Improving Security at the System-Call Boundary in a Type-Safe Operating System

by Jakob H. Weisblat

S.B., C.S., M.I.T., 2018

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science and Engineering

at the

Massachusetts Institute of Technology

February 2019

©2018 Jakob H. Weisblat. All rights reserved.

This author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: ______________________________
Department of Electrical Engineering and Computer Science
September 28, 2018

Certified by: ______________________________
Howard Shrobe, Principal Research Scientist and Director CyberSecurity@CSAIL, Thesis Co-Supervisor
September 28, 2018

Certified by: ______________________________
Hamed Okhravi, Senior Staff, Cyber Analytics and Decision Systems, MIT Lincoln Laboratory, Thesis Co-Supervisor
September 28, 2018

Certified by: ______________________________
Bryan Ward, Technical Staff, Cyber Analytics and Decision Systems, MIT Lincoln Laboratory, Thesis Co-Supervisor
September 28, 2018

Accepted by: ______________________________
Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Distribution Statement

Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

Improving Security at the System-Call Boundary in a Type-Safe Operating System

by Jakob H. Weisblat

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Computer Science and Engineering

Abstract

Historically, most approaches to operating sytems security aim to either protect the kernel (e.g., the MMU) or protect user applications (e.g., $W \oplus X$). However, little study has been done into protecting the boundary between these layers. We describe a vulnerability in Tock, a type-safe operating system, at the system-call boundary. We then introduce a technique for providing memory safety at the boundary between userland and the kernel in Tock. We demonstrate that this technique works to prevent against the aforementioned vulnerability and a class of similar vulnerabilities, and we propose how it might be used to protect against simliar vulnerabilities in other operating systems.

Acknowledgements

Thanks to Bryan Ward for reviewing this document a number of times, helping me phrase the work done here in the best way possible, helping me think about how to evaluate this work, and generally offering large amounts of advice.

Thanks to Howie Shrobe for letting me wander into his office and bounce ideas off of him repeatedly, and for offering wisdom when I have done so.

Thanks to Hamed Okhravi, Howie Shrobe, Bryan Ward, and Richard Skowyra, for helping formulate the ideas that led to this work and for their frequent advice and help on all aspects of this project, as well as for teaching me so much over the course of my time working on it.

Thanks to Hamed Okhravi and Howie Shrobe for the opportunity to work in this group.

Thanks to Jennifer Switzer, for her help and collaboration on various aspects of this project.

Thanks to Ivan Tadeu Ferreira Antunes Filho, Jennifer Switzer, Maddie Dawson, and Teddy Katz, for their help understanding parts of Tock and for their debugging help.

Thanks to Brendan Yap, Dory Shen, and Lucy Wang for their help writing and running tests to evaluate this work, as well as for helping me formulate my ideas by asking me questions when what I told them didn't make any sense.

Thanks to Amit Levy of the Tock project for providing insight into Tock and help with debugging.

Finally, I'd like to thank Ben Rosen-Filardo for their emotional support, especially during the particularly stressful periods, my parents for their encouragement, and all of my friends for helping me with advice, support, and companionship.

Contents

1 Introduction **1**

2 Background **5**

2.1 Memory Safety 6

2.2 Rust 8

2.3 Operating Systems 11

2.4 Tock 12

3 Related Work **15**

3.1 SAFE 15

3.2 CHERI 16

3.3 Singularity 18

3.4 System-call Approaches 19

4 Threat Model **19**

5 Example Vulnerability **21**

6 Methods **24**

6.1 Shadow Memory and Type Tracking 25

6.2 System-Call Handler Modifications 27

6.3 User-Mode Component 29

7 Evaluation **30**

7.1 Performance 30

7.1.1 Microbenchmarks 31

7.1.2 Macrobenchmarks 33

7.1.3 Synthetic Benchmark 35

7.1.4 Memory use 36

7.2 Security Evaluation 36
7.3 Broader Effects 37

8 Discussion **38**
8.1 Performance Overhead 39
8.2 Compatibility 39
8.3 Temporal Safety 40
8.4 Mutability 42
8.5 Extra System-Call 43

