Linux Kernel Memory Safety

Hans Liljestrand

Master's Thesis
University of Helsinki
Department of Computer Science

Helsinki, October 19, 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Hans Liljestrand

Työn nimi — Arbetets titel — Title

Linux Kernel Memory Safety

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	October 19, 2017	87 pages + 4 appendix pages

Tiivistelmä — Referat — Abstract

Linux kernel vulnerabilities are often long lived and in some cases challenging to patch after discovery. The current focus in upstream Linux security has therefore been on categorical protections against whole error classes, not only reactive patching of specific vulnerabilities. Our work contributes to these efforts by tackling memory errors in the Linux kernel from two different fronts. First, we contributed to the upstream Linux kernel by working on a mechanism to prevent use-after-free errors caused by reference counter overflows. Second, we explored the applicability of Intel MPX as a general mechanism to prevent spatial memory errors in the Linux kernel.

ACM Computing Classification System (CCS):
Security and privacy→Systems Security→Operating systems security

Avainsanat — Nyckelord — Keywords
Security, Operating Systems Security, Memory Safety, C, Linux

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

Abbreviations and Acronyms

API Application Programming Interface

ASLR Address Space Layout Randomization

BD Bound Directory

BT Bound Table

BTE Bound Table Entry

CFG Control Flow Graph

CVE Common Vulnerabilities and Exposures

GCC GNU Compiler Collection

GNU GNU's Not Unix

HAL Hardware Abstraction Layer

ICC Intel C++ Compiler

IoT Internet of Things

IPA Interprocedural Analysis

KASAN Kernel Address Sanitizer

KASLR Kernel Address Space Layout Randomization

KSPP Kernel Self Protection Project

MAC Mandatory Access Control

MPX Memory Protection Extensions

MPXK MPX in Kernel

MSR Machine Specific Register

RCU read-copy-update

RFC Request for Comments

ROP Return Oriented Programming

RTL Register Transfer Language

SGX Software Guard Extensions

SmPL Semantic Patch Language

1 Introduction

The application space is increasingly well protected, with various new development, testing, and runtime security mechanisms being able to detect and prevent several security-critical bugs. Therefore, the kernel, with its power to circumvent any application security, has become an increasingly attractive target. We look at two aspects of Linux kernel security, both related to memory safety. First, we look at use-after-free errors due to integer overflows in the handling of shared objects. Second, we look at preventing kernel buffer-overflows and similar errors using Intel Memory Protection Extensions (MPX) hardware. We propose solutions for both. The former leading to the introduction of the new `refcount_t` type, to which we contributed with API additions and kernel-wide conversions (Section 2). The latter leading to MPXK (MPX in Kernel), a proposed in-kernel pointer bounds checker with low memory and performance overheads (Section 3).

1.1 Linux Security

Linux is traditionally considered one of the more secure operating systems. This distinction has recently waned due to security improvements by competitors and the spread of Linux itself. In particular, the prevalence of Linux in embedded, Internet of Things (IoT) and mobile devices has significantly increased availability and interest in Linux systems, and thus shone new light on its security woes. The growing number of Android devices[1] has undoubtedly been a contributor to both favorable and malicious interest in Linux.

User-facing applications, such as mobile phone applications and desktop applications, are typically easy to upgrade with minimal to no user interaction. However, embedded devices and the growing IoT horde do not share this ease of upgrade and are typically either cumbersome or downright impossible to upgrade. On mobile phones, app stores ensure that applications on mobile devices get updates, but the operating system itself is often non-trivial to update. Due to costs Android OEMs, therefore, offer updates for only a relatively short period. In November 2016 only $24,4\%$ of active Android devices were running an operating system newer than or equal to Android Marshmallow, released in May 2016 [16]. Based on the Android Open Source Project, Marshmallow runs `v3.18` of the Linux kernel [17a], meaning that over 75% of devices at that time were running a Linux kernel older than two years.

Embedded devices, IoT, and mobile phones are long-lived, which, coupled with lack of updates, means that there is a growing number of devices running on aging software. Many low-cost devices also use unsecured mainline kernels without available security patches or tools. Updates for the underlying Linux

[1] The Android market share in the first quarter of 2017 was 85% according to IDC [17i]

kernel are typically not available despite possible application updates. Kernel vulnerabilities thus provide long-lived exploits for these devices. Kernel vulnerabilities can also be used to target a multitude of devices ranging from IoT devices to servers.

Kernel bugs themselves exhibit troublesome longevity. According to a 2011 examination of Linux kernel vulnerabilities [Cor10], a vulnerability has an average lifetime of about five years, i.e., there will be five years between the introduction of a vulnerability and its eventual discovery and patching. Another similar analysis based on Ubuntu Common Vulnerabilities and Exposures (CVE) tracker data from 2011 to 2016, found 33 high-risk CVEs based on bugs with an average lifetime of 6.4 years [Coo16]. Moreover, based on data from CVE Detail, the total rate of Linux Kernel related CVEs is also on the rise [17c].

The patching and vulnerability patterns suggest that the Linux kernel cannot rely solely on a retroactive approach to fixing security issues. Therefore, to provide wide-reaching protection, the mainline kernel should itself be as secure as possible. Kernel security development must also focus on security improvements that prevent whole classes of bugs, not individual cases. These conclusions motivated the 2015 introduction of the Kernel Self Protection Project (KSPP) [Coo15]. The KSPP is an upstream project, i.e., its goal is to introduce improvements directly to the mainline kernel itself. `PaX/Grsecurity` [17h] is a long-standing security patch-set separate from the mainline kernel. It inspired KSPP and served as an initial source for upstreamed features. The first part of our work is done under the KSPP and has its roots in `PaX/Grsecurity`.

1.2 Memory Safety

A fundamental security aspect of any programming language is its memory safety, i.e., its susceptibility to memory errors. Languages with lacking memory safety can be protected using several approaches, ranging from probabilistic mitigation and detection systems to full-fledged memory safety solutions offering complete mediations of memory access [SPW13]. Memory safety is viewable from two perspectives: the temporal and spatial. A typical *temporal memory error* is the use-after-free error, i.e., using a freed pointer, whereas the stack buffer overflow is the classic example of a *spatial memory error*.

In a stack buffer overflow attack a stack-based array is overflown to overwrite other stack-based data, including function return addresses. The attack thus allows the insertion and execution of arbitrary code. Use-after-free errors happen when a freed pointer is used and can be exploited to achieve arbitrary code execution. Modern systems typically prevent code injection with `W⊕X` memory schemes, i.e., ensuring memory regions are either writeable or executable but not both. The Linux kernel has since 1997 prevented the

execution of stack-based memory [Sol97b]. Return Oriented Programming (ROP) attacks circumvent this mechanism by reusing existing program code for needed behavior [Sha07]. Later techniques mitigate ROP attacks. Address Space Layout Randomization (ASLR), for instance, obscures the memory layout to prevent an attacker from locating specific code needed for ROP attacks [PaX03a; XKI03].

The protections above, despite being crucial in mitigating vulnerabilities and reducing attack surface, do not prevent all memory errors. There have however been many proposals to achieve this, from early proposals such as Shadow Guarding [PF97; PF95], CCured [Nec+05] and Cyclone [Jim+02] to more recent contenders such as Baggy Bounds Checking [Akr+09] and SoftBound [Nag+09]. Unfortunately, these tools typically are either incompatible with existing code and libraries or provide poor runtime performance. Some systems, such as the AddressSanitizer [Ser+12] and Valgrind [NS07b], are in frequent use, although mainly in development and testing, again due to performance concerns. A later contender, featuring acceptable runtime performance, is Intel MPX [Int16] which we use in the second part of our work.

1.3 Structure of Thesis

In this thesis, we will look at some aspects of both spatial and temporal memory safety. In Section 2 we look at reference counters, a technique used to manage the lifetime of shared objects [Col60; Kro04]. Reference counters by themselves are not a security mechanism, but recent related CVEs, CVE-2014-2851 [cve14a], CVE-2016-4558 [cve16b] and CVE-2016-0728 [cve16a], show that they are a source of critical security bugs. Section 2.1 provides necessary background information and Section 2.2 details our requirements. We present both our initial proposal and the eventually used solution, in Sections 2.3 and 2.4, respectively. Section 2.5 evaluates the solution based on our requirements.

The latter part of our work is focused on preventing in-kernel spatial memory errors using Intel MPX and is presented in Section 3. The presentation is similar: Section 3.1 provides background, Section 3.2 details the requirements, Sections 3.3 and 3.4 present the solution and challenges, which are evaluated in Section 3.5. Related work is discussed in Section 4, and finally the conclusions in Section 5.

2 Preventing Reference Counter Overflows

Memory-safe languages typically manage the lifetime of objects with little to no intervention from the programmer. Garbage Collection, which automatically frees an object's memory when it is no longer needed, has been around since early proposals for Algol-68 [BL70]. Another basic technique, in C++ for instance, is using smart pointers that detect and free orphaned objects. Neither of these techniques is readily available in the Linux kernel, not only due to language limitations — the kernel implementation language is C — but also because the kernel needs fine-grained control and timing of memory operations. The kernel thus employs reference counters to track the lifetime of objects.

A reference count is a value that keeps track of how many references, i.e., pointers, to an object are in use. Reference counters are error-prone because they require direct updating by the programmer. They are used to manage shared objects, which on multi-threaded systems means that they are also subject to concurrency issues. Due to the concurrency issues, reference counters are implemented using the `atomic_t` type [McK07]. The `atomic_t`, as an all-purpose integer, leaves ample room for errors, which prompted the introduction of the `kref` type [Kro04]. Unfortunately, both types are vulnerable to reference counter overflows, i.e., an integer overflow in the underlying implementation. An overflow can cause a counter to inadvertently reach zero, thereby causing the object's memory to be freed while still in use. In this part of our work we focus on the overflow vulnerability in particular, but also consider the performance, maintainability, and usability of our solution.

This work was proposed to us by the KSPP and was initially directly based on an existing `PaX/Grsecurity` feature. We discuss reference counters and their security considerations in Section 2.1, based on which we formulate the requirements for our work (Section 2.2). We initially proposed and implemented `HARDENED_ATOMIC`, a solution based on prior work (Section 2.3). This approach was ultimately discarded but prompted the introduction of `refcount_t` to the mainline kernel. Our contributions to the latter include kernel-wide analysis of reference counters, an extension to the `refcount_t` Application Programming Interface (API), and kernel-wide conversion of existing reference counting schemes (Section 2.4). We evaluate the solution against our requirements in Section 2.5.

2.1 Background

Reference counters are, on the surface, a simple mechanism used to maintain object lifetimes. They are, however, inherently tied to concurrency issues (Section 2.1.1) and hide surprising nuance in the details (Section 2.1.2). Reference counters in the Linux kernel (Section 2.1.3) are typically imple-

mented using atomic integers or the `kref` type, both of which can overflow (Section 2.1.4). Our initial work was based on an existing `PaX/Grsecurity` solution (Section 2.1.5).

2.1.1 Concurrency

Reference counters are needed for shared objects in particular. On multi-threaded systems, this means that concurrency is an inherent part of reference counters. Therefore, it is useful to briefly consider these issues before diving into the details of reference counters themselves. High-level languages often provide concurrency mechanisms, such as locks and semaphores, that hide issues such as compiler and CPU optimizations. These can, however, cause various problems in low-level C. While some of these mechanisms and are implemented in the Linux kernel, they must often be avoided for performance considerations.

Some concurrently accessed kernel objects, such as reference counters, have minimal critical sections and exhibit only brief periods of concurrency. In such cases, locking mechanisms can introduce substantial overheads when compared to the execution of the critical section itself. For example, atomic integers typically have critical sections consisting of a single instruction. One way to avoid blocking and context switches is the `spinlock`, which continuously keeps checking the lock instead of blocking. Moreover, using atomic operations can minimize the need for locking in the first place. All functions, however, do not have native atomic implementations.

Data-types that support an atomic *compare-and-swap* operation can use it to create a thread-safe implementation of any arbitrary value-changing function. On Linux, the compare-and-swap is typically named `cmpxchg`, which is also the name of the same x86 instruction. The `cmpxchg` is shown in Algorithm 1. The function takes a target, *atomic*, a compare value, *comp*, and a potential replacement value, *new*. The value of *atomic* is replaced with *new* only if its prior value was *comp*. Whether or not *atomic* is changed `cmpxchg` always returns its prior value. The function is, on Linux, guaranteed to be available on all supporting architectures, whether natively supported by the hardware or not. On x86 architectures, it is implemented as a single inline instruction.

The `cmpxchg` can be used to build a compare-and-swap loop. Intuitively the idea is to take a local copy of the value, modify that local copy as needed, and finally, unless concurrent modifications took place, write the temporary value into the original object. The last part is enforced by using `cmpxchg` to ensure that the initial local copy still matches with the value of the original. Algorithm 2 shows an example where a compare-and-swap is used to atomically apply `func` on an object. `func` can be an arbitrary long non-atomic function because the `cmpxchg` ensures that the value of *atomic*, i.e., *old*, is equal to its initial value *val*. The loop is iterated until the

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

Algorithm 1: `cmpxchg`(*atomic*, *comp*, *new*) sets the value of *atomic* to *new* if, and only if, the prior value of *atomic* was equal to *comp*. It always returns the prior value of *atomic*. On x86 it is implemented as a single inline CPU instruction.

```
1: old ← atomic.value
2: repeat
3:     val ← old
4:     new =func(val)
5:     old ←cmpxchg(atomic, old, new)
6:     if old = val then
7:         return
8:     end if
9: until false
```

Algorithm 2: `cmpxchg_loop`(*func*, *atomic*). This function uses a compare-and-swap loop to atomically perform the non-atomic function `func` on an object *atomic* that supports the `cmpxchg` function. Only the `cmpxchg` function must be atomic.

`cmpxchg` succeeds and thus potentially requires several executions of `func`. However, if the `func` is fast, for instance only a few arithmetic operations, it is likely that the loop will be executed only once. In the Linux kernel, this pattern is often used to implement various functions that lack direct hardware support for atomicity.

In addition to more traditional challenges with concurrency, the compiler and CPU might introduce some more esoteric concurrency issues related to *memory ordering*. Memory ordering refers to the order in which write operations are actually committed to memory, as opposed to where they appear in the source code. Compiler optimizations can remove statements that are deemed unnecessary and change code structure in ways that do not change the end result. Other concurrently executing code might, unfortunately, depend on intermediate states. The CPU itself can cause related problems due to the *relaxed memory consistency models* used by modern CPUs, such as the Intel and ARM CPUs [Int16; ARM]. A relaxed memory consistency model means that the CPU does not always guarantee a specific ordering for any sequence of instructions. This allows the CPU to optimize the order of writes and reads, as long as the resulting values stay the same. This can,

again, produce unexpected results when concurrently running threads rely on the same data [AG96].

Consider the code snippets in Listing 1. For a reader familiar with concurrency issues it should be evident that the output of the `printf` can be either 3, 12 or 30, depending on which CPU executes first. What might be surprising is that the result could also be 21; this could happen because the CPU optimized and swapped the ordering of the read and write instructions. This re-ordering would not be a problem in serialized execution environments but does necessitate further protections when concurrent execution is possible. The solution for this is *memory barriers*, i.e., instructions that ensure that writes or reads are all visible before the barrier. Because memory ordering is architecture dependent, related Linux APIs (e.g., atomic types) provide common guarantees which are then enforced by the architecture-specific implementations.

These specific intricacies of reference counter implementations will be further discussed where pertinent to our work. It is worth reiterating that while many of the discussed concurrency problems could be easily remedied with common high-level constructs, such as locks, mutexes, and semaphores, low-level methods allow much greater flexibility in implementing efficient and highly specific solutions. However, with this great power comes great responsibility, and low-level solutions can indeed be more error-prone.

2.1.2 Reference Counters

A reference counter is, at its core, an integer tracking the number of references to a specific object. The purpose is to manage the lifetime of the reference object, i.e., to ensure memory is not freed while in use and to free memory as soon as it is no longer used [Col60]. This is typically a problem for shared objects; when acquiring a shared object, often via an object-specific `get` function, the object might or might not be allocated. Similarly, when a code path stops using the shared object, it must ensure that either the object is in use elsewhere, or that it is freed. Again, this is typically done in an object-specific function, often with a name such as `put`. Readers familiar with object-oriented programming probably recognize that these are similar to constructor and destructor functions. Algorithm 3 shows a simplified `get` function, which is called when an object is taken into use. The `get` function checks the counter, allocates the object if needed, and updates the counter if not.

```
1              /* global shared definitions */
2              int var_A = 1;
3              int var_B = 2;
4
5  /* Split execution into two threads on differenct CPUs */
6  /*    CPU 1    */                          /*     CPU 2      */
7  var_A = 10;                                int x = var_A;
8  var_B = 20;                                int y = var_B;
9                                             printf("%d", x+y);
10
11     /* Possible outcomes:                                    */
12     /*    x:          1          10          10          1    */
13     /*    y:          2          2           20          20   */
14     /*    output:     3          12          30          21   */
```

Listing 1: Example code to illustrate issues resulting from relaxed memory consistency models. The three first outcomes are intuitively clear, but it is surprising that execution on some CPUs might also result in the fourth option of 21. Code is loosely based on [Tor, Linux v4.9, `Documentation/memory-barriers.txt`].

When an object is no longer used, a `put` function, similar to the one in Algorithm 4, is called. This function decrements the object's reference counter, and if it reaches zero, also deallocates the object. Note that both Algorithm 3 and 4 are simplified, particularly in that they ignore concurrency issues. A practical thread-safe implementation must, for instance, guarantee that memory allocation and freeing is performed only once, and that reference counter updates are thread-safe.

Reference counting is not typically needed in memory-safe, where garbage collection or other mechanisms automatically call associated constructors and destructors without any direct input from the programmer. In the context of

1: **if** $obj.refcount > 0$ **then**
2: $\quad obj.refcount = obj.refcount + 1$
3: $\quad$ **return** obj;
4: **end if**
5: $obj =$`allocate_and_init_obj`()
6: $obj.refcount = 1$
7: **return** obj;

Algorithm 3: A simplified `get` function, called when a user wants to get a shared object obj. The function checks whether the associated reference counter $obj.refcount$ is zero, and if so it allocates the object. In other cases the reference counter is non-zero, and therefore assumed to be allocated and in use.

```
1: obj.refcount = obj.refcount − 1
2: if obj.refcount = 0 then
3:   free_and_cleanup_obj(obj)
4: end if
5: return
```

Algorithm 4: A simplified `put` function, called when a user stops using a shared object *obj*. The function decrements the associated reference counter and if it reaches zero, performs necessary cleanup and frees associated memory.

the Linux kernel, implemented in C, such mechanisms are simply unavailable, thus justifying the heavy reliance on reference counters. Reference counters are used to manage many different objects in the kernel, some of which are apparent; specific hardware resources must, for instance, be initialized and shut down only once, despite potentially being shared extensively between different systems. Other use cases are less obvious; consider something as specific as an in-kernel data structure containing the mandatory access control (MAC) group membership for running processes. The data-structure is shared with descendant processes to avoid needless duplication and ensure consistency, hence requiring a reference counter to keep track of its use.

Reference counters must ensure not only consistency but also the timeliness of the counters and related operations. Using atomic integers ensures consistency of the counters, but does not guarantee that object construction and destruction is thread-safe. Consider for instance the code in Listing 2. Although individual operations are atomic, i.e., the atomic variable itself remains correctly updated, the execution order could otherwise be such that the object gets freed by Thread 1 before it is used by Thread 2. Value checks and corresponding value changes must therefore typically be executed atomically. A typical solution for this timeliness problem is to use atomic functions that combine the value check and modification into a single atomic call. For example, increments can be executed with an atomic function that prevents and reports increment on zero.

As mentioned, the reference counter operations, including object allocations and releases, are typically encapsulated in object specific `get` and `put` functions. These functions then, transparently to the caller, manage both the object lifetime and its reference counter. In practice, implementations prefer using atomic operations for performance reasons and only employ heavy-weight locking when strictly necessary. Either way, these are typically issues related to specific objects and their implementation, with one major exception: Reference counting schemes rely on the programmer manually calling the `put` and `get` functions. Not only must these functions be called, but users must ensure that `put` is called upon release exactly as many times

```
1  shared_object = { .refcount = 1, .data = ... };
2
3  /*        Thread 1        */   /*          Thread 2          */
4  a = get_shared_object();       b = get_shared_object();
5  atomic_dec(a->refcount);       if (!is_zero(a->refcount)) {
6  if (is_zero(a->refcount)           atomic_inc(a->refcount);
7      free(a);                       use_shared_object(b);
8
9  /*                possible execution order                  */
10 /*        Thread 1                          Thread 2          */
11 a = get_shared_object();
12                                b = get_shared_object();
13                                if (!is_zero(a->refcount)) {
14 atomic_dec(a->refcount);
15 if (is_zero(a->refcount))
16     free(a);
17                                    atomic_inc(a->refcount);
18                                    use_shared_object(b);
```

Listing 2: Example of a unsafe reference counter use. Code such as this would introduce a potential race condition causing already freed memory to be used in Thread 2, and is thus a clear programming error.

as `get`. The `put`, in particular, is problematic because it must be called in all instances, such as potential error handlers or other abnormal code paths.

A final issue of concern is compiler and CPU optimizations, discussed in Section 2.1.1. In particular, reference counter implementations must use memory barriers to ensures that reference counters, when needed, operate on timely data. These issues are partially avoided by using atomic types that provide full memory ordering. However, users must still be careful on object destruction and creation. For instance, an object must be initialized only once and must not be exposed, i.e., shared, before it is fully initialized.

