy, 65 kB	2340 (4.3)	+2673 (58.1)	+114%	+405 (5.1)	+17%
Bound load needed.					
memcpy, 256 B	45 (0.8)	+87 (0.9)	+195%	+70 (1.5)	+155%
memcpy, 65 kB	2332 (5.8)	+2833 (28.2)	+121%	+475 (15.0)	+20%

can be substantial, ranging from 13% for UDP to 42% for TCP. However, rows 3 and 4 in Table 3 show that this overhead does not in practice affect the overall TCP throughput (both in MB/s and tps). Such processing overhead can in many situations be acceptable; Desktop systems typically only use networking sporadically, and when they do, the performance is typically limited by the ISP link speed, not CPU bottlenecks. In contrast, this might not be the case in servers or embedded systems (especially routers) where processing resources may be limited and networking a major contributor to system load. However, such systems are typically closed systems, i.e., it is not possible to install additional applications or other untrusted software, so therefore their attack surface is already reduced. In these circumstances, the `CONFIG_REFCOUNT_FULL` kernel configuration option can be used to disable the protection offered by `refcount_t` and eliminate all performance overhead. Providing this configuration option is critical to ensure that we can accommodate special cases, such as the above, whilst still allowing all other systems to benefit from these new protection mechanisms (which are expected to be enabled by default in the future).

Out-of-bounds memory access

Some CPU overhead is to be expected due to the MPXK instrumentation and bound handling. To measure this, we conducted micro-benchmarks of specific functions, as well as end-to-end performance tests with MPXK applied to entire kernel subsystems. All MPXK benchmarks were run on an i3-6100U processor with 8GB of memory running Ubuntu 16.04 LTS with a v4.8 Linux kernel.

Table 4 shows the results of our micro-benchmark on the `memcpy` function, comparing the performance of MPXK and KASAN. The results indicate that, as expected, MPXK does introduce a measurable performance overhead. However, compared to KASAN, the performance overhead is relatively small. The first tests (rows 1 and 2) are conducted for cases in which the bound information is readily available in MPXK from the compiler instrumentation. This is the common case for MPXK and the resulting overhead percentage is much smaller than for KASAN, especially as the size of copied data increases (row 2). The second set of tests (rows 3 and 4) are for the case in which the pointer bounds have to be loaded from the kernel memory management metadata. This is the worst case for MPXK and the largest overhead percentage arises copying small amounts of data (row 3). However, as the size of the copied data increases (row 4), the overall overhead percentage decreases significantly, since only one load of bounds is needed for the whole copied area. Although KASAN also detects temporal memory errors, we controlled for this by only measuring the bounds check events, not the time spent on allocating and de-allocating the memory behind a pointer, which is required for implementing protection against temporal memory errors.

In order to demonstrate the performance overhead when MPXK is enabled for a certain kernel subsystem, we have measured the `xfrm` subsystem discussed in Section 7.1. Table 5 shows the result of our end-to-end benchmark in which we measured the impact on an IPSec tunnel where `xfrm` manages the IP package transformations. This was run under the default Ubuntu kernel configuration with our patches applied and MPXK enabled on `xfrm`. We used Netperf for measurements, running five-minute

TABLE 5 Netperf measurements over an IPSec tunnel with the `xfrm` subsystem protected by MPXK.

Netperf test	baseline	MPXK	change (stddev)
UDP CPU use (%)	24.97	24.97	0.00% (0.02)%
TCP CPU use (%)	25.07	25.15	0.31% (0.29)%
TCP throughput (MB/s)	646.69	617.95	-4.44% (4.61)%
TCP throughput (tps)	1586.79	1547.85	-2.45% (1.66)%

tests for a total of 10 iterations. As shown in Table 5 , the impact on CPU performance for both UDP and TCP is negligible, and the resulting TCP throughput is not affected either.

MPXK also increases run-time memory usage and kernel image size. Although we do not use the costly MPX bound storage, some additional memory is still required to store static global and intermediary bounds. However, our memory use comparisons indicate that this memory overhead is negligible. When deploying MPXK over the `xfrm` subsystem, the memory overhead for in-memory kernel code is $110KB$, which is a 0.7% increase in total size. Similarly, the kernel image size increased by $45KB$ or 0.6%. Note that since MPXK is specifically designed for modular deployment, it can be deployed incrementally to different subsystems and, if needed, any of the above overheads can be completely removed for performance-sensitive modules or subsystems.