9 Conclusion **43**

List of Figures

1 The gap between the kernel and user programs 1

2 Pointer base and bound 3

3 System calls 5

4 (a) Memory layout on a stack. (b) Stack smashing in C. 7

5 Tock Architecture, taken from [23] 13

6 System Calls in Tock 14

7 Our Threat Model 20

8 Syscall architecture in Modified Tock 24

9 Memory layout modification 26

10 Example of Shadow Memory Contents 28

11 Parts of a ALLOW system call under our system 32

12 CDF of times for an ALLOW syscall on 100 bytes of data. Yellow is vanilla Tock, blue is our sytem. 34

13 CDF of times for setting up an RNG transaction on 24 bytes of data. Yellow is vanilla Tock, blue is our sytem. 35

14 Time for an ALLOW operation vs size of buffer. Blue is modified tock, Yellow is vanilla. 36

List of Tables

1 Parts of an ALLOW Syscall under our system. Bolded steps take more substantial time. 32

2 Timing for parts of an ALLOW syscall, all numbers in ticks . . . 33

3 Timing on Microbenchmarks of modified Tock syscall interface . 33

4 Timing on Macrobenchmarks of modified Tock syscall interface . 35

1 Introduction

Historically, there are two broad research areas in systems security; protecting the *kernel* [19, 42, 39] - the priveleged part of the operating system that facilitates all the rest of the system's functionality, including scheduling and memory management - and protecting userspace programs [1, 5], which generally are doing the important work of the computer and where important data resides. However, comparatively less study has been done into protecting the interface between these layers. Even when a system is made up of components that are independently secure, vulnerabilities can still be induced by the way that these components interact with one another - in an operating systems context, these vulnerabilities appear in the locations indicated by the red circles in Figure 1.

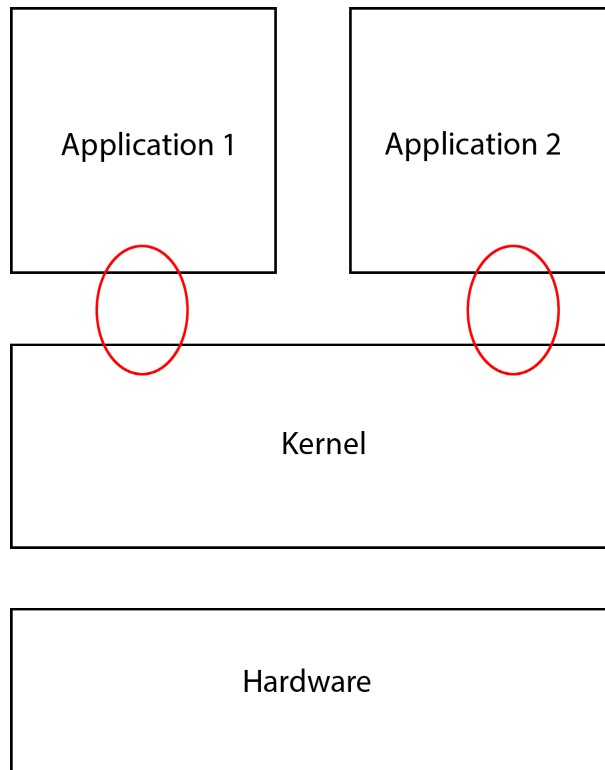

Figure 1: The gap between the kernel and user programs

A large class of attacks in systems security fall into the category of *memory corruption*. Memory corruption occurs when attackers gain access to memory they are not supposed to be able to access, giving them the ability to cause the system to behave in unexpected ways. In systems security, we try to protect against these attacks, sometimes by building systems that obey *memory safety*. Memory safety, as we define it, is the property that a system is not vulnerable to a certain class of *spatial* and *temporal* memory violations. A *spatial violation* occurs when a pointer is used to access a region of memory that it is not defined to point to; for example, if a pointer exists to a buffer in memory, it would be a violation for that pointer to access or overwrite data in an adjacent buffer (*buffer overflow*). See, for example, Figure 2; the base and bound for the `english_alphabet` pointer are shown; if spatial memory safety is not enforced, one could attempt to access `english_alphabet[28]` and instead get a Greek letter. A *temporal violation* occurs when a pointer is used outside of the lifetime during which it is defined; for example, if a pointer is accessed after it has been freed (*use-after-free*), the data it pointed to may not be there anymore. Worse yet, the data structure might have been allocated again, thus allowing the old pointer to corrupt data now serving a different purpose. If a system can violate neither spatial nor temporal safety, we label it *memory-safe*.