2.1.3 Reference Counters in Linux

The use of reference counters is quite widespread in the Linux kernel and they are used in all parts of the kernel. The kernel, by its very nature, needs to manage a huge number of shared resources, everything from files, user-data, memory allocations to hardware resources. Even in situations when not strictly necessary, it is often more efficient to share data rather than duplicate and recreate it. The specific performance and thread-safety requirements for different shared objects are also quite varied, further diversifying the reference counter landscape. We also found several cases with an unclear distinction between reference counters and statistical counters.

Typical Linux kernel reference counters follow the common `get+put` pattern, where the counter, memory allocations, and memory releases are hidden in the `get` and `put` functions. As an example, consider the MAC

groups of a process. The data structure for these groups is stored in the `struct group_info` data structure and is managed by the Linux credential management system. The data is shared between descendant processes and therefore uses a reference counter to ensure safe removal after all processes are dead. The `put_group_info` preprocessor macro, shown in Listing 3, encapsulates the required logic, allowing users to simply call it when discarding the local reference to the data. A process can then just call `put_group_info` upon termination, it need not concern itself with counter values, nor even if the data is freed or not.

The diverse use-cases for reference counters is evident when, for instance, contrasting the used `group_info` data structure with `struct sock` socket data structure. The `struct sock` is part of the networking subsystem and is defined in `include/net.sock`. It is used to store information on a network socket and uses a reference counter to track its uses. In contrast to the sporadically used `group_info`, the network sockets need to potentially serve thousands of requests per second, thus imposing very different performance requirements.

Linux kernel reference counting mechanisms can be divided into different categories depending on atomicity guarantees, value-check schemes, and memory barriers [McK07]. The plain counter, essentially a bare `int` or `long` integer, is appropriate in a situation where concurrent access is limited by other means. In such trivial cases reference counter related concurrency issues are typically non-existent and can thus be ignored. We have observed that Linux kernel reference counting schemes rely on atomic integer types to provide atomicity, timely value-checks, and required memory barriers. To be specific, most in-kernel reference counting schemes directly use one of the atomic types, or in, some cases, the `kref` type.

2.1.3.1 Atomic Types as Reference Counters

The `atomic_t` type is an integer data-type with an atomic API for accessing and modifying it. The type is frequently used in reference counting schemes and therefore provides several reference counter specific accessors, including atomic functions that combine value checks with modifications.

```
1 #define put_group_info(group_info)                          \
2 do {                                                        \
3     if (atomic_dec_and_test(&(group_info)->usage))          \
4         groups_free(group_info);                            \
5 } while (0)
```

Listing 3: Example of a reference counter put function that decrements and potentially frees related resource [Tor, Linux v4.8, `include/linux/cred.h`].

For example, the `atomic_dec_and_test` function, which atomically decrements the variable and returns true if the resulting value is zero, or the `atomic_inc_not_zero`, which atomically increments a non-zero value and returns true when the resulting value is non-zero. Listing 4 shows a typical use of these functions and essentially provides a concrete implementation for the `put` and `get` (Algorithms 3 and 4).

Due to performance requirements, different architecture support, and concurrency issues, the atomic types are quite challenging in their implementations. To further complicate matters the atomic integers constitute a suite of types — `atomic_t`, `atomic_long_t`, `atomic64_t`, and `local_t` — all with architecture-dependent differences. The `atomic_t` type is defined as a C integer, as shown in Listing 5, and thus is of architecture-specific size. `atomic_long_t` is similarly defined as an architecture-specific C `long`, thus typically being either 32bit or 64 bits of size. The `atomic_long_t` type was introduced to provide a way to use 64bit atomic integers on supporting platforms while cleanly falling back to 32bit integers for platforms that do not [Lam05]. The `atomic64_t`, on the other hand, provides an atomic integer guaranteed to be of 64bits size. Finally, `local_t` provides a single-CPU atomic integer, i.e., one that guarantees atomicity only when used by a single CPU.

What makes these types confusing is that they are cross-dependent. As shown in Listing 6, `atomic_long_t` is on architectures with a 64bit `long` defined as an `atomic64_t`, otherwise as an `atomic_t`. This way of defining the atomic types provides the described type sizes with minimal code duplication. From a performance and implementation perspective, architectures can implement only `atomic_t` and `atomic64_t`, and still guarantee that both `atomic_t` and `atomic_long_t` use native types. In contrast, `atomic64_t` guarantees the size but might default to a generic, non-architecture specific, implementation and thus provide lower performance.

Linux provides generic, i.e., not architecture-specific, implementations for most atomic API calls. These implementations can, and typically are, replaced by architecture-specific implementations. Because the generic implementation must rely on commonly available functionality, they are often slower than their architecture-specific counterparts. Consider for instance the generic implementation for `atomic_add`, shown in Listing 7. The function uses the compare-and-swap loop, described in Section 2.1.1, which requires only an atomic `cmpxchg` function. Architecture-specific implementation can, despite the elegance of the compare-and-swap loop, typically offer a faster architecture-specific implementation. The x86, for instance, implements the same `atomic_add` function with a single instruction, as shown in Listing 8. The compare-and-swap loop, by contrast, requires at least four instructions to complete, as it must separately read the original value, calculate the updated value, perform the compare-and-swap, and finally conditionally break out of the loop.

```
1  obj *shared;
2
3  void release_shared() {
4      if (atomic_dec_and_test(obj->refcnt))
5          free(shared);
6  }
7
8  obj *acquire_shared() {
9      if (atomic_add_unless(obj->refcnt, 1, 0))
10         return shared;
11     /* else allocate new one and return it */
12     ...
```

Listing 4: Example usage of `atomic_dec_and_test` and `atomic_add_unless`. The combined atomic value checks and writes of `atomic_t` allow safe implementation of reference values checks.

```
1  typedef struct {
2      int counter;
3  } atomic_t;
```

Listing 5: Definition of `atomic_t` [Tor, Linux v4.9, `include/linux/types.h`].

```
1  #if BITS_PER_LONG == 64
2
3  typedef atomic64_t atomic_long_t;
4
5  #define ATOMIC_LONG_INIT(i)      ATOMIC64_INIT(i)
6  #define ATOMIC_LONG_PFX(x)       atomic64 ## x
7
8  #else
9
10 typedef atomic_t atomic_long_t;
11
12 #define ATOMIC_LONG_INIT(i)      ATOMIC_INIT(i)
13 #define ATOMIC_LONG_PFX(x)       atomic ## x
14
15 #endif
```

Listing 6: Definition of `atomic_t` [Tor, Linux v4.9, `include/asm-generic/atomic-long.h`].

As an astute reader might have realized most of the various cross-dependencies — convoluted inheritance between `atomic_t`, `atomic_long_t` and `atomic64_t` and generic implementations with optional architecture-specific implementations — cannot rely on language-provided object-oriented mechanisms; the Linux kernel is implemented in C. This means that all of this is implemented using pre-processor macros controlled by various options and configuration flags. The declarations for the four types and their APIs also span several different header files, some mixing definitions for different types. The actual source code, as is typical in the Linux kernel, also heavily employs pre-processor macros to reduce code duplication; for example, the verbatim implementing for the generic `atomic_add` is shown in Listing 9.

The main atomic functions guarantee full memory ordering, i.e., all prior memory operations are committed to memory before the function call and subsequent operations after the call. The atomic API also provides functions with less stringent ordering. The relaxed variants give no guarantees on whether individual memory operations on the atomic variable are perceived as having happened before or after calling the atomic function. The relaxed variants are not used for reference counting in the Linux kernel. For more discussion on memory ordering and memory barriers see Section 2.1.1.

The Linux atomic types are convenient to use for reference counting because they sidestep any concurrency issues related to optimizations and memory ordering. They nonetheless leave ample room for incorrect and unsafe implementation of reference counting schemes. The seemingly simple acquire and release pattern, shown in Listing 4, is seldom as cleanly separated from other program logic. For example, in some cases frequently used objects might be recycled instead of repeatedly freed and reallocated. In Section 2.1.4 we will explore other subtleties that have historically caused reference counting errors.

```
1  static inline void atomic_add(int i, atomic_t *v)
2  {
3      int c, old;
4
5      c = v->counter;
6      while ((old = cmpxchg(&v->counter, c, c + i)) != c)
7          c = old;
8  }
```

Listing 7: The `atomic_add` implementation. While written concisely the function employs the already familiar compare-and-swap loop pattern which requires only the `cmpxchg` call to be atomic. This listing shows pre-processor macros expanded for readability, the actual source code is shown in Listing 9.

```
1  static __always_inline void atomic_add(int i, atomic_t *v)
2  {
3      asm volatile(LOCK_PREFIX "addl %1,%0"
4           : "+m" (v->counter)
5           : "ir" (i));
6  }
```

Listing 8: The x86 implementation of `atomic_t`, defined in inline assembly [Tor, Linux v4.9, `arch/x86/include/asm/atomic.h`].

```
1  #define ATOMIC_OP_RETURN(op, c_op)                                    \
2  static inline int                                                     \
3  atomic_##op##_return(int i, atomic_t *v)                              \
4  {                                                                     \
5      int c, old;                                                       \
6                                                                        \
7      c = v->counter;                                                   \
8      while ((old = cmpxchg(&v->counter, c, c c_op i))                  \
9             ) != c)                                                    \
10         c = old;                                                      \
11                                                                       \
12     return c c_op i;                                                  \
13 }
14 /* cut to line 120 */
15 ATOMIC_OP_RETURN(add, +)
16 /* cut to Line 186 */
17 static inline void atomic_add(int i, atomic_t *v)
18 {
19     atomic_add_return(i, v);
20 }
```

Listing 9: The source code implementing the generic `atomic_add` [Tor, Linux v4.9, `include/asm-generic/atomic.h`] is a representative example of code organization in Linux, and the atomic subsystem specifically.

2.1.3.2 The kref Reference Counter Object

To avoid error-prone reference counting schemes, `kref` — a dedicated reference counter type — was introduced in 2004 [Kro04]. The type replaces direct use of `atomic_t` reference counters, although it is itself internally implemented using `atomic_t`, as shown in Listing 10. By using `atomic_t`, the implementation provides efficient and secure functionality with minimal need for architecture-specific implementation details in `kref` itself. Thus, in contrast to `atomic_t`, `kref` provides a maintainable implementation with a tightly focused API.

The `kref` API provides only `put` and `get` functions, not functions for setting, adding or subtracting arbitrary values. Listing 11 shows the definition of `kref_put`. The API enforces safe value checks and object releases by requiring a pointer to a release function, i.e., a destructor. The `release` function is called when the reference counter reaches zero. By combining the release and decrement, `kref` avoids implementation pitfalls associated with `atomic_t` schemes, e.g., faulty schemes such as in Listing 2 (in Section 2.1.2).

The increment functions — `kref_get` and `kref_get_unless_zero` — are shown in Listings 12 and 13. Both of these function also promote safe use, the former by issuing a kernel `WARN` if called on a `kref` with a value less than two. The latter by refusing to increment the counter if it is zero, and also issuing a compile-time warning if the return value of the function is not checked. The compile-time warning is triggered by the `__must_check` attribute, which is defined as the GCC specific `__attribute__((warn_unused_result))` attribute. In a typical case, either `kref_get` or `kref_get_unless_zero` is used to increment of an object before using it. Note that only the latter notifies the caller of a failed attempt, hence the runtime warnings for the former.

`kref` provides clear semantics coupled with safe and, in comparison to bare atomic counters, relatively high-level API. Unfortunately, the API is also restrictive, which limits its deployability. This trade-off is understandable as it prevents unsafe and easily error-prone implementations. It should be stressed that since `kref` is built upon `atomic_t`, it suffers from some of the same underlying implementation problems, in particular, it exhibits the same integer overflow behavior our work is looking to address. The security implications are discussed further in Section 2.1.4.

```
1 struct kref {
2     atomic_t refcount;
3 }
```

Listing 10: Definition of `kref` [Tor, Linux v4.9, `include/linux/kref.h`].

```
1  static inline int kref_put(
2          struct kref *kref,
3          void (*release)(struct kref *kref))
4  {
5      if (atomic_dec_and_test(&kref->refcount)) {
6          release(kref);
7          return 1;
8      }
9      return 0;
10 }
```

Listing 11: Simplified definition of `kref_put` [Tor, Linux v4.10, `include/linux/kref.h`]. The second argument, `release`, is a pointer to a object specific destructor function.

```
1  static inline void kref_get(struct kref *kref)
2  {
3      WARN_ON_ONCE(atomic_inc_return(&kref->refcount) < 2);
4  }
```

Listing 12: Definition of `kref_get` [Tor, Linux v4.10, `include/linux/kref.h`]. Note the `WARN_ON` that causes a warning if the `kref` is close to being freed. The warning indicates that calling code should instead use `kref_get_unless_zero` to detect failed increments.

```
1  static inline
2  int __must_check kref_get_unless_zero(struct kref *kref)
3  {
4      return atomic_add_unless(&kref->refcount, 1, 0);
5  }
```

Listing 13: Definition of `kref_get_unless_zero` [Tor, Linux v4.10, `include/linux/kref.h`], implemented using `atomic_add_unless` for conditional increments. Callers are encourage to actually check the return value by enabling compile-time warnings for omitted checks, using the `__must_check` compiler attribute.

2.1.4 Security Implications

A reference counter is essentially a simple integer, and thus, poses little direct threat. They are, however, used to govern the lifetime of objects, which in C are implemented via direct memory pointers. Reference countering errors are therefore a source of temporal memory errors. Two typical errors are *use-after-free* and *double-free*. These can be the result of a programming error, i.e., some piece of code retains and uses a pointer after it has been freed, or some code incorrectly frees the pointer after it has already been freed. Both types of errors corrupt memory management structures and can be used to achieve arbitrary memory read and writes [SPW13].

The obvious reference counter errors are incorrect counter updates. Conceptually there are three types of attacks that unbalance a reference counter. First, by causing a reference counter to be too high, an attacker can cause a memory leak by preventing the reference counter from reaching zero. Second, by causing the counter to be too low, an attacker can cause the reference counter to prematurely reach zero, thus causing a use-after-free error. Third, by managing to modify reference counters with a value of zero an attacker can cause either use-after-free or double-free errors. However, there is a fourth option. Because the counter is an integer, an attacker can cause the value to grow high enough to overflow and thereby reach zero the other way around, again causing a use-after-free error.

We are specifically focusing on overflows for several reasons. First, underflow, i.e., missing increments or extra decrements, are typically easy to catch in testing. Even one incorrect iteration causes observable effects by triggering a use-after-free error. Second, there is no immediate remedy for an underflow. When an underflow or below-zero decrement is detected, the state of the associated object is unknown. It might be in use or it might be freed, or both. In addition, the underflow does not necessary indicate an error at the call site, i.e., it could be an unrelated code path that incorrectly decremented the counter to zero. Overflows, by contrast, require an excessive number of faulty iterations, i.e., up to `INT_MAX` or `LONG_MAX`, before causing side-effects. Also, overflows can be remedied; when the overflow happens it is clear that the object is still both in use and in a consistent state, the counter can thus be *saturated*: perpetually leaving it at its maximum value, which ensures that it will never be freed, hence never subject to use-after-free. Because the object will not be freed, this turns a potential use-after-free into a memory-leak.

While it is theoretically possible to overflow a counter without underlying programming errors, memory restrictions make it infeasible to hold enough — more than `INT_MAX` or `LONG_MAX` — references to cause an overflow. Therefore, the attacker needs to find a faulty code path that releases a reference without decrementing the counter. The faulty release allows the attacker to increment the counter by repeatedly acquiring and releasing the reference.

As an example, CVE-2014-2851 [cve14a] is a vulnerability that leverages two reference counter errors in the IPv4 implementation on Linux v3.14. The exploit uses an erroneous code path that allows object references to be dropped without decrementing the counter. The referenced object, in this case, is the `struct group_info` object, which holds the MAC group information. The specific vulnerability is in the `ping_init_sock` function in [Tor, Linux v3.14, `net/ipv4/ping.c`]. The object is typically acquired by invoking the `get_current_groups` macro, shown in Listing 14. What is not immediately apparent is that the `get_group_info` function (Listing 15) increments the reference counter for `struct group_info`, and therefore requires a corresponding `put_group_info` call. It was this `put_group_info` that was missing and caused the CVE.

The CVE above is an archetypical example of why `kref` and other efforts have tried to standardize reference counting schemes, or at the least make errors less subtle and thus easier to find via audit or other techniques. The reference counter overflow substantially increases the exploitability of missing-decrement errors by promoting them from memory leaks to potential use-after-free vulnerabilities. This leaves us with two major security concerns: first, the maintainability and usability issues that make reference counting schemes error-prone, and second, the reference counter overflows that increase the exploitability and severity of resulting vulnerabilities.

Vulnerabilities due to reference counter overflow are not limited to the one described here. Other recent cases are CVE-2016-0728 [cve16a], where an attacker can abuse error conditions in `security/keys/process_keys.c` to cause an overflow, use-after-free and finally privilege escalation. And, CVE-2016-4558 [cve16b] in which a reference counter in the Berkeley packet filter subsystem could be caused to overflow and cause a use-after-free.

2.1.5 PaX/Grsecurity and `PAX_REFCOUNT`

The `PaX/Grsecurity` patch is a monolithic patch that adds various security features to the Linux kernel. It is maintained by Open Source Security inc., whom recently removed public access to their patches [Pax17]. Therefore,

```
1 #define get_current_groups()                                \
2 ({                                                          \
3     struct group_info *__groups;                            \
4     const struct cred *__cred;                              \
5     __cred = current_cred();                                \
6     __groups = get_group_info(__cred->group_info);          \
7     __groups;                                               \
8 })
```

Listing 14: Definition of the `get_current_groups` macro [Tor, Linux v3.14, `include/linux/cred.h`].

```
1 static inline
2 struct group_info *get_group_info(struct group_info *gi)
3 {
4     atomic_inc(&gi->usage);
5     return gi;
6 }
```

Listing 15: Definition of the `get_group_info` function [Tor, Linux v3.14, `include/linux/cred.h`].

the discussion below is based on patches dating to late 2016 and the available `PAX_REFCOUNT` documentation [Bra15]. Due to recent mainstream kernel changes, this description might not reflect the current state of this feature; unfortunately, we cannot confirm this due to the lack of public patches.

Of interest here is the `PAX_REFCOUNT` feature that prevents reference counter overflows. It uses the saturation mechanism and provides architecture-specific implementations of the feature. On x86 architectures the CPU overflow flag is used to detect an overflow and revert the increment. Listing 16 shows the x86 implementation of the overflow detection logic in the addition function. On other supported platforms — ARM, MIPS, PowerPC, and SPARC — the increment is prevented before it is allowed to overflow.

An astute reader might notice a potential problem in the x86 saturation implementation, namely a race condition. The problem is that while the addition itself is properly atomic, the overflow check and subsequent value revert is not. Unless the counter is saturated or overflows, this does not matter. However, if multiple threads simultaneously increment the counter before either revert, then the overflow and revert happens only on one of the threads. `PaX/Grsecurity` recognizes the race condition but to quote their documentation: "*PaX considered this to be such an impractical case that it never implemented this additional logic*" [Bra15].

```
1 asm volatile(LOCK_PREFIX "addq %1,%0\n"
2
3 #ifdef CONFIG_PAX_REFCOUNT
4         "jno 0f\n"
5         LOCK_PREFIX "subq %1,%0\n"
6         "int $4\n0:\n"
7         _ASM_EXTABLE(0b, 0b)
8 #endif
```

Listing 16: The `PAX_REFCOUNT` detection logic implemented on x86 [Bra15], located in `arch/x86/include/asm/atomic64_64h`. The individual operations are atomic but the compound statement statement is not.

2.2 Problem Statement

The motivation for this work is to prevent use-after-free vulnerabilities caused by reference counter overflows in the Linux kernel. In addition to preventing the specific overflow vulnerability, the solution must also minimize the impact on the usability and maintainability. Any changes to current reference counting schemes must also incur minimal performance cost, which is of particular importance when considering networking and other performance-sensitive systems. Our requirements are thus:

1. **Protected reference counters must not be allowed to overflow and thereby cause use-after-free errors.**
2. **Reference counter protections should not cause undue performance degradation.**
3. **The protection mechanism should accommodate existing reference counting schemes.**
4. **The implementation must be maintainable.**

The main goal of our work is to prevent the overflow itself (Requirement 1) while introducing negligible performance cost (Requirement 2). However, the solution must not increase the likelihood of other problems via usability or maintainability issues. To achieve this, the API should be self-documenting, user-friendly, and minimal (Requirement 3). Moreover, upstream changes to the mainline Linux kernel must be reasonably maintainable and conform to Linux development standard (Requirement 4).

2.3 Approach I: Hardening the Underlying Type

Our initial proposal, `HARDENED_ATOMIC`, was based on existing KSPP plans, with initial work started in 2015 by David Windsor [Edg15]. This solution is based on the `PAX_REFCOUNT` feature by `PaX/Grsecurity` (see Section 2.1.5). The idea is to protect the underlying `atomic_t` types and thereby cover all reference counting schemes built around the `atomic_t`. While this work is built on a preliminary port of `PAX_REFCOUNT`, the work nonetheless was substantial. The available `PAX_REFCOUNT` source code touched an older kernel version and `PaX/Grsecurity` in general targets a restricted subset of available mainline architectures and configurations. Therefore, our solution had to update the available patch for the most recent mainline Linux kernel version. Moreover, our patch must ensure compatibility with all possible Linux architectures and configurations.