7.3 | Deployability

Reference counter overflows

Deploying a new data type into a widely used system such as the Linux kernel is a significant real-world challenge. From a usability standpoint, the API should be as simple, focused, and self-documenting as reasonably possible. The `refcount_t` API in principle fulfils these requirements by providing a tightly focused API with only 14 functions. This is a significant improvement over the 100+ functions provided by the `atomic_t` type previously used for reference counting. Of the 233 patches we submitted, 170 are currently accepted, and it is anticipated that the rest will be accepted in the near future. The `refcount_t` type has also been taken into use in independent work by other developers.[14] This is strong confirmation that the usability goals are met in practice. Our contributions also include the Coccinelle patterns for identifying and analysing reference counting schemes that are potential candidated to be converted to `refcount_t`. These patterns have been contributed by us to the mainline kernel, so that they can be used by other developers and potentially added to the Linux kernel testing infrastructure in the future.

Out-of-bounds memory access

As with user-space MPX, our MPXK design considers usability as a primary design objective. It is binary compatible, meaning that it can be enabled for individual translation units. It is also source-compatible since it does not require any changes to source code. In a limited number of cases, pointer bounds checking can interfere with valid pointer arithmetic or other atypical pointer use sometimes seen in high-performance implementations. However, such compatibility issues are usually only present in architecture-specific implementations of higher-level APIs, and can thus be accommodated in a centralized manner by annotating incompatible functions to prevent their instrumentation. Also, compatibility issues are usually found during compile time and are thus easily detected during development. MPXK is fully integrated into Kbuild and provides predefined compilation flags for easy deployment. Using the `MPXK_AUDIT` parameter, MPXK can also be configured to run in permissive mode where violations are only logged, which is useful for development and incremental deployment. This gives the system administrators a choice of when to enable the more restrictive mode, thus facilitating different deployment models.

The MPXK code-base is largely self-contained and thus easy to maintain. The only exceptions are the wrapper functions that are used to instrument memory manipulating functions. However, this is not a major concern because the kernel memory management and string APIs are quite stable and changes are infrequent. The MPXK compiler support is all contained within a GCC-plugin, and is thus not directly dependent on GCC internals. MPXK is thus both easy to deploy and easy to maintain.

[14] http://lkml.org/lkml/2017/6/1/762

8 | DISCUSSION

The solutions proposed in this paper and submitted as mainline Linux kernel patches are a step towards improving memory safety in the Linux kernel. One sobering fact when working with production systems and distributed development communities such as the Linux kernel is that there is always a trade-off between security and deployability. It is certainly possible to propose security solutions outside the mainline kernel, such as the long-lived PaX/Grsecurity[46] patches, but these must then be explicitly applied to production systems. Conversely, solutions that are integrated into the mainline kernel will provide security by default to a large number of diverse devices. In this section we reflect on our experience of working with kernel developers and subsystem maintainers in order to integrate our security mechanisms into the mainline kernel. We do not claim to provide a rigorous socio-technical analysis of this process, but rather we offer a set of recommendations, based on our experience, which we hope could be of value to other researchers and practitioners:

Understand context: It is not sufficient to simply read and follow the Linux kernel contribution guidelines[47] when developing your proposal. It is critical to understand the *context* of the subsystem to which you are attempting to contribute. By context, we mean the recent history and planned developments of the subsystem, the standard ways in which it is verified and tested, and the overall direction set by its maintainers. Presenting your contribution within the correct context removes any initial push-back from other developers and allows the discussion to proceed to the core technical contribution. For example, in our MPXK work, instead of contributing changes directly to the GCC core, we implemented our GCC instrumentation as a standalone GCC plugin (Section 6) using the GCC kernel plugin framework that had been recently added to the mainline Linux kernel. This aligned our contribution with the current direction of the GCC compiler development.

Enable incremental deployability: In the Linux kernel, different subsystems are typically managed by distinct maintainers. Any solutions that require simultaneous changes to all subsystems are thus unlikely to be accepted. This has been cited as a major reason for kernel developers turning down the PaX/Grsecurity solution for reference counters. Instead, solutions that can be incrementally deployed, whilst still providing benefits, are more likely to receive a favourable reception from at least some developers. The initial deployments can serve as demonstrators to support the case for deployment in other subsystems. For example, the conversion to the new `refcount_t` API can happen independently for each reference counter. Kernel maintainers can thus gradually change their code to use `refcount_t` rather than requiring all kernel code to be changed at once. Similarly, we designed MPXK such that it can be independently enabled for individual compilation units or subsystems.