Memory safety is impractical to achieve in C [30], the language in which the most systems code has been written over the last 40 years. Pointers to single objects are very easily conflated with arrays, and they have no manifest bounds, so they could point anywhere and it is often not clear whether a given reference is a violation. Memory management is mostly manual, making temporal safety difficult. There are nearly eighty different ways to invoke undefined behavior [16]. In part because memory safety in C is hard, programming languages with better guarantees, through more complex type systems and/or garbage collection,

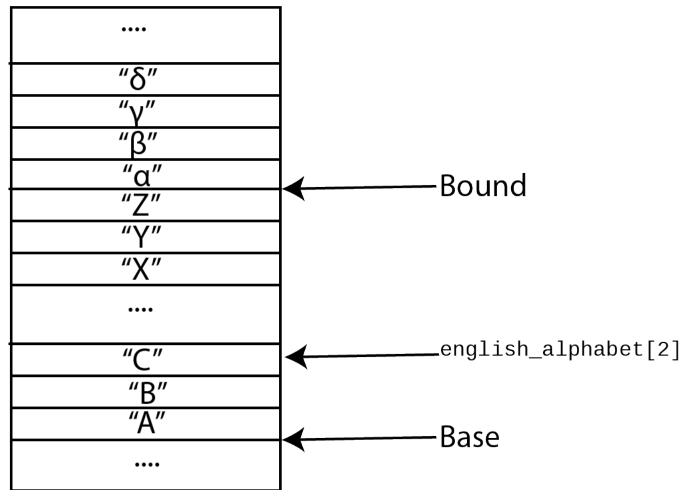

Figure 2: Pointer base and bound

have been developed. Recently, Go and Rust have been popular among systems programmers because they provide safety guarantees while still having comparably small runtime overhead, compared to older safe languages like Java. In particular, Rust is seeing increasing use due to its low runtime overhead: it is comparably performant with C, while making it much harder to violate memory safety [29].

Many of the innovations that improve memory safety, either in C or in more modern langauges, work by performing extensive analysis at compile time and using the results of this analysis to guarantee that memory safety is maintained. When such guarantees are impossible, the compiler generates runtime checks in the compiled code that dynamically check for memory safety violations [34]. The compiler only has the ability to add these checks and ensure safety because it has information provided by the programmer at compile time about what the program is supposed to be doing; without that compile-time data the guarantees these systems provide would not be feasible.

One recent application of memory-safe languages such as Rust is in operating

system kernels. Modern operating systems are generally structured as in figure 1. The kernel is responsible for managing resources and scheduling the other processes. It has direct access to the hardware, running in a priveleged mode defined by the hardware. The kernel also keeps track of user-mode applications running on the computer, their memory allocations, and their communication with hardware. Because the kernel often has access to all of memory and runs in a priveleged mode, it is often a target for attackers, so it is critical that the kernel be secure. One approach, that taken by Tock, is to implement the kernel in a memory-safe language - in this case, Rust. However, in an operating system context, all of the software is rarely compiled at once - the kernel is compiled and then the user programs are compiled separately and later. If these pieces of software are used in conjunction, any data going between them may not be subject to the guarantees provided by compile-time analysis.

User applications running under the kernel need a means to communicate with the kernel. For example, an application may need to interact with the hardware or the network, or it may need additional memory. The user application and the kernel run in different processor modes, and they have different memory spaces, as a means to protect the kernel memory from mailicious user code. As a result of these separations, the application cannot simply call a function in the kernel. Instead, the kernel defines an API for the user process to contact it; this API is called the *system call interface*.

Traditionally, system calls are implemented through the process shown in Figure 3: the user process puts data in a predesignated place - usually in registers or sometimes on the stack - and then triggers a specific mechanism to cause the kernel to take control of the processor execution (sometimes an interrupt, sometimes a special instruction). The kernel will then load the data from where the user process left it, perhaps including a pointer to a larger chunk of user

memory, upon which the kernel will operate.