`HARDENED_ATOMIC` prevents overflow by saturating the counter upon overflow (Section 2.1.4). This prevents overflows, but introduces a problem for `atomic_t` instances that rely on normal integer behavior. Statistical or sequential counters could in some cases even rely on specific overflow behavior[2]. This work, thus consisted of not only protecting the atomics, but also of providing a workaround and ensuring that it is used wherever needed. First, we provided the implementations for the hardened atomic types, i.e., `atomic_t`, `atomic_long_t`, `atomic64_t`, and `local_t`. Second, we ensured that alternate non-protected types, i.e., `atomic_wrap_t`, `atomic_long_wrap_t`, `atomic64_wrap_t`, and `local_wrap_t`, are used where required.

2.3.1 Implementation

The implementation of `HARDENED_ATOMIC` is conceptually simple: harden the atomic types and apply workarounds for cases where overflow is needed. New features to the mainline kernel must, however, typically be split into smaller individual patches, each of which must be applicable one-by-one without breaking intermediate compilations. New features should be configurable and allow easy integration with potential architecture-specific implementations. The `PAX_REFCOUNT` feature we built on, by contrast, is part of a monolithic patch-set that explicitly limits architecture support. The situation is further complicated due to the already complex implementation of the atomic types (Section 2.1.3). To provide an overarching protection and avoid inconsistency, we provide hardened versions for all the four basic atomic types, shown in Table 1.

To ensure the consistency across different architectures, both those implementing `HARDENED_ATOMIC` and those not, we needed to provide configuration options for *`Kbuild`*, the Linux kernel build system. The generic

[2] `Kbuild` uses compilation flags to specify overflow behavior, which in the C standard is undefined.

`atomic_t`	atomic integer of `int` size
`atomic_long_t`	atomic integer of `long` size
`atomic64_t`	64-bit atomic integer
`local_t`	`long` sized integer with single-CPU atomic operations

Table 1: The atomic types protected by `HARDENED_ATOMIC`

(i.e., architecture-independent) implementation is presented in Section 2.3.1.1 along with necessary `Kbuild` configuration. We provide only x86 support, which is presented in Section 2.3.1.2. The workaround for overflowing atomic variables is presented in Section 2.3.1.3. Our added tests are presented in Section 2.3.1.4.

2.3.1.1 Generic Implementation

To ensure that `HARDENED_ATOMIC` can be both completely disabled (e.g., to accommodate untested third-party code) and to differentiate between supported architectures we added appropriate `Kbuild` configuration variables. The `Kbuild` configuration variables are defined in various Kconfig files and specify the possible configurations when compiling the Linux kernel. The `HARDENED_ATOMIC` configuration is added to the `security/Kconfig` file and consists of the `HAVE_ARCH_HARDENED_ATOMIC` and `CONFIG_HARDENED_ATOMIC` options (Listing 17). The former is not user configurable but set by architectures that implement `HARDENED_ATOMIC`. The latter is user-configurable and allows the whole feature to be disabled or enabled.

While our focus is on x86, our generic implementation must not adversely affect other architectures, regardless of their support for `HARDENED_ATOMIC`. Therefore, it must accommodate a multitude of different configurations, architectures, and combinations thereof. The Linux 4.9 implementations for the types covered by `HARDENED_ATOMIC` are defined across several files (shown in List 1). All of these need to be updated to accommodate any valid configurations and architecture setup.

The workaround types (or, wrapping types) are suffixed with `_wrap`. These `wrap` functions are identical to the non-hardened atomics and must be used whenever an atomic variable should allow overflow. They must also be available whether or not `HARDENED_ATOMIC` is enabled. Our implementation provides `asm-generic` headers that define the wrapping types for kernel builds not using `HARDENED_ATOMIC`. `Kbuild` transparently uses either architecture-specific headers or the `asm-generic` headers depending on availability. `HARDENED_ATOMIC` explicitly requires hardware support; the `asm-generic` headers thus need to accommodate only configurations where `HARDENED_ATOMIC` is unused, i.e., when the non-`wrap` functions are unprotected and identical to the `wrap` functions. These `asm-generic` headers

```
config HAVE_ARCH_HARDENED_ATOMIC
   bool
   help
     The architecture supports CONFIG_HARDENED_ATOMIC
     by providing trapping on atomic_t wraps, with a
     call to hardened_atomic_overflow().

config HARDENED_ATOMIC
   bool "Prevent reference counter overflow in atomic_t"
   depends on HAVE_ARCH_HARDENED_ATOMIC
   depends on !CONFIG_GENERIC_ATOMIC64
   select BUG
   help
     This option catches counter wrapping in atomic_t,
     which can turn refcounting overflow bugs into
     resource consumption bugs instead of exploitable
     use-after-free flaws. This feature has a negligible
     performance impact and therefore recommended to be
     turned on for security reasons.
```

Listing 17: The `security/Kconfig` file specifies the added KBuild configuration options.

`include/linux/types.h`
: Defines various types, including the `atomic_t` and `atomic_long_t`.

`include/asm-generic/atomic.h`
: Defines generic implementation for `atomic_t` functions.

`include/asm-generic/atomic-long.h`
: Defines generic implementations for `atomic_long_t` functions.

`include/asm-generic/atomic64.h`
: Defines generic implementations for `atomic64_t` functions.

`include/asm-generic/local.h`
: Defines both the type `local_t` and its related functions.

`include/linux/atomic.h`
: Defines generic function implementations for all of the atomic types, including function with relaxed memory ordering.

List 1: List of main files related to the implementation of the atomic types and related functions.

```
1  static __always_inline void atomic_add(int i, atomic_t *v)
2  {
3      asm volatile(LOCK_PREFIX "addl %1,%0\n"
4
5  #ifdef CONFIG_HARDENED_ATOMIC
6                  "jno 0f\n"
7                  LOCK_PREFIX "subl %1,%0\n"
8                  "int $4\n0:\n"
9                  _ASM_EXTABLE(0b, 0b)
10 #endif
11
12                 : "+m" (v->counter)
13                 : "ir" (i));
14 }
```

Listing 18: Definition of the `atomic_add`, located in the x86 specific file `arch/x86/include/asm/atomic.h`.

simply define the `wrap` functions as aliases to the atomic functions. The `atomic_t`, `atomic64_wrap_t`, and `local_wrap_t` types have similar fallback definitions for builds with `HARDENED_ATOMIC` disabled, but because `atomic_long_t` is defined via either `atomic_t` or `atomic64_t`, it does not require special attention here.

2.3.1.2 x86 Implementation

x86 specific `Kbuild` configuration is specified in `arch/x86/Kconfig`, which is also where `HAVE_ARCH_HARDENED_ATOMIC` is set to indicate support for this feature. Other architectures would similarly indicate their support by setting the same variable in their corresponding `Kconfig` files. All atomic functions must then, depending on `CONFIG_HARDENED_ATOMIC`, be either protected or not protected. The `wrap` variants must also be provided but because they are identical to the non-protected functions their implementation is trivial. The implementation of `atomic_add` (Listing 18), and indeed most x86 implementations, use the `CONFIG_HARDENED_ATOMIC` preprocessor macro to determine whether to include the overflow protections or not.

The protected `atomic_add` on x86 is inlined and consists of just four assembler instructions and at best executes only two instructions. The first instruction, present in both the hardened and regular definition, performs the addition. The latter three instructions, present only when hardened, check if the CPU overflow flag (`OF`) is set (i.e., whether the preceding instruction caused an integer overflow) and if so, reverts the addition. The `atomic64_t` on 64bit builds is similar to the `atomic_t` implementation, but 32bit builds require some additional work to support the 64bit integers. Since `atomic_long_t` is defined using either `atomic_t` or `atomic64_t`, it does not require any implementation specific work. The single CPU atomic

`local_t` on x86 has a separate implementation and therefore also requires `HARDENED_ATOMIC` definitions. Due to its relaxed concurrency requirements, the implementation does not require a lock prefix but is otherwise similar to the other atomic types. For instance, the only difference between the `local_add` (Listing 19) and `atomic_add` is the lock labels[3].

The availability of the x86 CPU overflow flag improves the efficiency and size of the protection. Unfortunately, it is vulnerable to a race-condition when two or more simultaneous additions overflow the counter. While the additions remain atomic, the subsequent saturation mechanism does not. Only one of the threads would, therefore, encounter the `OF` flag, leaving the others free to work on the now overflown value. Considering single increments, this means that the non-overflowing thread would increment the counter to `MAX`+2 and the other thread revert by decrementing to `MAX`+1. This issue is recognized by `PaX/Grsecurity` (Section 2.1.5) and is discussed further in Section 2.3.3.

2.3.1.3 Wrapping Atomic Types

One major task, beyond the implementation of the `HARDENED_ATOMIC` types, is ensuring that no unintended behavior is introduced by the changed semantics of the atomic types. The specific problem is that several use cases, in fact, allow or expect that the atomic variable is going to overflow, and thus need to be converted to our alternate `_wrap` atomic variant. Although the C standard specifies overflow behavior as undefined, the kernel uses GCC compiler flags to enforce specific behavior on overflow; `-fwrapv` tells the compiler that overflows follow the semantics of two's-complement integers and the `-fno-strict-overflow` disables compiler optimizations that rely

[3]The `asm` lock labels are inherent to CPU instructions and in particular do not incur overheads comparable to locks commonly used in high-level concurrent programming.

```
1  static inline void local_add(long i, local_t *l)
2  {
3      asm volatile(_ASM_ADD "%1,%0\n"
4  #ifdef CONFIG_HARDENED_ATOMIC
5                  "jno 0f\n"
6                  _ASM_SUB "%1,%0\n"
7                  "int $4\n0:\n"
8                  _ASM_EXTABLE(0b, 0b)
9  #endif
10                 : "+m" (l->a.counter)
11                 : "ir" (i));
12 }
```

Listing 19: Definition of the `local_add`, located in the x86 specific file `arch/x86/include/asm/local.h`.

```
1  void lkdtm_OVERFLOW_atomic_add(void)
2  {
3      atomic_set(&atomic_var, INT_MAX);
4
5      pr_info("attempting good " "at omic_add" "\n");
6      atomic_dec(&atomic_var); atomic_add(1, &atomic_var);
7
8      pr_info("attempting bad " "atomic_add" "\n");
9      atomic_add(1, &atomic_var);
10 }
```

Listing 20: Definition of the `lkdtm_OVERFLOW_atomic_add` function that executes the `OVERFLOW_atomic_add` provided test. The definition is located in `drivers/misc/lkdtm_bugs.c`.

on overflows never happening. Moreover, because all supported Linux architectures use two's-complement to represent integers, the kernel can rely on overflows having predictable behavior.

In practice this part of our work was based on the `PaX/Grsecurity` implementations and initial conversions by David Windsor. However, due to intermediate kernel version changes most of these changes were manually inspected and modified to work with our targeted kernel version, `Linux v4.9`. The actual code changes consist of swapping out `atomic_t` and `atomic_long_t` declarations and function calls with the equivalent `wrap` variants. The conversion work is largely trivial because the `wrap` function signatures are equivalent to their non-`wrap` variants; the challenge being mainly to ensure that all required instances and uses are converted.

2.3.1.4 Testing Facilities

Testing facilities for `HARDENED_ATOMIC` are provided via *`ldktm`*, the Linux Kernel Dump Test Module. Testing is valuable because the feature is expected to be implemented for other architectures. The implemented tests use the `ldktm` direct input mechanism that allows tests to be run by writing commands into the `/sys/kernel/debug/provoke-crash/DIRECT` pseudo file. Such tests typically provoke a system crash and can be observed in the kernel log. The `HARDENED_ATOMIC` tests exercise code that results in overflowing the various atomic types. The test definitions rely on preprocessor macros to decrease code duplication. For readability, Listing 20 shows an expanded version of the `lkdtm_OVERFLOW_atomic_add` function.

The `lkdtm` tests do not provide testing of actual reference counting schemes. They only verify the implementation behavior on overflow. Because `HARDENED_ATOMIC` by default does not crash the kernel, the related `ldktm` tests could be used for automated testing. The full `lkdtm` additions test the most common functions for all the four covered atomic types.

2.3.2 Challenges

The related PAX_REFCOUNT feature is separate from the mainline kernel, as such, it need not consider kernel development guidelines, upstream maintainers, nor by PaX/Grsecurity unsupported architectures. In contrast, our work had to consider all of these. From a maintainability standpoint, this means HARDENED_ATOMIC must not interfere with other architectures and should accommodate the addition of new architecture-specific implementations (Section 2.3.2.1). The percpu-refcount is, as the name implies, a reference counter, yet it exhibits some surprising behavior that makes it challenging to protect or convert to atomic_t (Section 2.3.2.2).

2.3.2.1 Generic Definitions and Architecture support

A surprising challenge in the implementation of HARDENED_ATOMIC was the complex cross-dependencies between the atomic types and their numerous different implementations. All of the complexity is managed by preprocessor macros and Kbuild overloading asm-generic header files with architecture-specific files. Even the purely generic types are spread across several files — atomic_t, for instance, has various generic implementations in include/asm/atomic.h, include/asm/atomic-long.h, include/asm/atomic64.h and include/linux/atomic.h — not to mention all the architecture-specific additions. The Kbuild and preprocessor functionality is necessary to provide a flexible API backed up by a myriad of optimized architecture-specific implementations. Unfortunately, HARDENED_ATOMIC introduces massive changes to the implementations and essentially duplicates the whole APIs for the wrapping atomic variants.

To limit the noise introduced into the generic headers, we added completely new wrap headers. Because the wrap functions need separate definitions only when HARDENED_ATOMIC is enabled, and because HARDENED_ATOMIC requires architecture-specific implementations, these generic wrap headers can be kept minimal. Unfortunately, the generic and architecture-specific implementations are not always cleanly separated. For example, include/linux/atomic.h provides some convenience functions that might be used even when architecture-specific atomics are available.

These are not one-off problems; the resulting code base must be maintainable and robust amidst API updates and implementation changes. During our work, we encountered several occasions where generic changes worked on default x86 configurations, but produced errors on other architectures and configurations. To prevent problems like this we needed extensive macro definitions to manage dependencies between different implementations. We also collaborated with a simultaneous work to provide an ARM implementation for HARDENED_ATOMIC.

```
1 #define CPU_COUNT_BIAS  (1LU << (BITS_PER_LONG - 1))
```

Listing 21: Definition of the `PERCPU_CPUNT_BIAS` macro.

2.3.2.2 The `percpu-refcount` counter

Because `HARDENED_ATOMIC` accommodates existing use cases – provided they do not rely on overflow — we can mostly leave reference counters as they are. The `wrap` use cases do not require any further modifications since they are exactly identical to the original atomics. However, one particularly tricky reference counter is the per-CPU reference counter `percpu-refcount`. This type allows for the reference counter to be either in atomic joint mode or in a per-CPU mode where the individual CPUs manage their separate counters. When in per-CPU mode the individual counts do not trigger any resource freeing, nor are they guaranteed to be balanced (i.e., one CPU could have all the decrements, whereas another has all the increments). The problem arises when the joint counter must be checked, and the individual counts must be added together to a single atomic variable. However, because the individual counts can be imbalanced or even negative, they cannot safely be added together without potentially incorrectly reaching zero and triggering a user-after-free.

The `percpu-refcount` implementation prevents this issue by offsetting the true value of the counter during the mode switch. The used offset, `PERCPU_COUNT_BIAS`, is effectively set to a value as far away from zero as possible, i.e., right by the MIN/MAX barrier (Listing 21). The offset is then removed only after all individual values have been added to the underlying atomic variable, i.e., when the type operates again in atomic mode. With `HARDENED_ATOMIC` this will often trigger pointer saturation because the value already starts at `LONG_MAX`. To solve this, and still protect the counter, the `PaX/Grsecurity` patch decreased the offset so that the acceptable values are in the range [1,`LONG_MAX`], thereby decreasing the available range from the initial [1,`LONG_MAX`] $\cup$ [`LONG_MIN`, -1]. Unfortunately, while this prevents the overflow, it does not prevent the counter from prematurely reaching zero. At the time of writing, we have no satisfactory solution to this problem and have simply documented the vulnerable counter type.

2.3.3 The End of Line

The `HARDENED_ATOMIC` approach underwent four Request for Comments (RFC) cycles on the kernel-hardening mailing list, starting with the original RFC in October 2016, and ending with the final one in November 2016 [Res16b; Res16c; Res16d; Res16e]. The initial response was enthusiastic and accompanying work on an ARM implementation was also started within the KSPP community. In hindsight, it was a mistake not to involve the

maintainers of the existing atomic types. As it is, the maintainers were included only in the last RFC v4. Although the problem of reference counter overflows was accepted due to the abundance of related CVEs (Section 2.1.4); `HARDENED_ATOMIC` as a solution received several valid objections:

Usability: The changes to the atomics and the `wrap` APIs have usability problems. First, the `_wrap` names do not make sense for all functions, e.g., wrapping is meaningless for functions such as `cmpxchg`. Second, the changes to the all-purpose atomic types are not indicated by API or other changes. While naming changes could alleviate the former problem, it is not clear how to solve the latter without a complete overhaul of `HARDENED_ATOMIC`.

Race condition: The race condition in the saturation mechanism is problematic, despite `PaX/Grsecurity` deeming it unlikely enough to be ignored. It is highly dubious for an atomic type to exhibit non-atomic properties. Not only could the race-condition cause the saturation to fail, but it would also corrupt, i.e., imbalance, the counter value. If two or more CPUs increment the variable simultaneously, it is possible for the variable to be incremented twice before the saturation mechanism kicks in, at which point only one of the increments would be undone, thus leaving the variable in an overflowed state.

Maintainability: The changes to the atomic implementation and the added `wrap` APIs make an absolute mess of the already complex atomic subsystem. Some of this is due to design choices (e.g., use of preprocessor macros) and is trivial to change. Substantial changes and added complexity are however unavoidable regardless of implementation details.

Deployability: It is not possible to deploy `HARDENED_ATOMIC` piecemeal. Because it modifies all atomic variables, the required `wrap` changes must be completed beforehand. Moreover, the `wrap` changes specifically touch source code not affected by the overflow problem. Technically this is not a problem, but it is problematic for the upstreaming process that typically consists of incremental and focused changes.

`HARDENED_ATOMIC` was ultimately discarded but prompted productive discussions and motivated the eventual `refcount_t` implementation, discussed in Section 2.4. Because this approach was discarded, we have not completed extensive performance measurements. Based on early measurements by David Windsor [Win16] the impact was however negligible, with a 2.8% overhead on dbench and a 0.01% overhead when compiling the kernel.

2.4 Approach II: Providing a New Hardened Type

After `HARDENED_ATOMIC` was discarded, Peter Zijlstra, one of the maintainers of the Linux atomic types, implemented a new reference counter specific type, `refcount_t` [Coo17a; Zij17]. We contributed to this work by providing kernel-wide analysis and conversion of reference counting schemes [Coo17b]. Based on this analysis we also proposed additions to the `refcount_t` API [Res16a]; these extensions ensure that `refcount_t` is usable as a replacement for prior `atomic_t` reference counters. The conversion efforts consist of 233 patches, of which 125 are already merged into the mainline kernel, and the remaining on the way.

As opposed to `HARDENED_ATOMIC`, `refcount_t` has the benefits of being self-contained and possible to apply piecemeal. Because the API focuses on reference counting, it is minimal. The specific functions can also prevent functionality that would be needed in general-purpose types, but in reference counting indicates programming errors (e.g., increment on zero). `refcount_t` uses the same saturation mechanics as `HARDENED_ATOMIC`. The generic implementation is also sound from a technical standpoint and avoids the race-condition in `HARDENED_ATOMIC` (Section 2.3.3). In Section 2.4.1 we present the evaluation from the initial `refcount_t` API to our extended API extensions, including implementations and other ongoing reference counter work. Section 2.4.2 provides insights on the challenges we faced. Finally, in Section 2.5 we evaluate the solution from a usability, security and performance perspective.

2.4.1 Implementation

The complete `refcount_t` API was submitted by Peter Zijlstra and landed in Linux v4.12 [Zij17]. The API is reference counter specific, which avoids confusion with general-purpose types (like the atomic types) and allows several distinguishing features:

- **It is saturated instead of overflown**, i.e., a value change that would overflow instead sets the counter at its maximum value and a counter at its maximum value is not modified.
- **It emits runtime warnings on saturation**, thus ensuring that such otherwise potentially unnoticed situations are logged and remedied.
- **It refuses to increment on zero**, i.e., none of the functions modify a counter that is zero.
- **Its return value indicates a non-zero value, not whether it was changed**, allowing the use of an object with a saturated counter.

```
1  typedef struct refcount_struct {
2      atomic_t refs;
3  } refcount_t;
```

Listing 22: The definition of the `refcount_t` type `atomic_t` [Tor, Linux v4.12, `include/linux/refcount_t.h`].

- **It causes compile-time warnings if return values are not checked**, thus promoting either safe use or a conscious decision to use the `void` variant of the functions.

The type itself is defined, as shown in Listing 22, as a `struct` encompassing an `atomic_t` counter. In contrast to `atomic_t`, the `refcount_t` is unsigned. This design decision also simplifies to reasoning and implementation of reference counting schemes. For instance, `refcount_t` does not allow counterintuitive additions with negative values, which are typical for many `atomic_t` reference counting schemes. Another subtle difference is the memory ordering of `refcount_t`. All `atomic_t` reference counting schemes employ full memory ordering; `refcount_t`, in contrast, does not. The incrementing `refcount_t` functions provide no memory ordering (see Section 2.1.1), whereas the decrementing ones provide release ordering. The memory ordering change is typically unnoticed for `refcount_t` users but gives some leeway to architecture-specific implementations and CPU optimizations.

2.4.1.1 Initial API

The initial `refcount_t` provided a focused API and included a generic (i.e., not architecture-specific) implementation built on top the `atomic_t`. This initial API was focused on correct reference counter use, and as such provided only a minimal API, shown in List 2. The API is intended to enforce safe reference counting schemes, and in particular, allows only single decrements and increments.