Provide fine-grained configurability: The Linux kernel runs on a wide range of devices with different environments, threat models, and security requirements. Thus any security solution (and especially those with performance or usability implications) must support the ability to be configured to several different *grades*. For example, MPXK can be enabled on selected subsystems in order to provide protection where needed whilst minimizing performance overhead (see Section 7.3). Similarly, in our reference counter protection work, in addition to having a configuration flag for turning off the protection behind the `refcount_t` interface, there is ongoing work to provide an architecture-specific fast assembly implementation that provides similar security guarantees[48].

Consider timing: Deploying any new feature that affects more than a single kernel subsystem takes considerable time. This is largely due to the number of different people involved in maintaining various kernel subsystems and the absence of strict organization of the development process. Researchers should plan to allocate sufficient time for this process. Also, one has to take into account the different stages of the kernel release process in deciding when to send patches to maintainers for review and feedback. For example, it took us a full year to reach the current state of reference counter protection work and have 170 out of 233 patches merged.

Finally, even if a proposed feature is ultimately not accepted, both maintainers and security researchers can learn from the process, which can eventually lead to better security in the mainline kernel.

9 | CONCLUSION AND FUTURE WORK

Securing the mainline Linux kernel is a vast and challenging task. In this paper we present two solutions to proactively memory errors in the kernel: the first prevents temporal memory errors caused by reference counter overflows, whilst the second provides hardware-assisted checking of pointer bounds to mitigate spatial memory errors. Both solutions treat practical deployability as a primary consideration, with reference counter protection already being widely deployed in the mainline kernel. While our

solutions arguably exhibit some limitations, they nonetheless strike a pragmatic balance between deployability and security, thus ensuring they are in a position to benefit the billions of devices using the mainline Linux kernel.

In terms of future work, we will continue our efforts to convert the remaining reference counters to `refcount_t`, but the initial work has already sparked other implementation efforts and a renewed focus on the problem. There are active efforts to migrate to the mainline kernel solutions that provide high-performance architecture-independent implementations[48]. Several MPXK improvements are also left as future work. Our support for bound loading can be extended with support for other allocators and memory region-based bounds for pointers not dynamically allocated. More invasive instrumentation could be used to improve corner cases where function calls require bound loading. The wrapper implementations could similarly be improved by more invasive instrumentation, or by forgoing wrappers altogether and instead employing direct instrumentation at call sites. This approach is not feasible on vanilla MPX due complications caused by the bound storage, but would be quite reasonable for MPXK.

ACKNOWLEDGMENTS

We thank the many mainline Linux kernel developers and maintainers who have participated in discussions and patiently provided enlightening feedback and suggestions. This work was supported in part by the Intel Collaborative Research Institute for Secure Computing at Aalto University, and the Cloud Security Services (CloSer) project (3881/31/2016), funded by Tekes/Business Finland.

References

1. Smalley S, Vance C, Salamon W. Implementing SELinux as a Linux security module. *https://www.nsa.gov/resources/everyone/digital-media-center/publications/research-papers/assets/files/implementing-selinux-as-linux-security-module-report.pdf*. 2006;.
2. Bauer M. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal.* 2006;2006(148):13.
3. Integrity Measurement Architecture (IMA) wiki pages. *http://sourceforge.net/p/linux-ima/wiki/Home/*. 2017;.
4. dm-verity project pages. *http://source.android.com/security/verifiedboot/dm-verity*. 2017;.
5. National Vulnerability Database: Statistics for kernel vulnerabilities. *https://nvd.nist.gov/vuln/search/statistics?adv_search=trueŹform_type=advancedŹresults_type=statisticsŹquery=kernel*. 2017;.
6. Cook K. Status of the Kernel Self Protection Project. *www.outflux.net/slides/2016/lss/kspp.pdf*. 2016;.
7. Kernel Self Protection Project wiki. *http://www.kernsec.org/wiki/index.php/Kernel_Self_Protection_Project*. 2017;.
8. Raheja S, Munjal G, others . Analysis of Linux Kernel Vulnerabilities. *Indian Journal of Science and Technology.* 2016;9(48).
9. Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. *Proceedings of the Second Asia-Pacific Workshop on Systems.* 2011;:5.
10. Collins GE. A Method for Overlapping and Erasure of Lists. *Communications of the ACM.* 1960;3(12):655–657.
11. McKenney PE. *Overview of linux-kernel reference counting*. tech. rep. n2167=07-0027: Linux Technology Center, IBM Beaverton; 2007.
12. Nagarakatte S, Zhao J, Martin M, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2009;44(6):245–258.