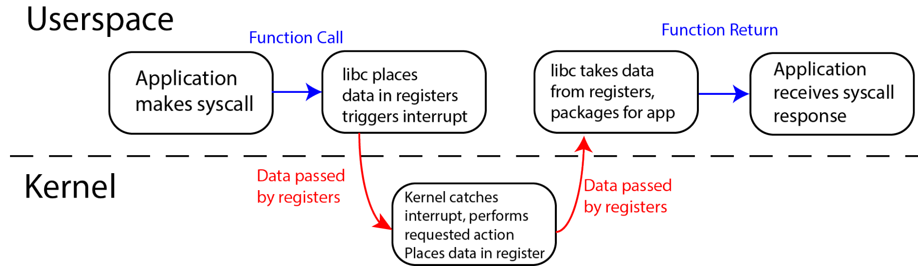

Figure 3: System calls

Recall that the memory safety of the kernel is enforced as a result of compile-time analysis of the entire kernel as a single unit. At the time the kernel takes data from the user application, information that was available when the user code was compiled is no longer available, and so the kernel is no longer able to provide its normal safety guarantees.

When designing a memory-safe operating system, the designer must consider both the security of the individual components and the way that they connect to one another. Even if their individual compilation can provide local safety guarantees, the fact that they are not compiled as a unit means these guarantees can be broken at this interface. In order to achieve memory safety across all layers of a system, we need interfaces like the system call boundary to be implemented in such a way as to ensure memory safety. In this thesis, we have demonstrated that these vulnerabilities exist in a memory-safe operating system (Section 5), proposed and implemented a solution (Section 6), and evaluated that solution (Sections 7 and 8).

2 Background

In order to better understand the work described here, we first discuss Rust at a high level and understand how it enforces memory safety. We will describe

Tock, the operating system upon which this work is based; we describe how the system call works in Tock, and how its system-call interface can be a source of vulnerabilities despite compiler-based memory safety properties.

2.1 Memory Safety

As defined in Section 1, a system is *memory-safe* if it is vulnerable to neither spatial nor temporal memory access violations; equivalently, that system has *memory safety*. In this section we will explore in more depth what some violations of memory safety look like, and we will examine some enforcement mechanisms.

In order to better understand memory safety, we need to understand how data is laid out in memory. In most programs, some data is allocated on a *stack*, as depicted in Figure 4(a): each variable is allocated in the order they are declared, with little or no wasted space between. The code shown in Listing 1 exploits that fact. The user is able to enter a password longer than the buffer allocated for the purpose on line 3. If the user enters the password `‘‘12345671234567’’`, the program will write all 14 characters of data to the variable `attempt`, breaking the abstraction that all chunks of memory are separate, and overwriting the value of `correct`, leading to the stack values shown in Figure 4(b). When the program compares the first 6 characters of `attempt` with `secret` on line 7, it will find that they are equal and grant access. The program had a spatial memory violation, wherein data overflowed from one buffer into another. This kind of attack is called a buffer-overflow attack, and it is very common in languages that are not memory-safe [21]. A similar type of vulnerability, with a malicious read from a buffer instead of a malicious write, is called a buffer-over-read. One well-known such vulnerability was HeartBleed [13], a vulnerability that affected over 5% of the top million sites on the internet, including

as many as 50% of the top sites in Korea and Japan [15].