Basic use cases (e.g., Listing 23) are straightforward to convert from

```
void refcount_set(refcount_t, unsigned int)
unsigned int refcount_read(refcount_t)
void refcount_inc(refcount_t)
__must_check bool refcount_inc_not_zero(refcount_t)
__must_check bool refcount_dec_and_test(refcount_t)
__must_check bool refcount_dec_and_mutex_lock(
                                    refcount_t, struct mutex *)
__must_check bool refcount_dec_and_lock(refcount_t, spinlock_ *)
```

List 2: Initial bare `refcount_t` API.

```
1  /* if (!atomic_inc_not_zero(obj->atomic, 1, 0)) { */
2  if (!refcount_inc_not_zero(obj->refcount)) {
3      /* Counter zero, object freed! */
4      /* Create new object instead:  */
5      obj = allocate_obj();
6      /* allocate_obj initializes refcount to 1 */
7  }
8
9  use_obj(obj);
10
11 /* if (atomic_dec_and_test(obj->atomic)) { */
12 if (refcount_dec_and_test(obj->refcount)) {
13     /* Counter reached zero!  */
14     /* Let's free the object: */
15     free_obj(obj);
16 }
```

Listing 23: Example of `refcount_t` use, with the analogous `atomic_t` implementation commented out.

`atomic_t` due to the corresponding calls. However, there are some caveats even in trivial cases. For instance, some reference counters are initialized to zero and then incremented in the object constructor, whereas `refcount_t` must be initialized to one without the increment. Alternatively, `refcount_set` can be used to change a zero reference counter after its initialization. While this is not a challenging modification, it demonstrates a problem with automated conversion from `atomic_t` to `refcount_t`.

The API implementation is found in `lib/refcount.c` and heavily utilizes the compare-and-swap loop, discussed in Section 2.1.1. As an example, consider the `refcount_dec_and_test` implementation shown in Listing 24. The `atomic_cmpxchg_release` function at line 17 ensures atomicity, the return at line 7 ensures saturated values remain unchanged, and the return at line 12 prevents value changes from zero. The security implications will be further examined in Section 2.5.2.

As discussed in Section 2.1.4 reference counter underflows cannot be cleanly remedied; `refcount_t` therefore returns `false` to indicate that the object is not safe to free and thus prevents additional double-free or use-after-free errors. An underflow nonetheless means that use-after-frees have likely already happened and `refcount_t` therefore also emits runtime warnings to indicate this.

2.4.1.2 Deploying `refcount_t`

A major part of our contribution to the `refcount_t` work was the kernel-wide analysis and conversion of reference counting schemes. The conversion, from `atomic_t` to `refcount_t`, was built upon the analysis but implemented by

```
1  bool refcount_dec_and_test(refcount_t *r)
2  {
3      unsigned int old, new, val = atomic_read(&r->refs);
4
5      for (;;) {
6          if (unlikely(val == UINT_MAX))
7              return false;
8
9          if (val == 0) {
10             WARN_ONCE(val == 0,
11                 "underflow; use-after-free.\n");
12             return false;
13         }
14
15         new = val - 1;
16
17         old = atomic_cmpxchg_release(&r->refs, val, new);
18         if (old == val)
19             break;
20
21         val = old;
22     }
23
24     return !new;
25 }
```

Listing 24: Implementation of `refcount_dec_and_test`.

hand. The analysis was conducted using Coccinelle [Pad+08; 17b], which is used by specifying code patterns to search or replace. Coccinelle is integrated to `Kbuild`, the Linux kernel build system, and can be used to automatically detect potential problems or apply semantic patches. We are currently in the process of merging our reference counter Coccinelle patterns to the mainline Linux kernel, which will allow automated testing systems and interested developers to test new code for potential `refcount_t` use cases.

Coccinelle patterns are defined using the Semantic Patch Language (SmPL). Patterns can use regular expressions to identify functions of interest, and then define specific relationships or code patterns using these functions. The system can be used to transform code, but due to numerous small variations on reference counters and subtle semantic changes, we used it only for detection. We used three specific patterns, all of them using a combination of regular expressions and static analysis. The regular expressions are used to find specific `atomic_t` API calls, and other function calls typically related to object lifetimes (e.g., function-names containing the words `destroy` and `allocate`). Our patterns (Appendix A.1) specifically looked for three distinct use cases:

- Using the return value of `atomic_dec_and_test` to determine whether an object should be freed.
- Using `atomic_add_return` to decrement a counter (by adding a negative value), and based on the return value determine how to handle the object.
- Using `atomic_add_unless` to conditionally modify a counter only when it is larger than one.

Using these patterns, we found 266 potential reference counting schemes, of which we have patches covering 233 cases. The remaining are either unsuitable for conversion, or simply too complicated to convert without major modifications to the affected code bases. For example, the `inode` conversion spanned a total of 10 patches[4] and has yet to be accepted due to its complexity. The latter two patterns also present problems with the initial `refcount_t` API. Namely, the API cannot accommodate them because it lacks required functionality (e.g., there is no way to perform atomic modifications while checking for a value of one). The two latter patterns are typical for *object pool patterns*, where objects are recycled rather than freed. Recycling avoids repeated freeing and allocating of the objects and is an important optimization for systems that may need to rapidly handle massive amounts of similar objects. Based on these findings we worked with Peter Zijlstra to include additions to the `refcount_t` API [Res16a].

[4] `http://lkml.org/lkml/2017/2/24/599`

2.4.1.3 API Extension

Our `refcount_t` API extensions (List 3) include functions allowing addition and subtraction of arbitrary values and functions behaving conditionally on counters with the value zero [Res16a]. It should be noted that some of these functions could be worked around using the basic API, e.g., perhaps by using a lock and then performing individual operations to reach same results. However, as is evident, such workarounds would be prone to implementation errors, cause code bloat, and induce performance overheads.

The `refcount_dec_not_one` and `refcount_dec_if_one` functions are needed for implementing object pool patterns. The value of one is needed to indicate recyclable objects in the pool, i.e., objects that are still valid but unused. The networking subsystem, in particular, often uses this pattern for frequently used data structures. Our patches include six cases of this pattern. The arbitrary additions and subtractions are needed to accommodate cases where multiple references need to be dropped at once. The `sk_wmem_alloc` variable, for example, serves as both a reference counter and transfer queue size, thus requiring larger-than-one additions and subtractions. Finally, the `void` variants avoid redundant return value checks. The `btrfs` filesystem, for instance, sometimes handles nodes that are known to be cached. The reference held by the cache is therefore guaranteed, making return value checks redundant.

The generic implementations for the functions in the extended API are also constructed using the compare-and-swap loop. As an example consider the implementation of `refcount_add_not_zero` shown in Listing 25. No locking mechanism is needed, and only the `atomic_try_cmpxhg_relaxed` function is atomic (line 18). This specific `cmpxchg` variant incorporates value updates and success checks with the basic compare-and-swap operation (i.e., the return value tells whether the swap succeeded, and the conditional value `val` is set to the value of `&r->refs` before it was potentially changed). The return on line 8 ensures that only non-zero values are changed, whereas the return on line 11 ensures that saturated values remain unchanged. The assignment to `UINT_MAX`, on line 15, saturates values instead of allowing overflow. Finally, a warning is issued when the counter is initially saturated (line 20). Unless the prior value was zero or `UINT_MAX`, the function eventually modifies the value and returns at line 23. Note, again, the return values; only a counter value of zero results in a `false` return value. A saturated value, while constituting a memory leak, nonetheless indicates that the object is safe to use.

2.4.1.4 Other `refcount_t` Efforts

At the time of this writing, there are ongoing efforts to introduce options to limit `refcount_t` protections and offer high-performance architecture-specific

```
void refcount_add(unsigned int, refcount_t)
__must_check bool refcount_add_not_zero(unsigned int, refcount_t)
__must_check bool refcount_sub_and_test(unsigned int, refcount_t)
void refcount_dec(refcount_t)
__must_check bool refcount_dec_if_one(refcount_t)
__must_check bool refcount_dec_not_one(refcount_t)
```

List 3: Initial bare `refcount_t` API.

```
1  bool
2  refcount_add_not_zero(unsigned int i, refcount_t *r)
3  {
4      unsigned int new, val = atomic_read(&r->refs);
5
6      do {
7          if (!val)
8              return false;
9
10         if (unlikely(val == UINT_MAX))
11             return true;
12
13         new = val + i;
14         if (new < val)
15             new = UINT_MAX;
16
17     } while (!atomic_try_cmpxchg_relaxed(
18                     &r->refs, &val, new));
19
20     WARN_ONCE(new == UINT_MAX,
21               "saturated; leaking memory.\n");
22
23     return true;
24 }
```

Listing 25: `refcount_add_not_zero` [Tor, Linux v4.12, `include/linux/refcount_t.h`] increments the `refcount_t` unless it was zero, or saturated. The return is false if, and only if, `refcount_t` was zero. The implementation utilizes the a compare-and-swap pattern, thus requiring only the `atomic_try_cmpxchg_relaxed` on line 18 to be atomic.

implementations [Coo17b]. The protections, and thus performance overhead, can be controlled via the new `REFCOUNT_FULL` kernel configuration option, which at present is disabled by default. A fast x86 implementation, via a new `FAST_REFCOUNT` option, is targeted for inclusion in `v4.14` of the Linux kernel but is yet to be merged. The architecture-specific implementations are intended to provide a compromise between the two by dropping some of the stringent checks while still preventing the overflow. Both `FAST_REFCOUNT` and `REFCOUNT_FULL` are helpful in promoting adoption of the `refcount_t`, which even without protections offers an API safer than `atomic_t`.

Of particular interest, due to its parallels to our `HARDENED_ATOMIC` work, is the x86 `FAST_REFCOUNT` implementation by Kees Cook [Coo17c]. It treats `refcount_t` as a signed integer and uses any negative value to indicate saturation. The counter is modified with single-instruction additions and subtractions. Overflow and saturation are detected using the `JS` instruction. `JS` is a conditional jump that checks the CPU `SF` flag, which is set whenever the preceding instruction resulted in a negative value. On saturation the value is set to $-$`INT_MAX`$/2$. This approach theoretically suffers from a race-condition similar to `HARDENED_ATOMIC` (see Section 2.3.3) but exploitation would require a single addition or subtraction of $\pm$`INT_MAX`$/2$. The regular `refcount_t` value-range is reduced from $[0, $`UINT_MAX`$-1]$ to $[0, $`INT_MAX`$-1]$ but this is not a practical problem because the amount of valid reference counters is already limited by memory-constraints.

As an example consider the x86 fast `refcount_add` implementation (Listing 26). The implementation is almost identical to the `atomic_t` equivalent but with an added `REFCOUNT_CHECK_LT_ZERO` check. The check simply uses the `JS` instruction to jump to an error handler if a negative value is detected. The CPU overflow flag (`OF`) tells the error handler whether the running process caused the value to saturate, or whether it simply encountered an already saturated value (i.e., in which case the value was already negative and the OF flag not set). Note that there are no checks that detect increment-on-zero, which is a conscious trade-off for performance.

```
1  static __always_inline
2  void refcount_add(unsigned int i, refcount_t *r)
3  {
4      asm volatile(LOCK_PREFIX "addl %1,%0\n\t"
5          REFCOUNT_CHECK_LT_ZERO
6          : [counter] "+m" (r->refs.counter)
7          : "ir" (i)
8          : "cc", "cx");
9  }
```

Listing 26: The implementation for the x86 specific fast `refcount_add` function [Tor, Linux v4.13-rc1, `arch/x86/include/asm/refcount.h`].

At the time of writing, there is ongoing work on fast `refcount_t` implementations for other architectures such as ARM64 [Bie17]. These fast implementations are imperative for the adoption of `refcount_t` and even by themselves offer a substantial security improvement. Performance sensitive subsystems have been, and are, reluctant to approve patches that affect performance without any means to mitigate the impact.

2.4.2 Challenges

The `refcount_t` related challenges are mostly related to our kernel-wide adoption efforts and discussions on how permissive the `refcount_t` API should be. The ongoing work on various architecture-specific fast `refcount_t` implementations do naturally pose some more specific challenges, but our work does not directly touch upon these.

2.4.2.1 Object Pool Patterns

One common problem for direct conversions was the object pool pattern, i.e., cases where unused objects, instead of being completely freed, were put in a pool, or graveyard, for later reuse. This type of recycling avoids costly memory allocations by reusing the memory of frequently used but short-lived objects. The reference counter for recycled objects must thus be able to distinguish between freed objects and unreferenced but recyclable objects. `atomic_t` schemes typically use zero or one to indicate recyclable objects and zero or -1 to indicate freed objects.

`refcount_t` supports neither increments on zero or negative values, which makes most of these schemes incompatible for direct conversion. Instead, the schemes must be modified to use zero only for freed objects and one for objects not in use. One example of these kinds of conversions is the `struct inet_peer` data structure in the `net` subsystem. It used the reference counter value of zero to indicate objects in a deletion queue, and -1 for deleted, i.e., freed, objects. Our `refcount_t` conversion incremented the whole scheme by one, thereby keeping zero for freed objects while using the value one for recyclable objects. The extended API was designed to accommodate schemes such as this. Conceptually this modification uses regular reference counter semantics but requires that the object pool itself keeps an explicit reference. While such changes can be relatively straightforward, there is room for subtle errors due to the added `refcount_t` restrictions when the counter is zero.

2.4.2.2 Unconventional Reference Counters Redux

Some challenging `refcount_t` conversions are best described as unconventional reference counters. These include reference counters that serve purposes beyond traditional reference counters. One example is the `sk_wmem_alloc` field of the `struct sock` data structure in the `net` subsystem. The field,

according to the in-line documentation, holds the "*transmit queue bytes committed*" but also functions as a reference counter. It is questionable whether such instances should be converted to `refcount_t` in the first place. We have mostly erred on the side of conversion and relied on the maintainers to decide whether the conversions make sense or not.

In Section 2.4.2.2, we discussed the `percpu-refcount`, and why we could not protect it with `HARDENED_ATOMIC`. Because `refcount_t` is unsigned we do not hit the same problem here. Using an offset of `UINT_MAX`/2 would still effectively halve the usable range but would otherwise work correctly until saturation. An imbalance between the counts on individual CPUs is expected and acceptable as long as the joint counter, i.e., the sum of all CPU counters, is correct. In other words, one CPU could have more increments, whereas the other more decrements. If one CPU, however, is saturated, this imbalance would no longer be corrected because the saturation would consume and loose added increments. The `percpu-refcount` thus remains built upon `atomic_long_t`. Moreover, `FAST_REFCOUNT` and potentially other implementations are incompatible with the `percpu-refcount`.

2.5 Evaluation

This evaluation will focus on the `refcount_t` part of our work. Because the `HARDENED_ATOMIC` approach was discarded when `refcount_t` was introduced, we have not performed extensive measurements nor evaluations of it beyond the summary in Section 2.3.3. We will focus on the generic `refcount_t` implementations, not upcoming architecture-specific implementations such as `FAST_REFCOUNT`.

2.5.1 Security

Our security requirement (Requirement 1) is to prevent reference counter overflows. Overflow and saturation are mechanisms related to the increment and decrement functions (i.e., `refcount_set` is affected by neither overflow nor saturation). We, therefore, define *invariants* for the other functions in groups. Namely, the incrementing functions maintain the invariants that:

The resulting value will not be smaller than the original (I_1).

A value of zero will not be modified (I_2).

Similarly, the decrementing functions maintain the invariants that:

The resulting value will not be larger than the original (D_1).

A value of `UINT_MAX` will not be modified (D_2).

Let us first consider the incrementing functions by looking at the implementation for `refcount_add_not_zero` (Listing 25). Algorithm 5 shows the same function without implementation details. We see that Algorithm 5 has four possible outcomes:

1. If *refcount* = 0, then the function returns `false` without modifying the counter value (line 4), satisfying both I_1 and I_2.

2. If *refcount* = `UINT_MAX` then the function returns `true` without modifying the counter value (line 7), enforcing I_1 and trivially satisfying I_2.

3. If *refcount* $\in [1, \texttt{UINT\_MAX} - 1]$ and *refcount* + *summand* $\geq$ `UINT_MAX` (i.e., the addition would overflow), then the function returns `true` and sets the value of *refcount* to `UINT_MAX` (line 11), thus satisfying both I_1 and I_2.

4. If *refcount* $\in [1, \texttt{UINT\_MAX} - 1]$ and *refcount* + *summand* $<$ `UINT_MAX`, then the addition is performed and the function returns `true`, trivially satisfying both I_1 and I_2.

Since these four cases exhaustively cover all situations, we can conclude that the two invariants are satisfied in all situations. This description was specific to `refcount_add_not_zero`, but other incrementing functions are similar, i.e., they all employ the compare-and-swap loop and use similar checks to maintain the same security invariants.

All of the decrementing functions are implemented using the compare-and-swap loop, and in particular, maintain the same invariants. To analyze the decrement function invariants let us consider the `refcount_dec` function shown in Algorithm 6. The function has three possible outcomes:

1. If *refcount* = `UINT_MAX`, then the function returns without modifying the counter value (line 4), satisfying D_1 and enforcing D_2.

2. If *refcount* = 0, then the function emits a runtime warnings and returns without modifying the counter value (line 8), enforcing D_1 and trivially satisfying D_2.

3. Otherwise *refcount* $\in [1, \texttt{UINT\_MAX} - 1]$ the counter is decremented, trivially satisfying D_1 and D_2.

The other decrement functions are again similar, with the main difference being the presence of return values and checks to accommodate arbitrary subtractions. For non-void functions, a return value of true indicates that the referenced object is safe to free. Algorithm 6 omits runtime warnings that the Linux implementation uses to log error conditions (e.g., a decrement

Ensure: $retval = \textbf{true} \Leftrightarrow refcount > 0$
Ensure: $value\_unchanged \Leftrightarrow refcount \notin \{1, \texttt{UINT\_MAX} - 1\}$
1: $val \leftarrow refcount$ {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** $val = \texttt{UINT\_MAX}$ **then**
7: **return true** {counter is saturated, thus not zero}
8: **end if**
9: $new \leftarrow val + summand$ {calculate new value}
10: **if** $new < val$ **then**
11: $new \leftarrow \texttt{UINT\_MAX}$ {saturate instead of overflow}
12: **end if**
13: $old \leftarrow$`cmpxchg`$(refcount, val, new)$
14: **if** $old = val$ **then** {if *refcount* was unchanged, then}
15: **break** {value was updated by cmpxchg}
16: **end if**
17: $val \leftarrow old$ {update *val* for next iteration}
18: **end while**
19: **return true** {value incremented or saturated}

Algorithm 5: `refcount_add_not_zero`(*refcount*, *summand*) adds *summand* to *refcount*. It saturates on overflow, does not modify zero, and returns `true` if the prior value was non-zero. A compare-and-swap loop (using `cmpxchg` at line 13) provides atomicity.

Ensure: $retval = \textbf{true} \Leftrightarrow refcount = 0$
Ensure: $value\_unchanged \Leftrightarrow refcount \notin \{1, \texttt{UINT\_MAX} - 1\}$
1: $val \leftarrow refcount$ {use local copy}
2: **while true do**
3: **if** $val = UINT\_MAX$ **then**
4: **return** {counter is saturated, thus not zero}
5: **end if**
6: $new \leftarrow val + 1$ {calculate new value}
7: **if** $new > val$ **then**
8: **return** {counter not decremented beyond zero}
9: **end if**
10: $old \leftarrow$`cmpxchg`$(refcount, val, new)$
11: **if** $old = val$ **then** {if *refcount* was unchanged, then}
12: **break** {value was updated by cmpxchg}
13: **end if**
14: $val \leftarrow old$ {update *val* for next iteration}
15: **end while**
16: **return** {value incremented or saturated}

Algorithm 6: `refcount_dec`(*refcount*) performs a single decrement on the counter, without any return values. The compare-and-swap pattern is used to preserve atomicity, and neither 0 or `UINT_MAX` is modified.

to zero with a `void` function indicates a memory leak and a decrement on zero indicates a reference counter underflow and a use-after-free error).

All the implementations, both for increment and decrement functions, include similar runtime warnings that help with the detection of errors. The warnings are particularly useful because errors often manifest as hard-to-detect memory leaks but can when exploited lead to use-after-free or double-free errors. During the conversion efforts, some erroneous reference counting scheme errors[5] were detected and fixed due to the `refcount_t` restrictions and warnings.

2.5.2 Performance

There are no standardized measurements, payloads or acceptable overheads for low-level Linux kernel utilities such as `refcount_t`. Moreover, because `refcount_t` is a highly optimized and has relaxed memory ordering its performance could be highly dependent on use-case. We decided to use the networking subsystem as a test case because the networking conversions encountered much concern over performance overheads. The concerns are well-founded because the networking subsystem heavily relies on reference counters for efficient data management. We used the Netperf [17k] network performance measurement suite to measure the processing overhead and throughput loss due to `refcount_t`.

The Netperf benchmarking results (Table 2) were taken using dedicated physically connected Ubuntu server and client machines equipped with Haswell-EP6 processors. The base measurements were taken using a default `v4.11-rc8` kernel measuring traffic between the two machines, whereas the `refcount_t` results used the same kernel with the `refcount_t` patches applied and all 78 reference counters in the networking subsystem converted. We ran each test case three times with individual durations of 300 seconds. The CPU utilization measurements use the `TCP_STREAM` and `UDP_STREAM` test cases, whereas throughput uses the `TCP_SENDFILE` and `TCP_RR` tests.