13. Rigo S, Polychronakis M, Kemerlis VP. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. *https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf*. 2017;.

14. Chengyu S, Byoungyoung L, Kangjie L, Harris WR, Taesoo K, Wenke L. Enforcing Kernel Security Invariants with Data Flow Integrity. *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. 2016;.

15. The Kernel Address Sanitizer (KASAN). *www.kernel.org/doc/html/v4.10/dev-tools/kasan.html*. 2017;.

16. Nikolenko V. Exploiting COF Vulnerabilities in the Linux kernel. *ruxcon.org.au/assets/2016/slides/ruxcon2016-Vitaly.pdf*. 2016;.

17. Ramakesavan R, Zimmerman D, Singaravelu P. Intel Memory Protection Extensions (Intel MPX) enabling guide. *http://pdfs.semanticscholar.org/bd11/4878c6471cb5ae28546a594bf25ba5c25c6f.pdf*. 2015;.

18. Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. *In Proc. of the Winter 1992 USENIX Conference*. 1991;:125–138.

19. Patil H, Fischer CN. Efficient Run-time Monitoring Using Shadow Processing. *AADEBUG*. 1995;95:1–14.

20. Patil H, Fischer C. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw., Pract. Exper.*. 1997;27(1):87–110.

21. Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. 1997;:13–26.

22. Yong SH, Horwitz S. Protecting C programs from attacks via invalid pointer dereferences. *ACM SIGSOFT Software Engineering Notes*. 2003;:307–316.

23. Xu W, DuVarney DC, Sekar R. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. *SIGSOFT Softw. Eng. Notes*. 2004;29(6):117–126.

24. Nethercote N, Fitzhardinge J. Bounds-Checking Entire Programs without Recompiling. *Proceedings of the Second Workshop on Semantics Program Analysis and Computing Environments for Memory Management SPACE 2004*. 2004;.

25. Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. *Proceedings of the 28th international conference on Software engineering*. 2006;:162–171.

26. Necula GC, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*. 2002;:128–139.

27. Grossman D, Hicks M, Jim T, Morrisett G. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*. 2005;23(1):112–139.

28. PaX address space layout randomization (ASLR). *http://pax.grsecurity.net/docs/aslr.txt*. 2003;.

29. Branco R. Grsecurity forum — Guest Blog by Rodrigo Branco: PAX_REFCOUNT Documentation. *https://forums.grsecurity.net/viewtopic.php?f=7Ź t=4173*. 2015;.

30. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007;:89–100.

31. Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A Fast Address Sanity Checker.. *USENIX Annual Technical Conference*. 2012;:309–318.

32. Solar Designer . Linux kernel patch to remove stack exec permission. *http://seclists.org/bugtraq/1997/Apr/31*. 1997;.

33. PaX non-executable pages design & implementation. *http://pax.grsecurity.net*. 2003;.

34. Krahmer S. X86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. *https://trailofbits.github.io/ctf/exploits/references/no-nx.pdf*. 2005;:1–20.

35. Cowan C, Pu C, Maier D, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium.* 1998;98:63–78.

36. A stack smashing technique protection tool for Linux. *http://www.angelfire.com/sk/stackshield.* 2011;.

37. Xu J, Kalbarczyk Z, Iyer RK. Transparent runtime randomization for security. *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on.* 2003;:260–269.

38. Ratanaworabhan P, Livshits VB, Zorn BG. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks.. *USENIX Security Symposium.* 2009;:169–186.

39. Kwon A, Dhawan U, Smith JM, Knight Jr TF, DeHon A. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* 2013;:721–732.

40. Kuvaiskii D, Oleksenko O, Arnautov S, et al. SGXBOUNDS: Memory Safety for Shielded Execution. *Proceedings of the 2017 ACM European Conference on Computer Systems (EuroSys).* 2017;.

41. Devietti J, Blundell C, Martin MK, Zdancewic S. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGARCH Computer Architecture News.* 2008;:103–114.

42. Coccinelle Project. *http://coccinelle.lip6.fr/*. 2017;.

43. Pike JP. Server CPU Predictions For 2017. *https://www.forbes.com/sites/moorinsights/2017/01/10/server-cpu-predictions-for-2017/}27adb50365a7*. 2017;.