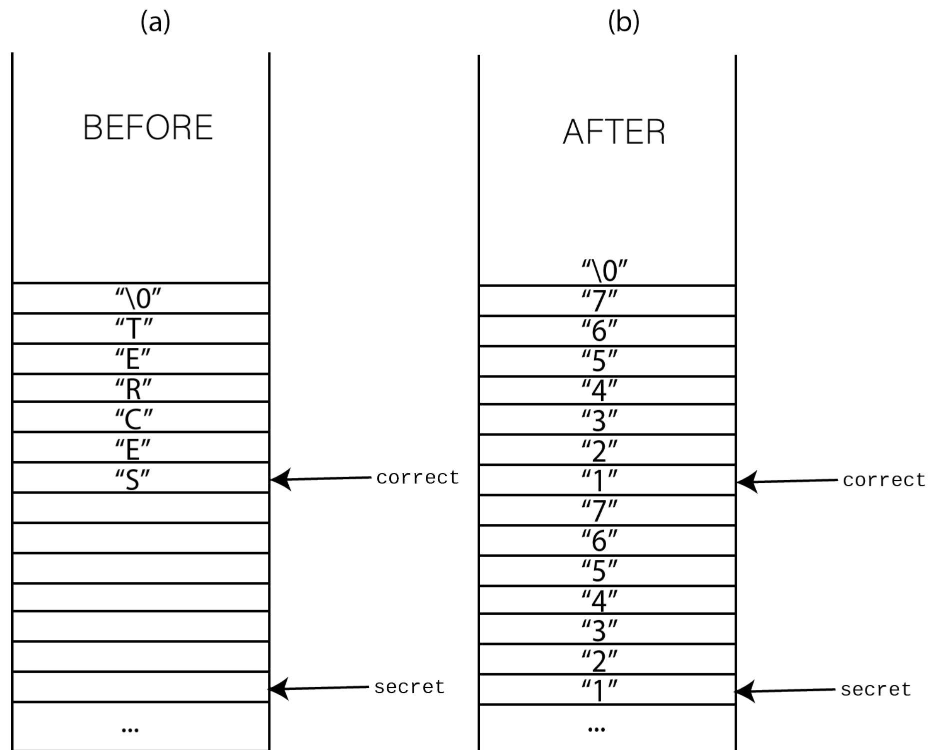

Figure 4: (a) Memory layout on a stack. (b) Stack smashing in C.

Listing 1: Stack smashing in C

```
1 bool check_access(){
2     char correct[7] = "SECRET";
3     char attempt[7]; // allocate a buffer for the user to write to
4 
5     printf(‘‘Enter your password: ’’);
6     scanf(‘‘\%s’’,str);  // let the user write to that buffer
7     return strncmp(attempt, correct, 6));
8 }
```

Buffer-overflow attacks are not the only common vulnerability in C and C++ programs; vulnerabilities range from user-after-free bugs in which the programmer uses a pointer that could now point to a different piece of data,

to undefined behaviors that could be accidentally activated, even by optimizing code that appears not to contain undefined behavior [36]. In C code, buffer overflows and other memory-layout and memory-management vulnerabilities are sufficiently common to persistently cause problems in popular software.

Compounding this problem, much of the systems code written today, and an overwhelming majority of the systems code written in the last 30 years, is written in C. The C/C++ programming language is used in the most popular web servers (e.g., Apache), OS kernels (e.g., Linux), and web browsers (e.g., Chrome), but it lacks safety in many ways. For one, there is no bounds-checking on pointers, which are used for purposes ranging from arrays/buffers to imitating object-oriented programming, and are a basic building-block of C programming. For another, there are a number of undefined behaviors (e.g., signed integer overflow) that are difficult to avoid while adhering to the C standard - and they are implemented differently by different compilers. Several existing “solutions” that try to make C code memory-safe [2, 31] have been proposed, but they have been repeatedly shown to be incomplete [14, 34].

More recently, much work on programming language design has been done, and thankfully we now have a plethora of better and safer choices in which to write software; e.g., Java, C#, Go, Rust. Of particular interest to systems progreamming in recent years have been Golang and Rust, which have been developed by Google and Mozilla respectively specifically for large highly parallel workflows, with security in mind. In this thesis, we focus on Rust.

2.2 Rust

Rust is a language developed by Mozilla to make it easier to parallelize the rendering engine in Firefox. By providing memory safety it allows parallel threads to be implemented safely [29]. Rust has several features designed to provide

memory safety at compile-time. In order to achieve spatial safety, Rust disallows raw pointers and requires that pointers be encapsulated in a *slice*, which has a base and bound, or in other simliar constructions allowed by its capable type sytem.