Based on these results `refcount_t` introduces a measurable processing overhead. This overhead is negligible for systems with abundant processing resources and where network use is sporadic, such as typical desktop computers. Networking systems, like routers, however, have different traffic patterns and could be considerably affected by this processing overhead. While the network performance, both regarding TCP transactions per second and throughput, itself was unaffected, the relatively low CPU utilization indicates that bottlenecks were elsewhere. The results justify the `REFCOUNT_FULL` and `FAST_REFCOUNT` efforts (Section 2.4.1.4) that allow performance-critical systems to benefit from the new API without performance loss. `FAST_REFCOUNT`, in particular, is set to provide architecture-specific performance on par

[5] `http://lkml.org/lkml/2017/6/27/409`, `http://lkml.org/lkml/2017/3/28/383`

Test	Base	`refcount_t`	Δ (stddev)
UDP CPU use (%)	0.53	0.75	0.22 (0.18)
TCP CPU use (%)	1.13	1.28	0.15 (0.03)
TCP throughput (Mbps)	9358	9305	−53 (0)
TCP throughput (tps)	14909	14761	−148 (0)

Table 2: `refcount_t` performance measurements using Netperf. The CPU use indicates system CPU utilization (i.e., low values are better). The other two measure throughput, TCP_SENDFILE in Mb per second, and TCP_RR in transactions per second.

with `atomic_t` while still providing the overflow protection. The generic `refcount_t` does, however, impose a trade-off between security and performance (Requirements 1 and 2).

2.5.3 Usability

The usability of kernel source code is ultimately decided by its acceptance and use. In general, additions should be minimal, focused and self-documenting. The `refcount_t` API unambiguously denotes its use for reference counting through its naming, thus avoiding any confusion with general purpose integers. It is also focused and consists of only 14 functions, whereas the `atomic_t` API consists of over 100 functions. As noted in our security evaluation, some erroneous use cases cannot even be implemented with `refcount_t`. The various runtime and compile-time warnings also promote safe use and easy error-detection. Independent work has also adopted the `refcount_t` API[Zab17], lending further credence to its usability. The continued acceptance of our patches — currently 125 patches merged to mainline — indicate that the usability goals have been reached.

2.5.4 Maintainability

While maintainability can be challenging to measure, we can provide an estimate based on code size and cross-dependency. The generic `refcount_t` implementation is self-contained and depends only on the stable `atomic_t` API. Architecture-specific implementations (e.g., `FAST_REFCOUNT`) can be provided, but are not needed. The maintenance overhead is thus restricted to the implementations in `code/refcount.c`, the `include/refcount.h` header and the dependency to the stable `atomic_t` API. This is a marked improvement to the `HARDENED_ATOMIC` approach that touched numerous files and added several header files with complex macro-defined inheritance and cross-dependencies.

2.5.5 Summary of Evaluation

Table 3 shows a summary of our requirements. The primary goal — preventing reference counter overflows — is fulfilled by `refcount_t`. Performance overheads, while small, can be problematic for some systems. Nonetheless, coupled with ongoing efforts such as `FAST_REFCOUNT`, we regard the performance requirements met to a reasonable degree. The usability and maintainability concerns are also largely met due self-contained and minimal `refcount_t`. Moreover, these assessments are supported by the continued uptake of `refcount_t` patches into the remaining subsystems.

Requirements	
Req. 1 Security	Overflows completely prevented.
Req. 2 Performance	Minimal overhead, none with `FAST_REFCOUNT`.
Req. 3 Usability	Clean and minimal API.
Req. 4 Maintainability	Self-contained implementation and API.

Table 3: Evaluation of `refcount_t` against our requirements.

3 Preventing Pointer Bound Violations

The second part of our work focuses on spatial memory errors in the Linux kernel. Spatial memory errors include the infamous buffer overflow and happen when a pointer is used to dereference memory outside the bounds of the object referenced by the pointer [SPW13]. Memory-safe languages are not prone to these problems. The Linux kernel, however, is implemented in C and thus susceptible to memory errors. The kernel employs various tactics to restrict the exploitability of spatial memory errors, including probabilistic features such as Kernel Address Space Layout Randomization (KASLR) [Edg13]. However, full spatial memory safety requires complete mediation of pointer access, which can be achieved by *pointer bounds checking* [SPW13]. Linux provides built-in support for pointer bounds checking via the Kernel Address Sanitizer (KASAN) [17j]. KASAN is mainly used as a testing tool because the performance overheads make it unsuitable for most production use.

Intel recently introduced the Memory Protection Extensions (MPX) CPU extension. It provides efficient hardware-assisted pointer bounds checking [Int16]. We set out to explore whether MPX could be used to prevent in-kernel spatial memory errors in Linux. While MPX is is a promising new technology, its existing software components are geared towards user-space, thus making its use for kernel-space code non-trivial. In Section 3.1 we provide background on pointer bounds checking and Intel MPX specifically. In particular, we look at implementation aspects of MPX, including the GCC compiler instrumentation and Linux kernel GCC-plugins. In Section 3.2 we present a detailed formulation of our problem statement. In Section 3.3 we dive into the design and implementation of MPXK, our solution for using MPX to protect the Linux kernel from spatial memory error vulnerabilities. Finally, in Section 3.5 we will evaluate the current state of MPXK, including security and performance considerations.

3.1 Background

Pointer bounds checking is a method for mitigating or preventing spatial memory errors (Section 3.1.1). In our work we are using Intel MPX specifically (Section 3.1.2). Intel MPX consists of both hardware and software components (Sections 3.1.3). It is supported by the GCC compiler (Section 3.1.4) and uses it for compile-time instrumentation (Section 3.1.5). To adapt it for MPXK we use the Linux kernel GCC plugin system (Section 3.1.6).

3.1.1 Pointer Bounds Checking

Pointer bounds checking, when used for complete mediation of pointer use, guarantees the spatial memory safety of an application. The size and memory address of the pointed-to data defines the bounds of a pointer. For pointers

to static data, the bounds are [`address`, `address` + `size_of_type`] and to dynamic memory [`address`, `address` + `size_of_allocation`]. Pointer bounds checking means that each memory access is checked against those bounds, with failed checks resulting in a bound violation (Figure 1).

The bounds of a pointer: A pointer bounds checking mechanism tracks pointers and checks their values before use, typically through compile-time or binary instrumentation [SPW13]. Bounds are defined on pointer assignment or allocation. However, the extent of the bounds is clouded by varied pointer use. Consider dynamically allocated memory regions containing several distinct objects, e.g., a nested data structure, or an array. It is no longer clear whether a pointer to the inner elements should have bounds defined based on the outer or inner structure. The problem is not purely theoretical; for instance, pointers to array items are often used as iterators and must thus have bounds defined by the containing array. On the one hand, restricting the bounds to the innermost structure, i.e., *narrowing* the bounds, provides finer granularity and can catch more errors.

Source and binary compatibility: A related aspect is that of *source compatibility*, i.e., whether existing source code can be used as is. Source compatible schemes infer the correct semantics for pointer bounds based on existing source code and must often make choices such as narrowing based on incomplete data, typically necessitating instrumentation that favors compatibility over security. Another practical distinction is *binary compatibility*, i.e., whether it is possible to combine instrumented code with legacy (i.e., non-instrumented) code. Binary compatibility is required when using system libraries or proprietary code that cannot be recompiled. Pointers manipulated by legacy code are by definition unknown to the instrumentation. Binary compatible systems therefore typically assign infinite bounds to pointers originating or manipulated by legacy code [SPW13].

Storing the bounds: In practice, any bounds checking mechanism needs means to track the bounds of specific pointers. In trivial cases, such as locally used, scoped and isolated pointers, this can be accomplished by hard-coded bounds or by storing the bounds directly on the stack. In many cases, this is however not feasible. Consider for instance pointers in a dynamically sized array; because the compiler does not know the size of the array, it cannot reserve space for the bounds associated with the contained pointers. Existing systems solve this problem by storing bounds in some external data

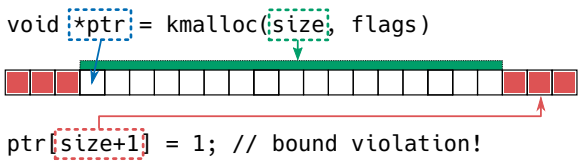

Figure 1: The pointer bounds of a malloc allocated pointer.

structure, in *shadow-memory* or using *fat-pointers*. Fat-pointer systems, such as CCured [Nec+05], change the pointer implementation to include the bounds in the pointers themselves, thus changing the memory layout of pointers and making the systems binary-incompatible. Shadow memory systems (e.g., Valgrind [NS07a] and AddressSanitizer [Ser+12]) maintain a parallel memory mapping — the shadow memory — to track pointer bounds. They typically suffer from high memory use, e.g., AddressSanitizer reports memory overheads between 140% and 2455%. Pointer bounds can also be stored in a separate data-structure without using fat-pointers or shadow-memory; for example, SoftBound [Nag+09] and Intel MPX [Int16] does this. This approach can reduce memory overheads, e.g., a comparison between MPX and AddressSanitizer reported total memory utilization of 190% and 280%, respectively [Ole+17].

Pointer- and object-based bounds: A final distinction is whether bounds are pointer- or object-based (Figure 2). That is, whether the bounds are loaded based on the address of the pointer or the address of the pointed-to object, i.e., the pointer's value. Object-based bounds must rely on the pointer addressing the correct object. Consider for instance the scenario in Listing 27. Because the bounds depend on the value of the pointer, the mechanism cannot detect if the pointer is corrupted to point into another valid object. However, exploiting object-based bounds is non-trivial and in particular necessitates that the attacker can overwrite the pointer using some other vulnerability (i.e., overflow or indexing errors would not trigger this behavior).

3.1.2 Intel Memory Protection Extensions

Intel MPX is a pointer-based bounds checking mechanism that prevents spatial memory errors. It is both source and binary compatible. MPX at its core is a hardware extension to the x86 architecture, with support available from Intel Skylake and Atom Goldmount onwards. As a pointer-based mechanism, MPX associates each pointer with appropriate bounds and then checks those bounds before dereferencing said pointers. In addition to the hardware, MPX requires software support form the operating system, compiler and MPX libraries. This section provides a high-level overview of MPX, whereas Section 3.1.3 provides a more in-depth look of MPX in user-space, and finally Section 3.1.5 looks at some of the implementation details of GCC compiler support for MPX.

The MPX ecosystem is designed, not only for performance and security but also to support easy adoption and usability via binary compatibility. MPX enabled binaries are backward compatible with legacy systems and will run on hardware and operating systems without MPX support, although naturally without the MPX functionality. Backwards compatibility is achieved by ensuring that the added hardware instructions map to NOPs, i.e., in-

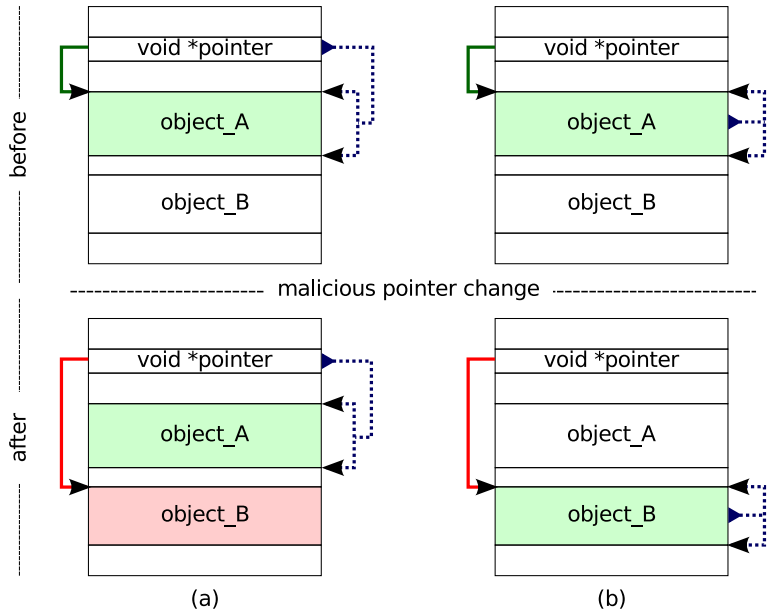

Figure 2: The bounds of a pointer can be either pointer-based (a) or object-based (b). Pointer-based bounds do not change when the pointer's value is changed. Object-based bounds change if the pointer's value is changed so that it points into another object.

```
1  struct user {
2      name* name;
3      pw* pw;
4  }
5
6  struct name *n = get(user);
7
8  check_bounds(n) /* OK. */
9  /* Bounds from user.name */
10
11 /* Bad pointer arithmetic! */
12 name += sizeof(struct name*);
13
14 check_bounds(n) /* Still OK! */
15 /* Bounds were loaded from user.pw */
```

Listing 27: Problem with naive-object oriented bounds checking. Because the bounds are associated with the object, not the pointer, it is impossible to determine whether the object itself is correct.

structions that do nothing, on legacy systems. The initialization code uses operating system calls to check for MPX support before doing any work. Because MPX is source compatible, its use requires little effort and no source code modifications. Only the appropriate compiler flags are required to use MPX.

GCC exposes the new MPX instructions (Table 4) to the programmer via built-in functions for reading, creating and checking bounds [17f]. In regular use, there is little need for the programmer to directly call the MPX instructions or built-ins. MPX also provides new registers; most notably the four bounds registers, `bnd0` to `bnd3`, which are used for storing and using bounds. The new `bndcfgu` and `bndcfgs` registers control MPX in user-space (ring 3) and kernel-space (ring 0), respectively. These registers also specify the memory address of the *Bounds Directory* (BD), discussed shortly, and the `BNDPRESERVE` and `ENABLE` flags. A final new register, `BNDSTATUS`, is used to convey information on similarly new MPX *BOUND Range Exceeded* (`#BR`) exceptions. The user-space registers are process-specific, which means that MPX must be enabled individually for each process using it.

Storing pointer bounds: The MPX bounds are pointer-based but stored in separate metadata. The instrumentation, when possible, stores bounds either statically, on the stack, or in the four added `bndx` registers. Function call instrumentation propagates bounds into called functions. The compiler can thus often treat bounds as any other variable and propagate them along stack and registers. However, it cannot pre-allocate bounds for dynamically stored pointers, such as those in dynamically sized arrays. In such cases, MPX uses the new `bndstx` and `bndldx` instructions to store and load the pointer bounds via a process-specific BD. The BD contains pointers to Bound Tables (BT), which in turn contains the Bound Table Entries (BTE). The pointer's linear address and `bndcfgu` determine the exact BTE to use (Figure 3). A BTE contains reserved bits and the pointer's value, lower bound and upper bound. On 64bit systems, the BD is 2GB, each BT 4MB, and the individual entries 32-bytes. The operating system or the process must manage the BD nd BTs since the MPX hardware does not. In practice,

`BNDMK`	Create bounds in bounds register
`BNDCL`	Check pointer lower bound against bounds register
`BNDCU`	Check pointer upper bound against bounds register
`BNDCN`	Check upper against 1's complement register register
`BNDMOV`	Copy/load bounds from/to memory or bounds register
`BNDLDX`	Store bounds into bounds table
`BNDSTX`	Load bounds from bounds table

Table 4: The MPX instruction set

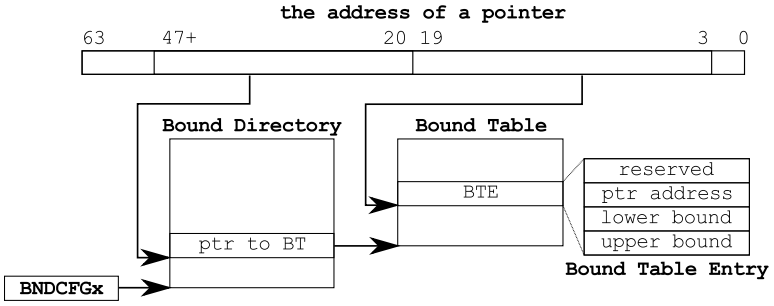

Figure 3: The BTE address translation used when loading or storing bounds via `bndldx` or `bndstx`.

these are allocated on demand partially by the MPX enabled program itself, and partially by the MPX aware operating system.

Checking bounds: The checking mechanism using the new `bndcu/l` instructions operates on a given `bndx` register and a pointer. Depending on the situation the bounds might need to be loaded with `bndldx` or copied with `bndmove` from directly accessible memory, i.e., the stack or data section. Failed checks are not handled by the running process but instead cause the CPU to issue a `#BR` exception which is handled by the kernel exception handler. The CPU also populates the `BNDSTATUS` register with additional information for error handling and diagnosis. The kernel handler kills the process outright or allows the process to continue while ignoring or logging errors. The default behavior is to only log errors to `stderr` but this can be controlled with environmental variables.

The code in Listing 28 shows an example program that would benefit from MPX because it can cause a buffer overflow at line 11. While the erroneous input handling in the example code is evident, it is conceptually similar to the real spatial memory errors. This example also shows a situation where `bndldx` must be used. The compiler can only pass the bounds of the outer pointer into the function and therefore must, at line 10 use `bndldx` to load the bounds for the pointed-to pointer.

3.1.3 Implementation of MPX

This section presents the MPX process initialization and Linux kernel support for MPX. The discussion is limited to the Linux kernel and GCC compiler. However, the Intel C++ Compiler (ICC) and the Microsoft Windows operating system also support MPX. The Linux kernel, before our work, only provides user-space support and does not support MPX for securing kernel-space.

```c
#include <stdio.h>
#include <stdlib.h>

char *hello = "Hello World\n";
char *secret = "password";

void printptr(int i, char **ptr2str)
{
    /* Bounds for ptr2ptr passed in via function call */
    char *str = (*ptr2str); /* BNDLDX needed!*/
    char chr = str[i]; /* can overflow! */
    printf("%d: %c\n", i, chr);
}

void main(int argc, char **argv)
{
    int i;
    scanf("%d", &i);
    printptr(i, &hello);
}
```

Listing 28: Example to illustrate bound propagation in MPX. While the bounds for the `ptr2ptr` pointer can be passed via the function arguments, the bounds of inner pointer `*ptr2ptr` cannot. The compiler therefore must instead use `bndldx` to load the bounds for `*ptr2ptr` and then assign those bounds to the `str` pointer.

MPX is a per-process mechanism and must to be initialized by the process during its startup. The initialization checks the CPU for MPX support, reserves memory space for the Bound Directory (BD), registers exception handlers with the kernel, and finally enables MPX by writing the configuration into the `bndcfgu` register via a system call. Both `bndcfgu` and its in-kernel equivalent, `bndcfgs`, follow the same layout and contain the BD address, reserved space, and the `BNDPRESERVE` and `ENABLE` flags (Figure 4).

Memory use: The Intel Developer's Manual states that the running process must itself allocate storage for BD and used Bound Tables (BTs) [Int16]. In practice, this is not strictly the case. The BD is allocated by the process using `mmap` as an anonymous and private memory area. The anonymous memory range is reserved but not mapped to physical memory. The CPU issues a *Page Fault* (`#PF`) if unmapped memory is accessed, which allows the kernel to swap or reserve physical memory transparently before transferring control back to the process. This is not MPX specific and is indeed why user-space processes can treat the virtual address-space as endless (although within addressing constraints and subject to hardware limitations). An unallocated BT entry similarly causes MPX to issue a `#BR`. The kernel can, based on `BNDSTATUS`, distinguish this `#BR` from a bound violation and again transparently allocate the missing BT. Because of these on-demand mechanisms, MPX consumes significantly less physical memory than what is reserved for the 2`GB` BD and the 4`MB` BTs. However, memory usage is highly dependant on the memory access patterns of applications [Kuv+17].

Binary compatibility: While the binary compatibility of MPX is convenient, it poses a problem when pointers are passed into legacy code. The legacy code could potentially modify the pointer but cannot update the associated bounds. MPX attempts to remedy the situation by using `bndstx` and `bndldx` to detect pointer modifications by legacy code. The BT entries store not only the bounds but also the value of the pointer, which `bndldx` compares to the current pointer value. If the pointer value has been modified MPX sets the pointer's bounds to infinite, i.e., it accepts any pointer [Ram+16]. Bounds for pointers passed into legacy code are always stored with `bndstx` and `bndldx`, thus ensuring that legacy modifications are detected.

Figure 4: The MPX configuration register layout

Narrowing: MPX employs selective narrowing, i.e., it sometimes attempts to narrow the bounds of pointers into nested data structures so that the bounds correspond to the innermost object. If an assignment involves `struct` fields or array items, it would seem natural to narrow the bounds to those of the specific item, not the whole container. However, using a pointer as an iterator is not compatible with such strict narrowing (Listing 29). Another common but incompatible idiom is makeshift inheritance using initial `struct` fields to hold the parent class (Listing 30). GCC provides compiler flags that disable narrowing and when narrowing is enabled applies the following rules [17f]:

- "If there are static array accesses then bounds of the outermost array are taken"
- "If there are no static array accesses then bounds of the innermost field, which is not the first in outer object, are taken"

3.1.4 The GCC Compiler

Before diving into GCC MPX instrumentation, it behooves to take a brief look at the GCC compiler in general [17d]. Conceptually GCC works by parsing source code into an internal format, and then executing compiler passes that do stepwise changes and finally produces an executable binary. Figure 5 shows a high-level representation of the different compilation stages. We will here focus on passes modifying the GIMPLE[6]and Register Transfer Language (RTL) intermediate representations (IR), but a typical compilation includes various other stages, including Interprocedural Analysis (IPA) passes.

GIMPLE passes process one function at a time. The initial passes represent a function as a flat sequence of statements, which in later passes is split into basic blocks. Each basic block is a self-contained chunk of code that can be jumped into, e.g., an if-else clause produces one basic block for each execution path. GIMPLE and RTL are target-independent but often describes target specific features (e.g., it can include MPX-specific calls). GIMPLE uses trees to represent data (e.g., attributes and arguments) and GIMPLE tuples to represent statements (e.g., functions and operands). In our work, we mostly interact with `GIMPLE_CALL` statements that represent function calls. The `GIMPLE_CALL` statement includes various information on the function declaration, arguments, and return value but it also encapsulates information for the specific call instance (i.e., values for function parameters), definition availability (i.e., is the function external or in the same translation unit), and potentially the whole function body.