44. Akritidis P, Costa M, Castro M, Hand S. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. *USENIX Security Symposium.* 2009;:51–66.

45. Netperf Project. *http://hewlettpackard.github.io/netperf*. 2017;.

46. Grsecurity project. *https://grsecurity.net.* 2017;.

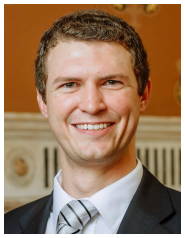
47. Submitting patches: the essential guide to getting your code into the kernel. *https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html.* 2017;.

48. Cook K. codeblog: security things in Linux v4.13. *https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/*. 2017;.

AUTHOR BIOGRAPHIES

Elena Reshetova is a security architect and researcher at the Intel Open Source Technology Center in Finland working with various Open Source platform security projects across the whole Linux platform security community. Elena is also a PhD student in the Secure Systems Group at Aalto University and part of Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

Hans Liljestrand is a PhD student in the Secure Systems Group at Aalto University. He received his MSc in Computer Science from the University of Helsinki in 2017.



Andrew Paverd is a Research Fellow in the Department of Computer Science at Aalto University and Deputy Director of the Helsinki-Aalto Center for Information Security – HAIC. He received his BSc in Electrical Engineering from the University of the Witwatersrand, Johannesburg in 2010, his MSc in Engineering from the University of Cape Town in 2012, and his DPhil in Computer Science from the University of Oxford in 2016. https://ajpaverd.org

N. Asokan is a Professor of Computer Science at Aalto University where he co-leads the Secure Systems Group and directs the Helsinki-Aalto Center for Information Security – HAIC. Before joining academia, he spent 17 years in industry research labs with IBM Research and Nokia Research Center. He received his formal education from University of Waterloo, Syracuse University, and Indian Institute of Technology, Kharagpur. http://asokan.org/asokan/

APPENDIX

Listing 3: Coccinelle pattern for finding reference counters in the Linux kernel

```
// Check if refcount_t type and API should be used
// instead of atomic_t type when dealing with refcounters
// Confidence: Moderate
// URL: http://coccinelle.lip6.fr/
// Options: --include-headers
virtual report
@r1 exists@
identifier a, x;
position p1, p2;
identifier fname =~ ".*free.*";
identifier fname2 =~ ".*destroy.*";
identifier fname3 =~ ".*del.*";
identifier fname4 =~ ".*queue_work.*";
identifier fname5 =~ ".*schedule_work.*";
identifier fname6 =~ ".*call_rcu.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)           |
 atomic_dec_and_lock@p1(&(a)->x, ...)      |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)      |
 atomic64_dec_and_test@p1(&(a)->x)         |
```

```
 local_dec_and_test@p1(&(a)->x)
)
...
(
 fname@p2(a, ...); |
 fname2@p2(...);   |
 fname3@p2(...);   |
 fname4@p2(...);   |
 fname5@p2(...);   |
 fname6@p2(...);
)
@script:python depends on report@
p1 << r1.p1;
p2 << r1.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r4 exists@
identifier a, x, y;
position p1, p2;
identifier fname =~ ".*free.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)          |
 atomic_dec_and_lock@p1(&(a)->x, ...)     |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)     |
 atomic64_dec_and_test@p1(&(a)->x)        |
 local_dec_and_test@p1(&(a)->x)
)
...
y=a
...
fname@p2(y, ...);
@script:python depends on report@
p1 << r4.p1;
p2 << r4.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r2 exists@
identifier a, x;
position p1;
@@
(
atomic_add_unless(&(a)->x,-1,1)@p1      |
atomic_long_add_unless(&(a)->x,-1,1)@p1 |
atomic64_add_unless(&(a)->x,-1,1)@p1
)
@script:python depends on report@
p1 << r2.p1;
@
msg = "atomic_add_unless"
coccilib.report.print_report(p1[0], msg)
@r3 exists@
identifier x;
position p1;
@@
(
x = atomic_add_return@p1(-1, ...);       |
x = atomic_long_add_return@p1(-1, ...); |
x = atomic64_add_return@p1(-1, ...);
)
@script:python depends on report@
p1 << r3.p1;
@@
msg = "x␣=␣atomic_add_return(-1,␣...)"
coccilib.report.print_report(p1[0], msg)
```

How to cite this article: E. Reshetova, H. Liljestrand, A. Paverd, and N. Asokan (2018), title, *Software: Practice and Experience 2018;XX:X–X*.