In order to ensure temporal safety, the more complicated of the two components of memory safety [31], Rust has a novel memory management system, as demonstrated in Listing 2: every variable has an owner, and exactly one owner. When a variable's owner goes out of scope, that variable is no longer accessible, its memory is dropped. What makes this complicated is that it is difficult to ensure that a variable has only one owner at a time. Rust enforces the property that any value can only have one mutable reference at a time; when that reference is no longer accessible, the value is freed. This ownership system makes it easy for the compiler to enforce temporal safety: the compiler is always aware at compile-time[1] of where any given piece of memory is in scope, so temporal violations and data races are prevented without runtime overhead.

Listing 2: Ownership in Rust example taken from [18]

```
fn main() {
    let s = String::from(‘‘hello’’);  // s comes into scope

    takes_ownership(s);  // s’s value moves into the function...
                         // ... and so is no longer valid here

    let x = 5;           // x comes into scope

    makes_copy(x);       // x would move into the function,
                         // but i32 is Copy, so it’s okay to still
                         // use x afterward

} // Here, x goes out of scope, then s. But because s’s value was
  // moved, nothing special happens.

```

[1] Rust does have reference-counted variables as an option, but they are not used frequently by most rust code, and there is runtime code to enforce themporal safety on them

```
16 fn takes_ownership(some_string: String) { // some_string into scope
17     println!(``{}'', some_string);
18 } // Here, some_string goes out of scope and `drop` is called.
19   // The backing memory is freed.
20
21 fn makes_copy(some_integer: i32) { // some_integer into scope
22     println!(``{}'', some_integer);
23 } // Here, some_integer goes out of scope. Nothing special happens.
```

It is very difficult, however, to implement a complex system in Rust while staying entirely within all the constraints imposed by this type system [25]; it is impossible, for example, to implement a doubly linked list, because in a circular reference there cannot be a single owner who owns everything else. Because these constraints are hard to work under, Rust provides another option: the programmer may declare a region of code as *unsafe*, thereby giving it permission to break some rules. An *unsafe* block of code is allowed to manipulate raw pointers, create multiple mutable references to the same region of memory, and perform other unchecked operations. Rust programmers using a library expect that an unsafe block will be *externally* safe. For example, the double-ended queue in the standard library, `std::collections::VecDeque`, uses unsafe code to implement a circular buffer, but it encapsulates that unsafety. A programmer can use a `VecDeque` without fear of corrupting memory, even though internally the `VecDeque` implementation manipulates raw pointers.

It is important to note that there is no *enforcement* that all unsafe code is properly encapsulated. A bug in an unsafe region could create multiple references to the same piece of data, and those references would continue to exist outside of the unsafe block and wherever they were passed; it could create a pointer of the wrong type, and the rest of the code would try to manipulate that data as if it were the wrong type. For a simple example, see Listing 3; the function called on line 16 is safe but can corrupt whatever memory is being passed to it from the unsafe `get_data()`. Thus, it is important that unsafe

code be written very carefully, or else the whole program may not obey Rust's guarantees.

Listing 3: Unsafe code in Rust can cause problems for safe code

```
fn get_data() -> &'static mut [usize] {
    use std::slice;
    let ptr = 0x1234 as *mut usize;
    let amt = 10;
    unsafe {
        return slice::from_raw_parts_mut(ptr, amt);
    }
}

fn main() {
    // get_data() makes a slice that points to arbitary memory.
    let data = get_data();

    // clearly, this could cause problems,
    // depending on what data is at 0x1234.
    do_something_safe(data);
}
```

In the five years since its creation, Rust has seen increased adoption, including components of Firefox [3], some virtualization projects [35], and several research operating systems [26, 27].

2.3 Operating Systems

Modern computers have hundreds of different processes running on them at the same time. A modern operating system must make sure that there is memory available for all these processes; it must provide them with resources and the ability to access hardware features; it must provide process isolation so they cannot access or overwrite one another's data. The kernel is responsible for these guarantees so that each process can perform its tasks successfully but without the ability to access other processes' data or to cause problems for the system or other processes.

Isolating processes' memory from one another is important for controlling this access and making sure that each process (or, more granularly, each bit of code) only has access to the appropriate areas of memory. The traditional solution to the process-isolation problem on a process-granularity level is *Virtual Memory*. Virtual memory as a concept dates back to MULTICS in the 1970s [4] - it was originally invented to solve other problems, but now one of its primary purposes is to segment separate processes' memory accesses by giving them different address spaces - separate numberings of memory so that each process does not have any concept of there being other memory on the machine. This is a partial solution to the process-isolation problem - if different processes have different address spaces, they cannot overwrite one another's data; however, the kernel has access to all data. If a malicious process can get the kernel to act on its behalf, it can still gain access to other processes' data.