[6]GIMPLE is based on the older GENERIC intermediary representation and is influenced by the SIMPLE Intermediary Language used in the McCat compiler project [17d; Hen+93].

```
1  void upper_case(char *string, int start) {
2      char *c = &(string[start]);
3      /* bounds of c  =
4       *       bounds of string?
5       *       bounds of one char?
6       *       bounds of string[start, ...]?
7       */
8
9      for ( ; *c != '\0'; c++) {
10         *c = *c - 32;
11     }
12 }
```

Listing 29: The example code implements a naive `upper_case` function. It uses a `*char` pointer as an iterator, which depending on how the bounds are assigned on line 2 can cause bound violations.

```
1  struct base_class {
2      int type;
3      int common;
4  }
5
6  struct sub_class {
7      struct base_class parent;
8      char *stuff;
9  }
10
11 void use_obj(struct base *o) {
12     struct *sub s;
13
14     switch(o->type) {
15     case 1:
16         s = (struct sub *)o;
17         ...
```

Listing 30: The example code uses a base `struct` field which can be used either as the base or by casting as the larger surrounding `struct`. It is not immediately clear how bounds to such fields should be handled correctly and without introducing source compatibility issues.

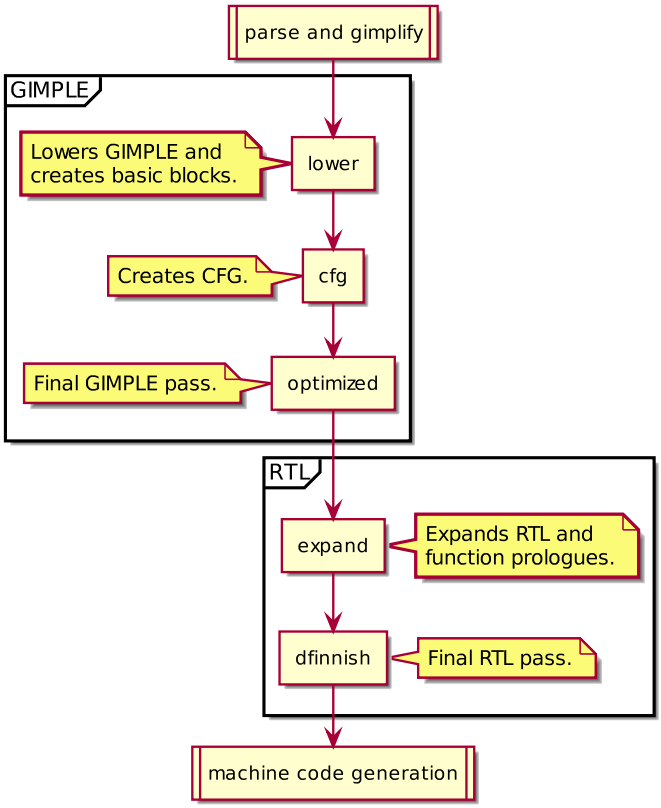

Figure 5: Some GCC compiler passes. The first and last stages include various intermediate stages not discussed here. Only a small fraction of GIMPLE and RTL passes are shown, they number over 150 on a default Ubuntu kernel compilation.

The compiler expands the code into an RTL representation after the GIMPLE passes. The RTL passes are also processed one function at a time. Whereas GIMPLE is quite high-level and resembles C source code, RTL uses a Lisp-like syntax and does not use high-level abstractions such as variables and statements. The GIMPLE variables are assigned to registers or memory, and statements are converted to RTL instructions. The RTL expansion also expands function calls, i.e., populates the stack and registers with function arguments and uses an RTL `call` instruction to enter the functions. After the RTL passes are completed, the compiler eventually produced machine-readable code.

The stepwise separation into individual passes allows each individual pass to be highly specific, and ideally, self-contained. That said, the individual passes can, in fact, interfere with each other and have cross dependencies. Passes can depend on specific ordering; it is, for instance, not possible to use the control flow graph (CFG) before the `cfg` pass has created it. Optimizations and internal house-keeping also pose challenges. For example, the compiler can use the CFG to remove unreachable basic blocks and unused static functions. Among other things, the compiler thus needs to ensure that both the code (i.e., GIMPLE or RTL) is correct and that the CFG is current. The default kernel and configuration on Ubuntu 17.04 uses over 200 passes during compilation, showing that this cross-dependency is far from trivial.

3.1.5 MPX Instrumentation

The GCC MPX instrumentation uses several passes, including IPA, GIMPLE and RTL passes. One could loosely divide it into the generic *Pointer Bounds Checker* instrumentation and lower-level hardware-specific MPX instrumentation. The Pointer Bounds Checker consists of GIMPLE and IPA passes that insert bounds checks and replaces function calls with instrumented variants. The MPX-specific implementation, in contrast, works at the RTL layer. For instance, pointer bounds in GIMPLE are of the abstract `pointer_bounds_type_node` type, whereas in RTL they are assigned storage on the stack or in registers. Bounds for function arguments are also handled during the RTL expansion, i.e., GIMPLE adds bounds to function definitions, whereas RTL writes the bounds to specific registers or invokes `bndstx`.

Although the GCC Pointer Bounds Checker is a generic high-level implementation for pointer bounds checking, its use currently requires MPX hardware [17e]. GCC provides compiler built-ins and attributes for explicit control of the instrumentation. The built-ins, i.e., functions provided by the compiler, can be used for direct checking, creating and assigning of bounds. The compiler attributes modify instrumentation behavior and include the `bnd_legacy` attribute that can be used to disable instrumentation of specific functions. GCC also provides the MPX initialization code and automatically calls it on process startup.

```
1 void *
2 __mpx_wrapper_malloc (size_t size)
3 {
4   void *p = (void *)malloc (size);
5   if (!p) return __bnd_null_ptr_bounds (p);
6   return __bnd_set_ptr_bounds (p, size);
7 }
```

Listing 31: MPX replaces all `malloc` calls with `__mpx_wrapper_malloc` calls [17g, Version 6.3.0, `libmpx/mpxwrap/mpx_wrappers.c`, Line 487]. The wrapper executes the `malloc` call and assigns bounds to the returned pointer.

MPX instruments memory-altering functions using wrapper functions, i.e., it replaces the original function calls with wrapper functions. The wrapper for `malloc` (Listing 31), for instance, first allocates the pointer by calling `malloc` and then returns it with `__bnd_set_ptr_bounds` assigned bounds. When the compiler handles function calls during RTL expansion, it also ensures that bounds are returned via the `bnd0` register. In more complex cases the wrapper might also need to verify bounds and update already existing bounds. For instance, the `memmove` wrapper (Listing 32) checks that the given memory range does not overflow. The memory could, however, potentially contain pointers. Because the bounds are associated with the pointer's memory address, they will no longer be valid when the pointer is moved. This is solved by also moving related Bound Tables (BTs), either by assignment or using the `move_bounds` function found at [17g, Version 6.3.0, `libmpx/mpxwrap/mpx_wrappers.c.h`, Line 205]. The finer details are not of interest here, particularly since our MPXK implementation does not need the BTs.

An astute reader might have realized that the limited number of bounds registers might prove problematic if several pointers are passed into a function. And indeed, if the four bounds registers cannot hold all the bounds for arguments MPX instead stores the bounds via `bndstx` before the call and again restores them inside the function via `bndldx` (although there are some exceptions, which are discussed in Section 3.4.2). Some language features also impose variations to this scheme. For example, functions with variable argument lists cannot use the bounds registers for the variable arguments. Variable list functions take an arbitrary number of arguments and use special accessors to retrieve the arguments via a `va_list` data structure [Lin]. The instrumentation handles variable argument lists similarly to other data structures containing pointers, i.e., with `bndstx` and `bndldx`.

```
1  void *__mpx_wrapper_memmove(void *dst,
2                              const void *src,
3                              size_t n)
4  {
5    if (n == 0)
6      return dst;
7
8    __bnd_chk_ptr_bounds (dst, n);
9    __bnd_chk_ptr_bounds (src, n);
10
11   /* When we copy exactly one pointer it is faster to
12      just use bndldx + bndstx.  */
13   if (n == sizeof (void *))
14     {
15       void *const *s = (void *const *) src;
16       void **d = (void **) dst;
17       *d = *s;
18       return dst;
19     }
20
21   memmove (dst, src, n);
22
23   /* Not necessary to copy bounds if size is less
24    * then size of pointer or SRC==DST.  */
25   if ((n >= sizeof (void *)) && (src != dst))
26     move_bounds (dst, src, n);
27
28   return dst;
29 }
```

Listing 32: The `__mpx_wrapper_memmove` function [17g, Version 6.3.0, `libmpx/mpxwrap/mpx_wrappers.c`, Line 487] checks bounds for incoming arguments and updates destination bounds.

```
1  static unsigned int cyc_complexity_execute(void)
2  {
3      int complexity;
4      expanded_location xloc;
5
6      /* M = E - N + 2P */
7      complexity = n_edges_for_fn(cfun) -
8          n_basic_blocks_for_fn(cfun) + 2;
9
10     xloc = expand_location(
11         DECL_SOURCE_LOCATION(current_function_decl));
12     fprintf(stderr, "Cyclomatic Complexity %d %s:%s\n",
13         complexity, xloc.file,
14         DECL_NAME_POINTER(current_function_decl));
15
16     return 0;
17 }
```

Listing 33: The execution function for the Linux kernel cyclomatic complexity GCC plugin [Tor, Linux v4.8, `scripts/gcc-plugins/cyc_complexity_plugin.c`]. The plugin itself requires a bit more code for registering the plugin, but most of the work is handled by the common Linux GCC plugin headers.

3.1.6 Linux GCC Plugin Infrastructure

Since version 4.5.0, released in 2010, GCC has supported plugins that allow easy incorporation of custom compiler passes. In 2011 `PaX/Grsecurity` implemented Linux `Kbuild` support for using plugins in the kernel build process [Cor11]. The GCC-plugin framework was upstreamed into the mainline Linux kernel and has been available there since Linux v4.8, released in 2016 [Cor16]. It is built on the existing GCC infrastructure and adds Linux-specific header files for the common plugin management tasks. In practice, the plugin itself needs only to define some setup variables and provide an initialization function. Typically the plugin also includes some functionality that is implemented in its `execute` function. Listing 33 shows the GIMPLE `execute` function for the Cyclomatic complexity plugin, one of the first plugins that landed in the mainline kernel [Cor16].

The compiler exposes several global variables which can be used to manipulate the current state, including the `cfun` variable for the currently processed function. The `cfun` data structure contains the function body and declaration, including argument and attributes. Depending on the compilation phase the function body consists either of a flat sequence of GIMPLE instruction or if the CFG has been constructed, of separate basic blocks. The GIMPLE sequences include representations of source code statements, including function calls, but also instructions inserted by the compiler (e.g., MPX instrumentation). The RTL passes are similar, but

instead of the GIMPLE statements have sequences of RTL instructions. The Linux GCC-plugin framework supports both RTL and GIMPLE passes, but also IPA passes.

While the plugins are ill-suited for complex internal modifications, not to mention completely new languages, they provide a convenient way to add functionality without GCC source code modifications. As such, plugins are particularly well suited for specific problems targeting only a subset of users. Since the introduction of the Linux plugin infrastructure, there have indeed been many efforts to add kernel security features using plugins. The plugins can not only be convenient to implement, but they also sidestep the problem of doing kernel-wide source code changes such as we faced with our `refcount_t` conversion patches (see Section 2.5).

3.2 Problem Statement

The goal of this work is to prevent spatial memory errors in the Linux kernel. We specifically want to explore the usability of Intel MPX to protect in-kernel execution of code with minimal performance overhead. We aim to maintain the binary compatibility of MPX, i.e., allow it to be applied to only selected subsystems or modules without causing kernel-wide performance overheads. Finally, our solution should not require extensive code changes; otherwise, there is little chance for its inclusion in the mainline kernel.

The solution must thus fulfill the following requirements:

1. **Spatial memory errors in instrumented code must be prevented.**
2. **The solution must be compatible with existing source code.**
3. **The solution must be modular and applicable to only select subsystems.**
4. **The solution must impose minimal performance overheads.**

3.3 MPXK

Our pointer bounds checking system, MPXK, uses Intel MPX to prevent spatial memory errors. The main challenge in using MPX is the memory use, i.e., a 2GB BD and several 4MB BTs. MPXK cannot use the user-space method to minimize memory use (Section 3.1.3), because the kernel cannot handle page faults or missing BTs caused by itself.

Preallocating potentially used BD ranges and BTs would increase memory consumption by at least 500% or introduce excessive performance overheads. MPXK solves this by using existing in-kernel metadata to reacquire bounds when needed, instead of using `bndldx` and `bndstx`. Therefore, MPXK does not need to manage BD or BT data. Memory overheads are thus minimal and consist only of increased code size.

The MPX hardware, the GCC compiler, and the Linux kernel already provide a user-space implementation. The MPX libraries and user-space support are self-contained and provide a good starting point for MPXK. The complexity and size the GCC Pointer Bounds Checker and MPX instrumentation is however not as easily manageable (see Section 3.1.5). Because of this, we use the existing GCC instrumentation along with a new GCC-plugin that adapts the instrumentation for our purposes. The MPXK implementation consists of three parts: in-kernel initialization (Section 3.3.1), in-kernel support (Section 3.3.2), and a kernel GCC-plugin (Section 3.3.3).

3.3.1 MPX Initialization and Setup

MPXK is an optional feature. It is integrated into `Kbuild` and provides a new `Kconfig` option, `CONFIG_X86_INTEL_MPX_KERNEL`, that controls its use during compilation. The compile-time option requires architecture-support for MPX, but MPXK also performs a CPU feature check before attempting to enable itself during boot. The CPU has a separate MPX configuration register for ring 0 execution; this must be set up by MPXK. The register, `bndcfgs`, is a Machine Specific Register (MSR) and is accessed via standard kernel accessors. The `bndcfgs` layout is identical to the user-space equivalent (Figure 4). In addition to the `enable` flag, the configuration must include a valid linear address to the BD. Because MPXK does not use the BD, the address is never needed. However, if the BD address points to mapped memory, an attacker could introduce `bndstx` instructions (e.g., via a third-party driver) and potentially cause arbitrary memory writes, i.e., write BTE data over other memory. Therefore MPXK reserves a memory address for the BD but does not back it with physical memory. Any attempts to access the BD thus cause a page fault and subsequent kernel panic.

To be available as early as possible MPXK performs the MPX configuration during basic setup after the memory system initialization. The `mpxk_enable_mpx` function (Listing 34) enables the instrumentation and is

```
1  __attribute__((bnd_legacy))
2  static void mpxk_enable_mpx_cfgs_cpu(void *info) {
3      (void) info;
4      wrmsrl(MSR_IA32_BNDCFGS, bnd_cfg_s.q);
5  }
6
7  __attribute__((bnd_legacy))
8  void mpxk_enable_mpx(void) {
9      void * ptr = get_vm_area(MPX_BD_SIZE_BYTES_64 +
10                              PAGE_SIZE, VM_MAP);
11
12     bnd_cfg_s.q = PAGE_ALIGN((unsigned long) ptr);
13     bnd_cfg_s.q |= MPX_BNDCFG_ENABLE_FLAG;
14
15     on_each_cpu(mpxk_enable_mpx_cfgs_cpu, NULL, 1);
16 }
```

Listing 34: The `mpxk_enable_mpx` function initializes MPX on all available CPUs.

called from the standard `do_basic_setup` function in `init/main.c`. The function uses `get_vm_area` to reserve a memory region without backing it with physical memory. Based on MPX requirements the address is aligned. The configuration is stored in a global variable, which is then written to each CPUs `bndcfgs` register using `on_each_cpu`.

3.3.2 Supporting Memory Functions

MPXK uses its `bndldx` replacement, `mpxk_load_bounds` (Listing 35), to acquire bounds via existing in-kernel memory management metadata. It is implemented as a statically linked in-kernel function in `arch/x86/lib/mpxk.c`. No header file exposes the function; the MPXK GCC-plugin instead inserts calls to it where needed. The function currently supports pointers allocated with `kmalloc` based allocators and uses `kmalloc` metadata to determine the memory range reserved for specific pointer values.

To avoid page faults `mpxk_load_bounds` checks (using `virt_addr_valid`) that given pointers are valid and that they belong to a `PageSlab` (i.e., are allocated with `kmalloc`). Undetermined bounds are set to infinite bounds using `__bnd_init_ptr_bounds` for compatibility. This focus on compatibility is practical but does limit the security guarantees provided by MPXK. Because `mpxk_load_bounds` is pointer-based it loads bounds based on a pointer's value, not its address. We discuss the security implications of `mpxk_load_bounds` behavior and current limitations in Section 3.5.

We use the regular MPX compiler options to disable the MPX wrappers, which leaves our GCC-plugin free to insert our wrappers where needed. The wrapper implementations are similar to their MPX equivalents. However,

```
1  void *mpxk_load_bounds(void *ptr)
2  {
3      size_t size;
4
5      do {
6          if (ptr == NULL) {
7              break;
8          }
9
10         if (!virt_addr_valid(ptr)) {
11             break;
12         }
13
14         if (!PageSlab(virt_to_page(ptr))) {
15             break;
16         }
17
18         size = ksize(ptr);
19
20         if (size == 0)
21             return __bnd_null_ptr_bounds(ptr);
22         return __bnd_set_ptr_bounds(ptr, size);
23     } while (0);
24
25     return __bnd_init_ptr_bounds(ptr);
26 }
```

Listing 35: The in-kernel bounds load function checks that the pointer is valid, points into a PageSlab, and then loads the bounds based on `ksize` return value. If any checks fail the returned bounds are set to infinite.

```
1  __attribute__((bnd_legacy)) __attribute__((always_inline))
2  void *
3  __mpxk_wrapper_memmove(void *d, const void *s, size_t c)
4  { return memmove(d, s, c); }
5
6  void *
7  mpxk_wrapper_memmove(void *d, const void *s, size_t c)
8  {
9      if (c == 0)
10         return d;
11
12     __bnd_chk_ptr_bounds(d, c);
13     __bnd_chk_ptr_bounds(s, c);
14
15     return __mpxk_wrapper_memmove(d, s, c);
16 }
```

Listing 36: Our wrapper for `memmove`. The second wrapper, `__mpxk_wrapper_memmove`, ensures that `memmove` is not instrumented, regardless of whether it is a preprocessor macro or a function call.

MPXK does not use the BD and does not need to maintain BD or BT data. For instance, the `memmove` wrapper in user-space (Listing 32) copies any potentially associated BD and BT entries, whereas the `mpxk_wrapper_memmove` (Listing 36) function does not. `mpxk_wrapper_memmove` calls the `memmove` via another wrapper to disable instrumentation of `memmove`, even if it is defined as a macro.

3.3.3 Adapting the MPX Instrumentation

The GCC Pointer Bounds Checker provides much of the instrumentation used by MPXK but does not accommodate our modifications. MPXK relies on the instrumentation to insert bounds checks and do compile-time propagation of bounds (e.g., bounds for local variables that can be stored on the stack). Unfortunately, when needed the MPX instrumentation also uses the Bound Directory (BD), i.e., it inserts `bndstx` and `bndldx` calls. The MPXK instrumentation must remove such calls and instead insert `mpxk_load_bounds` calls where needed. It must also instrument memory altering function calls, i.e., insert MPXK-specific function wrappers.

The straightforward solution would be to directly modify the GCC Pointer Bounds Checker. Due to the complexity of the Linux build process and the GCC implementation we opted to depend on the existing and tested MPX instrumentation. We use the recently added GCC-plugin support in the mainline Linux kernel (Section 3.1.6) to modify the instrumentation for MPXK and therefore do not need direct changes to the compiler. Our MPXK plugin works on both GIMPLE and RTL (see Section 3.1.4) and

consists of four passes (Figure 6).

The `mpxk_wrappers` pass runs before the GCC Pointer Bounds Checker passes and, therefore, allows the Checker to instrument the wrappers themselves. The `execute` function (Listing 37) iterates through all basic-blocks looking for `GIMPLE_CALL` statements, i.e., function calls. It replaces specific functions with wrappers using the `mpxk_wrappers_gimple_call` function. A wrapper retains function arguments and return value assignments, but has a different assembler identifier that points to the wrapper. The wrappers are externally linked, and therefore the instrumentation also sets the `DECL_EXTERNAL` attribute to one.

The `mpxk_cfun_args` pass handles `bndldx` calls in function prologues, i.e., `bndldx` calls for the arguments of the current function. The instrumentation passes bounds either via the four bounds registers or when needed, by using `bndstx` and `bndldx`. The compiler manages this by modifying GIMPLE function calls and definitions to include pointer bounds, e.g., a call to `do(void *ptr)` becomes `do(void *ptr, bounds ptr_bounds)`. The RTL expansion then modifies these abstract `bounds` types to actual registers or `bndstx`/`bndldx` calls. It is specifically the `bndldx` calls that must be removed by MPXK. To achieve this, the `mpxk_cfun_args` implementation (Listing 38) analyzes the function declaration to determine which bounds need replacement and inserts `mpxk_load_bounds` calls for them. Standard GCC behavior removes the `bndldx` call because it notices that `mpxk_load_bounds` immediately overwrites the `bndldx` result.

The `mpxk_bnd_store` pass removes `bndstx` calls and replaces `bndldx` calls with `mpxk_load_bounds`. Compiler builtins, i.e., compiler defined functions, represent these calls in GIMPLE. Therefore, we find them by iterating through all the `GIMPLE_CALL` statements. The argument loads processed by the previous passes cannot be handled here because the function prologue is created during the RTL expansion and is thus not available in GIMPLE. Our design favors GIMPLE because it is high-level and, therefore, easier to manipulate.

The `mpxk_sweeper` pass must work on the RTL representation. The RTL expansion creates register writes for function arguments but also inserts `bndstx` calls where needed (i.e., calls to the functions modified by the `mpxk_cfun_args` pass). This pass removes any such `bndstx` instructions that are introduced by the RTL expansion. The `execute` function (Listing 39) iterates through the RTL basic-blocks looking for `bndstx` instructions. The `contains_unspec` function (Listing 40) is used to perform a recursive search through potentially nested RTL instructions.

MPXK combines these passes into one plugin using the Linux GCC-plugin framework. The plugin is integrated to `Kbuild` and requires no separate compilation or configuration. Only one `CFLAG`, the added `MPXK_PLUGIN` variable, is needed to use MPXK on a specific `Makefile` target. The compile-time `Kconfig` configuration determines whether MPXK is enabled at all.

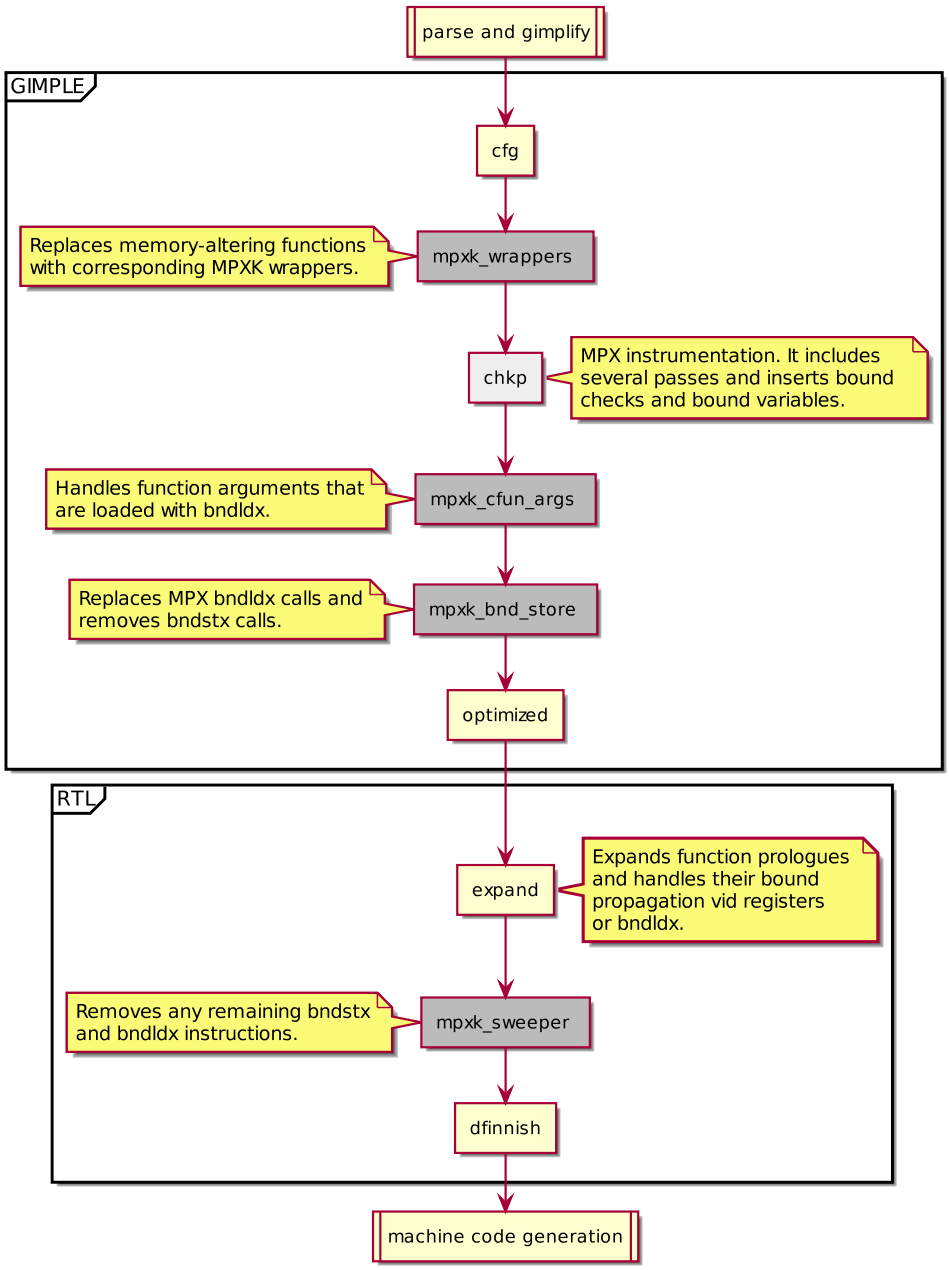

Figure 6: The MPXK compiler passes. The chkp pass performs regular MPX instrumentation and is provided by GCC, our MPXK passes are shown in the darker boxes.

```
1  static unsigned int mpxk_wrappers_execute(void) {
2    gimple_stmt_iterator iter;
3
4    basic_block bb = ENTRY_BLOCK_PTR_FOR_FN(cfun)->next_bb;
5    do {
6        basic_block next = bb->next_bb;
7        for (iter = gsi_start_bb(bb); !gsi_end_p(iter); ) {
8            gimple stmt = gsi_stmt(iter);
9
10           if (gimple_code(stmt) != GIMPLE_CALL)
11               gsi_next(&iter);
12
13           tree fndecl = gimple_call_fndecl(
14                   as_a<gcall *>(stmt));
15           const char *fn_name = DECL_NAME_POINTER(fndecl);
16
17           if (fndecl && mpxk_is_wrappable(fn_name)) {
18               mpxk_wrappers_gimple_call(&iter);
19           }
20
21           gsi_next(&iter);
22       }
23       bb = next;
24   }
25   while (bb);
26
27   return 0;
28 }
```

Listing 37: The `execute` function for the `mpxk_wrappers`, implemented in `scripts/gcc-plugins/mpxk_pass_wrappers.c`. The function iterates through each statement in each basic block of the processed function declaration, and executes the `mpxk_wrappers_gimple_call` function on any function call statements.

```
1  static unsigned int mpxk_cfun_args_execute(void) {
2    int bound_count = 0;
3    int arg_count = 0;
4    basic_block bb = ENTRY_BLOCK_PTR_FOR_FN(cfun)->next_bb;
5    gimple_stmt_iterator iter = gsi_start_bb(bb);
6    tree list = DECL_ARGUMENTS(cfun->decl);
7    tree prev = NULL;
8
9    for (tree *p = &list; *p; ) {
10       tree l = *p;
11
12       /* Keep count of the encountered bounds args */
13       if (TREE_TYPE(l) == pointer_bounds_type_node) {
14           bound_count++;
15
16           iter = gsi_start_bb(bb);
17
18           if (bound_count > 4 || arg_count > 6) {
19               /* Insert MPXK bounds load function */
20               insert_mpxk_bound_load(&iter, prev, l);
21           }
22       } else {
23           prev = l;
24           arg_count++;
25       }
26
27       *p = TREE_CHAIN (l);
28   }
29
30   return 0;
31 }
```

Listing 38: Execute function for our `mpxk_cfun_args` pass, which inserts `mpxk_load_bounds` call in function prologues when needed. This happens when either there are more bounds than bounds registers, or when the bounds are associated with pointer arguments not passed via hardware registers.

```
1  static unsigned int mpxk_sweeper_execute(void) {
2    basic_block bb = ENTRY_BLOCK_PTR_FOR_FN(cfun)->next_bb;
3    do {
4        basic_block next = bb->next_bb;
5        for (rtx_insn *insn = BB_HEAD(bb);
6                insn != BB_END(bb);
7                insn = NEXT_INSN(insn)) {
8
9            if (PATTERN(insn) != NULL &&
10                   contains_unspec(r, UNSPEC_BNDSTX)) {
11               delete_insn(insn);
12           }
13       }
14       bb = next;
15   } while (bb);
16
17   return 0;
18 }
```

Listing 39: The `mpxk_sweeper_execute` function iterates through all instructions in each basic block, removing the instructions that invoke `bndstx`.

```
1  static bool contains_unspec(rtx r, const int code) {
2    enum rtx_code r_code = GET_CODE(r);
3
4    if (r_code == UNSPEC || r_code == UNSPEC_VOLATILE) {
5        if (XINT(r, 1) == code) {
6            return true;
7        }
8    } else if (r_code == PARALLEL || r_code == SEQUENCE) {
9        for (int i = 0; i < XVECLEN(r, 0); i++) {
10           if (contains_unspec(XVECEXP(r, 0, i), code)) {
11               return true;
12           }
13       }
14   } else if (r_code == SET) {
15       if (contains_unspec(SET_SRC(r), code))
16           return true;
17       if (contains_unspec(SET_DEST(r), code))
18           return true;
19   }
20
21   return false;
22 }
```

Listing 40: Recursive function that looks into an RTL instructions to check whether it invokes a specific instrumentation.

3.4 Challenges

3.4.1 Incompatible Source Code

MPX is envisioned as an easy drop-in memory protection system, as such, much of its actual implementation and behavior is undocumented. Unfortunately, it is incompatible with some specific use cases (e.g., many valid pointer uses are in violation of strict pointer bounds semantics). The instrumentation can also be problematic for concurrently running code [Ole+17]. A shared global pointer and its bounds could simultaneously be changed by multiple threads. However because the pointer write and bounds write are separate instructions it is possible that the loading thread ends up with an updated pointer but old bounds, or vice versa. The kernel read-copy-update (RCU) synchronization mechanism, for instance, provides high-performance simultaneous updates but is not compatible with MPX or MPXK. We assume MPX attempts to mitigate concurrency issues by loading bounds registers immediately before their use, i.e., minimizing the potential race window. MPXK uses the same mitigation, but does not solve this problem; it instead exempts incompatible functions from instrumentation (based on current findings, exemptions are needed for the RCU and kernel linked list implementation).

3.4.2 Undocumented and Obtuse MPX Behavior

Some MPXK specific challenges arise from compiler internals and hardware instructions — e.g., caveats in the handling of function argument bounds and `bndldx` behavior when loading modified pointers [Ram+16]. The argument handling, to our knowledge, is an implementation shortcut. The compiler passes the six first arguments through CPU registers and presumably therefore uses the bounds registers only to store the bounds for those six arguments (i.e., `bndldx` is used when there are over four bounds or when bounds belong to the seventh or later argument). Some such implementation details are trivial engineering problems but can be challenging to recognize. Unfortunately, they can also have security implications for MPXK.

The `bndldx` behavior is problematic because if a pointer's value has changed after the `bndstx`, then `bndldx` treats the pointer's bounds as infinite [Ram+16]. This is possible because the BTEs contain not only the bounds but also the original value of the pointer. Unfortunately, while this improves compatibility with legacy code it also means that such pointers cannot be checked. MPXK does not use `bndldx` and instead solves this issue using `mpxk_load_bounds`. The bounds are loaded based on the pointer's value, which means that the bounds can still be acquired for the changed pointer. Note that MPXK does not offer any protection on legacy code operation, and thus does not guarantee that the loaded bounds are correct (i.e., a pointer manipulating attack in legacy code would not be detected upon returning that pointer to the instrumented code).

3.5 Evaluation

The evaluation of MPXK is challenging due to the need to maintain binary compatibility. Binary compatibility inherently imposes some security limitations (Section 3.1.1) that are challenging to evaluate. Furthermore, practical targets for comparison, e.g., KASAN [17j], are monolithic and do not support modularity. We have performed micro-benchmarks comparing MPXK and KASAN. As a test-case, we have also applied MPXK on the `xfrm` subsystem (i.e., the IP transformation subsystem), and measured its performance impact and confirmed it prevents a related CVE.

3.5.1 Security

MPXK prevents spatial memory errors in instrumented code for any pointer with known bounds. The protection is thus limited by whether the bounds are known. Directly instrumented bounds are always known. But dynamic bounds are loaded with `mpxk_load_bounds`, which is limited by the following:

- It currently supports only pointers to memory allocated with `kmalloc` based allocators.
- Loaded bounds are based on a memory area and thus cannot restore narrowed bounds (i.e., a pointer to an array element will get the bounds of the whole array).
- The bounds are loaded based a pointer's value, which differs from the MPX `bndldx` that uses the pointer's address.

The main limiting factor is the current restriction to `kmalloc`, but we expect that this could be improved without major changes to MPXK or the kernel. As future work, we are looking at expanding the load support. Other dynamic memory allocators could be supported similarly to `kmalloc`. Static and stack memory, i.e., global and local variables, do not necessarily have similar data, but could still be bound to specific code regions or stack-frames. The object-based bounds do have some security implications: if an attacker modifies a dynamically stored pointer before it is loaded with `mpxk_load_bounds` the bounds would be loaded based on the modified value, not the unmodified, presumably correct value (Section 3.1.1). However, this requires that the attacker can overwrite a specific pointer and that the pointer is checked against `mpxk_load_bounds`, not static bounds. This object-based load is no worse than MPX, where the pointer bounds would be ignored when `bndldx` recognized the changed value.

To confirm MPXK functionality we have implemented tests using the `lkdtm` (Linux Kernel Dump Test Module) framework. The `lkdtm` tests confirm that the basic instrumentation works, but not whether it is effective against real-world exploits. To gauge this, we tested MPXK against an

exploit of CVE-2017-7184 [cve17], a vulnerability in the `xfrm` subsystem. The attack exploits a missing data-size check, and allows an attacker to gain root privileges; a working exploit was demonstrated at CanSecWest 2017[7]. We confirmed that the exploit works on the Linux v4.8 kernel running the Ubuntu 16.04 operating system. The same setup, but with the `xfrm` subsystem covered by MPXK, resulted in the exploit being successfully detected and stopped. While this is a single case, it shows that MPXK can work even when narrowly applied.

3.5.2 Compatibility

MPXK requires only a single `CFLAG` to enable it on a specific `Makefile` target. However, some valid code can cause compatibility issues through false bound violations. Issues are avoidable using the `bnd_legacy` attribute to disable instrumentation on problematic functions. Intuitively these incompatibilities seem restricted to specific utility libraries, such as the lists discussed in Section 3.4.1, but we have not performed comprehensive tests to confirm this. The `xfrm` test case did not cause any compatibility issues. Kernel-wide testing and code analysis is needed to track down incompatibilities and provide an estimate on compatibility issues but this is left for future work.

3.5.3 Modularity

MPXK is binary compatible and can thus be applied in a modular fashion (Requirement 3). While the hardware itself is globally enabled it does not affect non-instrumented code. To enable MPXK on `xfrm` we modified only one `Makefile` by adding appropriate `CFLAGS`.

3.5.4 Performance

To estimate the performance impact of MPXK we performed two types of measurements. First, we used specifically crafted test code to measure `memcopy` performance differences between non-protected, MPXK and KASAN. Second, we performed Netperf measurements to estimate the impact of applying MPXK on the `xfrm` subsystem. The results for the micro-benchmarks are shown in Table 5. The results include separate test cases for when MPXK needs to load the bounds, and for when the bounds are statically available, i.e., the actual measurements start before the `memcopy` call and end right after it has returned. These were added as separate cases due to the assumption, supported by the data, that the MPXK would be a major contributor to overheads.

The `memcopy` benchmarks are mostly in line with expected results. KASAN, which uses shadow memory, suffers substantial overheads particularly when

[7] `https://cansecwest.com/index.html`

	baseline time (stddev)	KASAN *ns* diff (stddev)	MPXK *ns* diff (stdev)
No bounds load.			
memcpy, 256 B	45 (0.9)	+85 (1.0)	+11 (1.2)
memcpy, 65 kB	2340 (4.3)	+2673 (58.1)	+405 (5.1)
Bounds load needed.			
memcpy, 256 B	45 (0.8)	+87 (0.9)	+70 (1.5)
memcpy, 65 kB	2332 (5.8)	+2833 (28.2)	+475 (15.0)

Table 5: MPXK and KASAN CPU overhead comparison.

large memory areas need updating. It is somewhat surprising that the MPXK overheads differ this much due to the size of the copied area. The results predictably indicate that the `mpxk_load_bounds` incurs some overhead, in this case, an overhead of $65ns$ per call, which in the case of smaller targets is a 155% increase.

For larger scale measurements we used a setup where our test machine was directly connected to another machine and communicated through an IPSec tunnel. The IPSec IP transformations are implemented by `xfrm`, which in turn means that applying MPXK on the `xfrm` framework impacts the performance of the IPSec tunnel. The test machine was equipped with a i36100U processor and 8`GB` memory. The tests were conducted running a Ubuntu 16.04 Server installation with a Linux v4.8 kernel using the default Ubuntu kernel configuration. Each test had a duration of 300s and was repeated three times. Comparisons were made between an MPXK on `xfrm` enabled kernel, and a kernel with MPXK completely disabled via kernel configuration options. The results are shown in Table 6 and indicate a small to negligible performance impact.

We used the same setup to compared memory and size overheads caused by MPXK when enabled on the `xfrm` subsystem; these show a 110`KB` increase

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)%

Table 6: Netperf measurements over an IPSec tunnel with the `xfrm` subsystem protected by MPXK.

for the in-memory size of the kernel, which amounts to a 0.7% increase. The overall size of the kernel image increased by 110KB which is a 0.7% on whole kernel size. When looking at only the `xfrm` targets there was an increase of about 125KB, i.e., an increase of 7%. The memory and binary size will vary depending on kernel configuration and where MPXK is applied but the results, nonetheless, indicate that the size and memory increases are modest. Moreover, they are affected only by added static instrumentation, not accumulating metadata.

One source of performance overhead is the function calls introduced by the instrumentation. For future work, MPXK can be modified to inline both wrapper and `mpxk_load_bounds` functions. This would slightly increase code size but could nonetheless be a good trade-off. The MPXK wrappers are typically small compared to MPX because they do not deal with BD and BT updates. The instrumentation of legacy function calls could also be modified to use the stack as a temporary bounds storage thus completely removing function-call overhead and potential precision loss due to `mpxk_load_bounds`.

3.5.5 Summary of Evaluation

The requirements and evaluations are briefly summarized in Table 7. Both the security and compatibility requirements (Requirements 1 and Requirement 2) are to some extent fulfilled but are subject to some uncertainty. MPXK is fully modular (Requirement 3) and performance overheads are largely negligible (Requirement 4). The `xfrm` test case provides a positive example of MPXK being applied to prevent a specific error, but cannot be generalized.

Req. 1: Security	Limited by modularity and `mpxk_load_bounds`.
Req. 2: Compatibility	Compatibility issues with some use cases.
Req. 3: Modularity	Is completely modular.
Req. 4: Performance	Small CPU use, negligible memory overhead.

Table 7: Evaluation of MPXK against our requirements.

4 Related Work

Linux security is an actively researched topic. Some recent surveys provide insight into Linux kernel vulnerabilities and their causes. A 2008 survey by Mokhov et al. [MLB08] found that faulty error handling and precondition validation are a significant source of security vulnerabilities. Chen et al. [Che+11] found that security measures in the kernel tend to be partial and unable to cover higher-level semantic errors. Reference counting errors often fit into these categories, e.g., it is often high-level errors in the error handling code that lead to omitted reference counter decrements. Palix et al. [Pal+11] in 2011 found that the Hardware Abstraction Layer (HAL) is the major contributor of errors in the Linux kernel. Raheja et al. [RMS16] in 2016 used machine learning techniques to analyzed kernel vulnerabilities. They report that buffer overflows and integer overflows — vulnerabilities that are addressed by MPXK and `refcount_t`, respectively — are the most significant source of memory corruption exploits.

Reference counters: McKenney [McK07] provides a look at reference counting schemes in the Linux kernel. The `kref` type [Kro04] is an early addition that was designed to improve the usability and security of reference counters in the kernel. However, reference counters are not unique to the Linux kernel. They were used as early as 1960 by LISP [McC60] and Collins [Col60].

Exploit mitigations: There have been many efforts to prevent and mitigate memory errors. Stack buffer overflows are mitigated using stack canaries, such as StackGuard [Cow+98] or StackShield [Sta11], that detect manipulated stack data. Other techniques, e.g., avoiding the canary-data, can circumvent stack canaries, although StackShield also incorporates range checks to detect function pointer manipulations. Traditional attacks inject code into writeable memory and can be prevented by `W⊕X` memory schemes that restrict memory areas from being both writeable and executable simultaneously [PaX03b; Sol97b]. However, these techniques are vulnerable to Return Oriented Programming (ROP) attacks, i.e., code-reuse attacks that redirect the control flow into executable memory chosen by the attacker [Sol97a; Kra05]. Memory randomization techniques, such as Address Space Layout Randomization (ASLR) [PaX03a; XKI03], mitigate ROP attacks techniques, but cannot completely prevent them. The mentioned mitigations prevent specific attack techniques but do not address the underlying memory safety issues. Nonetheless, they drastically decrease the attack surface and increase the cost of exploitation.

Memory safety: Memory safety solutions require complete mediation of either pointer dereferences or manipulations [SPW13]. Many solutions have been proposed; some, such as CCured [Nec+05] and Cyclone [Jim+02] used fat-pointers to keep pointers tightly coupled with their respective bounds. CCured is interesting because it, like MPX, attempts to handle bounds during

compile time and uses separate bounds metadata only when absolutely necessary. However, fat-pointers typically break binary compatibility and cause compatibility problems due to changed pointer memory layout (i.e., pointer arithmetic is broken). One notable exception is SGXBounds [Kuv+17]. It is restricted to Intel Software Guard Extension (SGX) enclaves and therefore can embed the bounds into the pointers without changing their memory layout.

An alternative to fat-pointers is to use some separate metadata to store bounds. Object-based systems typically use either shadow-memory or disjoint metadata. Such systems are J&K [JK97], Dhurjati et al. [DA06] and Baggy Bounds Checking [Akr+09]; of which the latter offers the best performance with an average overhead of 60%. Other systems use pointer-based bounds and store bounds in disjoint metadata. Examples are SoftBound [Nag+09] and Intel MPX [Int16] itself. Systems such as Purity [HJ91], PIN [Luk+05], Valgrind [NS07b], and AddressSanitizer [Ser+12] technically also provide pointer bounds checking but are positioned as development and testing tools. Performance overheads and other issues typically make them unsuitable for end-use.

Linux kernel memory safety: There are some Linux kernel specific solutions, including Kernel Address Sanitizer (KASAN) [17j] , the kernel-specific AddressSanitizer implementation. KASAN is conceptually similar to MPXK, i.e., it instruments the code and performs runtime bounds checking, but it also provides probabilistic prevention of use-after-free errors. However, its implementation is different in that it uses a shadow-memory to track object allocations and therefore has high performance overheads. Of recent research projects, kCFI [RPK17] and KENALI [Son+16] both target the Linux kernel. KENALI maintains control flow integrity inside the kernel and achieves good performance by focusing on security-critical portions only. Similarly, kCFI also combines static analysis with runtime enforcement of control flow integrity. Unfortunately, neither is targeting the mainline Linux kernel; kCFI, for instance, is built on LLVM and requires two rounds of compilation.

`PaX/Grsecurity`: Many common security features have been pioneered by `PaX/Grsecurity` [17h]. It has also been a inspiration for the Kernel Self Protection Project (KSPP) and is the source of many features upstreamed by KSPP. `PaX/Grsecurity` provided the initial incentive for dealing with reference counter overflows [Bra15] and first implemented commonly known features such as ASLR [PaX03a]. It is also `PaX/Grsecurity` that initially incorporated the GCC-plugin framework to the Linux kernel. Despite efforts to port specific individual features, the monolithic `PaX/Grsecurity` patch-set has remained firmly separated from the mainline kernel. This separation allows the patch-set to support architectures selectively, do extensive modification to the code-base and disregard, by `PaX/Grsecurity`, unsupported features.

5 Conclusion

This work has provided some tangible security improvements for the Linux kernel. Our `HARDENED_ATOMIC` work served as an instigator to the addition of the `refcount_t`. Beyond that, we contributed to the `refcount_t` design via our kernel-wide analysis and conversion of individual reference counting schemes. With a total of 233 patches, of which about half are already accepted, this work has been substantial. Elimination of reference counter overflows would have prevented severe CVEs in the past, and assuredly makes future exploits more challenging to implement. The MPXK work, in contrast, has been more exploratory but has yielded the novel, kernel-specific, mechanism of using memory management metadata to determine bounds. This allows low overhead pointer checking that is neatly integrated to the `Kbuild` system using newly added GCC-plugin functionality. As a whole, this work has provided concrete improvements to the mainline kernel, interesting insights on kernel memory safety, and yielded new techniques applicable for future work.

References

[16] *Android Developers: Dasboard*. 2016. URL: https://developer.android.com/about/dashboards/ (visited on 11/25/2016).

[17a] *Android Open Source Project*. Oct. 2017. URL: https://source.android.com/ (visited on 10/10/2017).

[17b] *Coccinelle: A Program Matching and Transformation Tool for Systems Code*. 2017. URL: http://coccinelle.lip6.fr/ (visited on 07/28/2017).

[17c] *CVE Details: Linux Kernel Vulnerability Statistics*. 2017. URL: https:///www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 (visited on 05/31/2017).

[17d] *GNU Compiler Collection (GCC) Internals*. 2017. URL: https://gcc.gnu.org/onlinedocs/gccint/ (visited on 10/10/2017).

[17e] *Using the GNU Compiler Collection (GCC): Instrumentation Options*. 2017. URL: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (visited on 10/10/2017).

[17f] *GCC Wiki: Intel Memory Protection Extensions (Intel MPX) support in the GCC compiler*. 2017. URL: https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler/ (visited on 05/24/2017).

[17g] *GNU Project: GCC source*. 2017. URL: https://gcc.gnu.org/gcc-6/ (visited on 10/10/2017).

[17h] *Grsecurity*. 2017. URL: https://grsecurity.net (visited on 10/10/2017).

[17i] *IDC: Smartphone OS Market Share 2016, 2015*. 2017. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 10/10/2017).

[17j] *The Kernel Address Sanitizer (KASAN)*. 2017. URL: http://www.kernel.org/doc/html/v4.10/dev-tools/kasan.html (visited on 10/10/2017).

[17k] *Netperf Homepage*. 2017. URL: https://hewlettpackard.github.io/netperf/ (visited on 10/10/2017).

[AG96] Sarita V Adve and Kourosh Gharachorloo. "Shared Memory Consistency Model: A Tutorial". In: *Computer* 29.12 (Dec. 1996), pp. 66–76. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=546611.

[Akr+09] Periklis Akritidis et al. “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors”. In: *USENIX Security Symposium* (Aug. 2009), pp. 51–66. URL: https://www.usenix.org/legacy/event/sec09/tech/full_papers/akritidis.pdf.

[ARM] ARM. *ARM v8-M Architecture Reference Manual*. URL: https://developer.arm.com/docs.

[Bie17] Ard Biesheuvel. *kernel-hardening mailing list: [PATCH v2] arm64: kernel: implement fast refcount checking*. 2017. URL: http://www.openwall.com/lists/kernel-hardening/2017/07/25/29 (visited on 07/28/2017).

[BL70] Paul Branquart and Johan Lewi. “A scheme of storage allocation and garbage collection for ALGOL 68”. In: *Proceedings of Working Conference (IFIP) on ALGOL 68 Implementation*. 1970, pp. 199–238.

[Bra15] Rodrigo Branco. *Grsecurity forum — Guest Blog by Rodrigo Branco: PAX_REFCOUNT Documentation*. Mar. 2015. URL: https://forums.grsecurity.net/viewtopic.php?f=7&t=4173 (visited on 10/10/2017).

[Che+11] Haogang Chen et al. “Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys ’11. New York, NY, USA: ACM, 2011. URL: https://dl.acm.org/citation.cfm?id=2103805.

[Col60] George E. Collins. “A Method for Overlapping and Erasure of Lists”. In: *Communications of the ACM* 3.12 (Dec. 1960), pp. 655–657. URL: https://dl.acm.org/citation.cfm?id=367501.

[Coo15] Kees Cook. *kernel-hardenign mailing list: Kernel Self Protection Project*. Nov. 2015. URL: http://www.openwall.com/lists/kernel-hardening/2015/11/05/1 (visited on 10/10/2017).

[Coo16] Kees Cook. *Status of the Kernel Self Protection Project*. 2016. URL: https://www.outflux.net/slides/2016/lss/kspp.pdf (visited on 07/28/2017).

[Coo17a] Kees Cook. *codeblog: security things in Linux v4.11*. May 2017. URL: https://outflux.net/blog/archives/2017/05/02/security-things-in-linux-v4-11/ (visited on 10/10/2017).

[Coo17b] Kees Cook. *codeblog: security things in Linux v4.13*. Sept. 2017. URL: https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/ (visited on 10/10/2017).

[Coo17c] Kees Cook. *LKML: PATCH v7 0/3] x86: Implement fast refcount overflow protection.* 2017. URL: https://lkml.org/lkml/2017/7/23/139 (visited on 07/28/2017).

[Cor10] Jonathan Corbet. *lwn.net: Kernel vulnerabilities: old or new?* Oct. 2010. URL: https://lwn.net/Articles/410606/ (visited on 10/10/2017).

[Cor11] Jonathan Corbet. *lwn.net: Better kernels with GCC plugins.* 2011. URL: https://lwn.net/Articles/461696/ (visited on 10/10/2017).

[Cor16] Jonathan Corbet. *lwn.net: Kernel building with GCC plugins.* June 2016. URL: https://lwn.net/Articles/691102/ (visited on 10/10/2017).

[Cow+98] Crispin Cowan et al. "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks". In: *USENIX Security Symposium* 98 (Jan. 1998), pp. 63–78. URL: https://dl.acm.org/citation.cfm?id=1267554.

[cve14a] *CVE-2014-2851.* 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2851 (visited on 05/31/2017).

[cve16a] *CVE-2016-0728.* 2016. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728 (visited on 11/01/2016).

[cve16b] *CVE-2016-4558.* 2016. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4558 (visited on 11/01/2016).

[cve17] *CVE-2017-7184.* 2017. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7184 (visited on 09/01/2017).

[DA06] Dinakar Dhurjati and Vikram Adve. "Backwards-compatible array bounds checking for C with very low overhead". In: *Proceeding of the 28th international conference on Software engineering.* ICSE '06. ACM, 2006, pp. 162–171. URL: https://dl.acm.org/citation.cfm?id=1134309.

[Edg13] Jake Edge. *lwn.net: Kernel address space layout randomization.* https://lwn.net/Articles/569635/. 2013.

[Edg15] Jake Edge. *lwn.net: Two PaX features move toward the mainline.* https://lwn.net/Articles/668876/. Dec. 2015.

[Hen+93] Laurie Hendren et al. "Designing the McCAT compiler based on a family of structured intermediate representations". In: *Languages and Compilers for Parallel Computing: 5th International Workshop New Haven, Connecticut, USA, August 3–5, 1992 Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 406–420. URL: https://doi.org/10.1007/3-540-57502-2_61.

[HJ91] Reed Hastings and Bob Joyce. “Purify: Errors of Memory Leaks and Access Fast Detection”. In: *In Proc. of the Winter 1992 USENIX Conference*. 1991, pp. 125–136. URL: https://courses.cs.washington.edu/courses/cse484/14au/reading/purify.pdf.

[Int16] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. 2016.

[Jim+02] Trevor Jim et al. “Cyclone: A safe dialect of C”. In: *USENIX Annual Technical Conference* (2002), pp. 275–288. URL: https://dl.acm.org/citation.cfm?id=647057.713871.

[JK97] Richard W M Jones and Paul H J Kelly. “Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs.” In: *Proceedings of the 3rd International Workshop on Automatic Debugging*. Vol. 1. AADEBUG-97. Linköping University Electronic Press; Linköpings universitet, 1997, pp. 13–26. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.7027.

[Kra05] Sebastian Krahmer. *X86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique*. Sept. 2005. URL: http://forum.ouah.org/no-nx.pdf (visited on 08/10/2017).

[Kro04] Greg Kroah-Hartman. “kobjects and krefs: lockless reference counting for kernel structures”. In: *Proceedings of the Linux Symposium*. Vol. 2. July 2004, pp. 295–300. URL: https://www.kernel.org/doc/ols/2004/ols2004v2-pages-9-14.pdf.

[Kuv+17] Dmitrii Kuvaiskii et al. “SGXBounds: Memory Safety for Shielded Execution”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys ’17. ACM, Apr. 2017, pp. 205–221. URL: https://dl.acm.org/citation.cfm?id=3064192.

[Lam05] Christoph Lameter. *LKML: Re: [PATCH] atomic_long_t & include/asm-generic/atomic.h V2*. 2005. URL: https://lkml.org/lkml/2005/12/14/2 (visited on 10/10/2017).

[Lin] Linux Programmer’s Manual. *stdarg(3) Linux Programmer’s Manual*. URL: http://man7.org/linux/man-pages/man3/stdarg.3.html (visited on 05/31/2017).

[Luk+05] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 40. PLDI ’05. ACM, 2005, pp. 190–200. URL: http://dl.acm.org/citation.cfm?id=1065010.1065034.

[McC60] John McCarthy. “Recursive Functions of Symbolic Expression and their Computation by Machine”. In: *Communication of the ACM* 3 (Apr. 1960), pp. 184–195. URL: https://dl.acm.org/citation.cfm?id=367199.

[McK07] Paul E McKenney. *Overview of Linux-Kernel Reference Counting*. Tech. rep. Linux Technology Center, IBM Beaverton, 2007. URL: http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2007/n2167.pdf.

[MLB08] Serguei A. Mokhov, Marc André Laverdière, and Djamel Benredjem. “Taxonomy of Linux kernel vulnerability solutions”. In: *Innovative Techniques in Instruction Technology, E-Learning, E-Assessment, and Education*. Ed. by Magued Iskander. Springer Netherlands, 2008, pp. 485–493. URL: https://doi.org/10.1007/978-1-4020-8739-4_86.

[Nag+09] Santosh Nagarakatte et al. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009, pp. 245–258. URL: http://doi.acm.org/10.1145/1542476.1542504.

[Nec+05] George C. Necula et al. “CCured: type-safe retrofitting of legacy software”. In: *ACM Transactions on Programming Languages and Systems* 27.3 (2005), pp. 477–526. URL: http://portal.acm.org/citation.cfm?doid=1065887.1065892.

[NS07a] Nicholas Nethercote and Julian Seward. “How to Shadow Every Byte of Memory Used by a Program”. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. VEE ’07. ACM, June 2007, pp. 65–74. URL: https://dl.acm.org/citation.cfm?id=1254820.

[NS07b] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, June 2007, pp. 89–100. URL: http://doi.acm.org/10.1145/1250734.1250746.

[Ole+17] Oleksii Oleksenko et al. “Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches”. In: *ArXiv e-prints* (Feb. 2017). URL: https://arxiv.org/abs/1702.00719.

[Pad+08] Yoann Padioleau et al. “Documenting and automating collateral evolutions in linux device drivers”. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. Vol. 42. Eurosys ’08. 2008, pp. 247–260. URL: http://dl.acm.org/citation.cfm?id=1357010.1352618.

[Pal+11] Nicolas Palix et al. “Faults in Linux : Ten Years Later Faults in Linux : Ten Years Later”. In: *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*. ASPLOS XVI. 2011, pp. 305–318. URL: https://dl.acm.org/citation.cfm?id=1950401.

[PaX03a] PaX Team. *PaX address space layout randomization (ASLR)*. 2003. URL: http://pax.grsecurity.net/docs/aslr.txt (visited on 10/10/2017).

[PaX03b] PaX Team. *PaX non-executable pages design & implementation*. 2003. URL: http://pax.grsecurity.net (visited on 08/10/2017).

[Pax17] Pax Team. *kernel-hardening mailing list: Re: It looks like there will be no more public versions of PaX and Grsec*. 2017. URL: http://www.openwall.com/lists/kernel-hardening/2017/05/11/2 (visited on 10/10/2017).

[PF95] Harish Patil and Charles N. Fischer. “Efficient run-time monitoring using shadow processing”. In: *Automated and Algorithmic Debugging*. Vol. 95. AADEBUG’95. 1995, pp. 119–132. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2066.

[PF97] Harish Patil and Charles Fischer. “Low-cost, Concurrent Checking of Pointer and Array Accesses in C Programs”. In: *Software: Practice and Experience* 27.1 (1997), pp. 87–110. URL: https://dl.acm.org/citation.cfm?id=250910.

[Ram+16] Ramu Ramakesavan et al. *Intel® Memory Protection Extensions Enabling Guide Rev 1.01*. 2016.

[Res16a] Elena Reshetova. *Conversion from atomic_t to refcount_t: summary of issues*. Nov. 2016. URL: http://www.openwall.com/lists/kernel-hardening/2016/11/28/4 (visited on 10/10/2017).

[Res16b] Elena Reshetova. *kernel-hardening mailing list: [RFC PATCH 00/13] HARDENING_ATOMIC feature*. 2016. URL: http://www.openwall.com/lists/kernel-hardening/2016/10/03/1 (visited on 10/10/2017).

[Res16c] Elena Reshetova. *kernel-hardening mailing list: [RFC v2 PATCH 00/13] HARDENING_ATOMIC*. 2016. URL: http://www.openwall.com/lists/kernel-hardening/2016/10/20/5 (visited on 10/10/2017).

[Res16d] Elena Reshetova. *kernel-hardening mailing list: [RFC v3 PATCH 00/13] HARDENING_ATOMIC feature*. 2016. URL: http://www.openwall.com/lists/kernel-hardening/2016/10/31/5 (visited on 10/10/2017).

[Res16e] Elena Reshetova. *kernel-hardening mailing list: [RFC v4 PATCH 00/13] HARDENING_ATOMIC*. 2016. URL: http://www.openwall.com/lists/kernel-hardening/2016/11/10/4 (visited on 10/10/2017).

[RMS16] Supriya Raheja, Geetika Munjal, and Shagun. "Analysis of Linux Kernel Vulnerabilities". In: *Indian Journal of Science and Technology* 9.48 (2016). URL: http://www.indjst.org/index.php/indjst/article/view/105819.

[RPK17] Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. *DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel*. 2017. URL: 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 (visited on 10/10/2017).

[Ser+12] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX Annual Technical Conference*. June 2012, pp. 309–318. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[Sha07] Hovav 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*. Vol. 22. 4. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. URL: http://portal.acm.org/citation.cfm?id=1315313.

[Sol97a] Solar Designer. *Bugtraq mailing list: Getting around non-executable stack (and fix)*. 1997. URL: http://seclists.org/bugtraq/1997/Aug/63 (visited on 10/10/2017).

[Sol97b] Solar Designer. *Bugtraq mailing list: Linux kernel patch to remove stack exec permission*. 1997. URL: http://seclists.org/bugtraq/1997/Apr/31 (visited on 10/10/2017).

[Son+16] Chengyu Song et al. “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.

[SPW13] László Szekeres, Mathias Payer, and Tao Wei. “SoK: Eternal War in Memory”. In: *SP ’13 Proceedings of the 2013 IEEE Symposium on Security and Privacy*. Vol. 12. 3. May 2013, pp. 48–62. URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6824529`.

[Sta11] StackShield. *A Stack Smashing Technique Protection Tool for Linux*. 2011. URL: `http://www.angelfire.com/sk/stackshield` (visited on 10/10/2017).

[Tor] Torvalds, Linus, et al. *Linux kernel source*. `https://kernel.org`.

[Win16] David Windsor. *kernel-hardening mailing list: HARDENED_ATOMIC benchmarks*. July 2016. URL: `http://www.openwall.com/lists/kernel-hardening/2016/10/22/1` (visited on 07/31/2017).

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. “Transparent runtime randomization for security”. In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. 2003, pp. 260–269. URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1238076`.

[Zab17] Philipp Zabel. *LKML: [PATCH v5 1/6] reset: use kref for reference counting*. 2017. URL: `https://lkml.org/lkml/2017/6/1/762` (visited on 10/19/2017).

[Zij17] Peter Zijlstra. *Patchwork [tip:locking/core] refcount_t: Introduce a special purpose refcount type*. 2017. URL: `https://patchwork.kernel.org/patch/9569859/` (visited on 10/10/2017).

A Appendix

A.1 Coccinelle patterns

Listing 41: 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)
```

A.2 `refcount_t` API

`void refcount_set(refcount_t, unsigned int)`

`unsigned int refcount_read(refcount_t)`

`void refcount_add(unsigned int, refcount_t)`
Add value to `refcount_t` unless it was 0, in which case the addition is aborted and a warning is issued. Intended for use when the `refcount_t` value is known to be positive, hence the warning when it is not.

`bool refcount_add_not_zero(unsigned int, refcount_t)`
Add value to `refcount_t` unless the value before was 0 and return true if the addition was done.

`void refcount_inc(refcount_t)`other_func
Increment `refcount_t` by one unless it already is zero, in which case a warning issues. Architecture independent version is implemented using the `refcount_inc_not_zero` function.

`bool refcount_inc_not_zero(refcount_t)`
Increment `refcount_t` unless its value was zero, returns true on increment. Note that this function also returns false when the value is saturated, although a warning is issued when the saturation takes place.

`bool refcount_sub_and_test(unsigned int, refcount_t)`
Subtract value from `refcount_t`. If the subtraction overflows a warning is issued. True is returned only if the `refcount_t` value reaches zero due to the subtraction. Note that a saturated value incurs neither warnings nor value changes, it simply stays at the saturation value, `UINT_MAX`.

`bool refcount_dec_and_test(refcount_t)`
: Analogous to `refcount_sub_and_test`, the architecture independent implementation is even implemented as `refcount_sub_and_test(1, refcount)`.

`void refcount_dec(refcount_t)`
: Decrement the `refcount_t` value by one and issue a warning if the `refcount_t` reaches zero.

`bool refcount_dec_if_one(refcount_t)`
: Decrement the `refcount_t` only if it was 1, i.e., either swaps the value from 1 to 0, or does nothing. The return value is true if the swap succeeded. While this function is not strictly needed for ideal reference counting schemes it is in practice needed to accommodate many existing reference counting schemes.

`bool refcount_dec_not_one(refcount_t)`
: Decrement the `refcount_t` only if it's value is other than 1, i.e., decrement only when the resulting value will remain greater than zero.

`bool refcount_dec_and_mutex_lock(refcount_t, struct mutex *)`
: Decrements the reference counter and locks the provided mutex before invoking the decrement operation. If the decrement caused the counter to reach zero the mutex remains locked and `true` is returned, otherwise the mutex is unlocked and `false` is returned.

`bool refcount_dec_and_lock(refcount_t, spinlock_ *)`
: Identical to the `refcount_dec_and_mutex_lock` function, but instead of a mutex this variant uses a spinlock. The difference being that a mutex causes blocked CPUs to sleep wherease the spinlock essentially keeps the CPU spinning and checking the lock continuously.