2.4 Tock

Tock [26] is an operating system kernel written in Rust, taking advantage of Rust's memory safety features to ensure memory safety in the kernel [24]. The Tock developers chose Rust because it is memory-safe without high runtime overhead, in contrast to many other safe languages (Java, C#, arguably Go), which is a desirable property for an operating system. Figure 5 shows Tock's architecture at a high level: tock uses a microkernel architecture, so the core kernel (henceforth, just the *kernel*) implements just the minimum possible; memory management, scheduling, and allowing the processes to communicate with the system. In addition to the kernel, Tock has *capsules*, which are similar to drivers but are less priveleged than the kernel. They still run in kernel mode, with access to all of memory, but they are only allowed to run safe rust (no *unsafe* code blocks). Thus, Tock uses the memory safety properties of the Rust compiler to

guarantee that the capsules can only access the memory regions that the kernel lets them access.

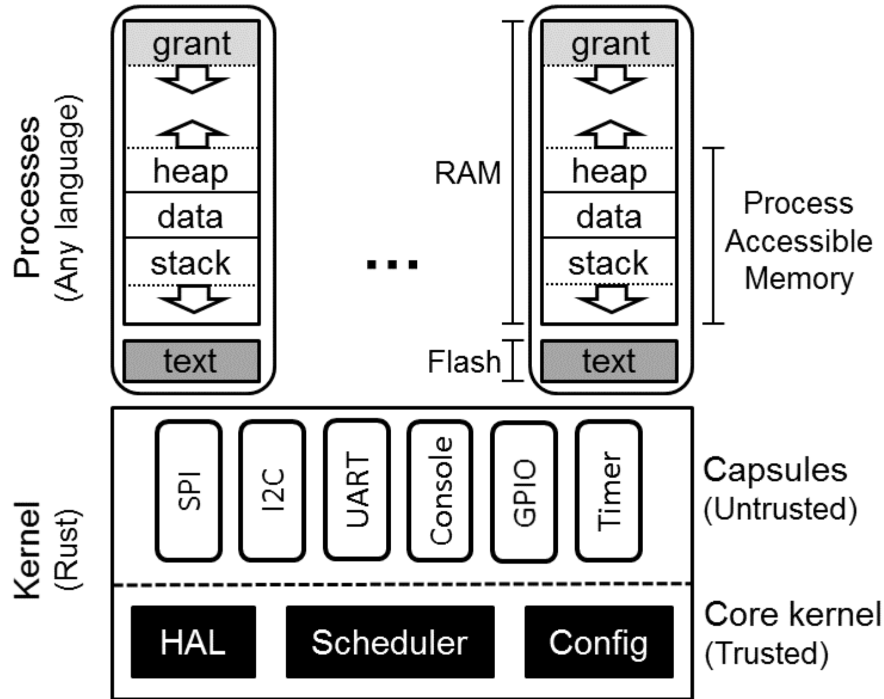

Figure 5: Tock Architecture, taken from [23]

Tock focuses on the security of the kernel. However, the kernel does not exist in isolation - it needs to be considered as part of the entire system, from userland code down to the hardware. Because the security is considered in isolation, there are vulnerabilities at the boundaries. Like every operating system, Tock needs to provide a way for the applications to communicate with the kernel, in order to interact with hardware and the outside world. This is the *system-call interface*. The way the system-call interface works in Tock (shown in Figure 6 and explained below) is simliar to how it works in most other operating systems (e.g., Linux, see Figure 3):

1. The process places instructions (what kind of system call it wants to perform) in registers.

2. The process places parameters - data or pointers to data - in registers.

3. The process makes a *superviser call*, triggering a context switch to the kernel, which reads these registers.

4. The kernel interprets the registers, packages any data or raw pointers in a Rust structure with base, bound, and Rust type.

5. The kernel sends the information on to whichever capsule the process wanted to contact.

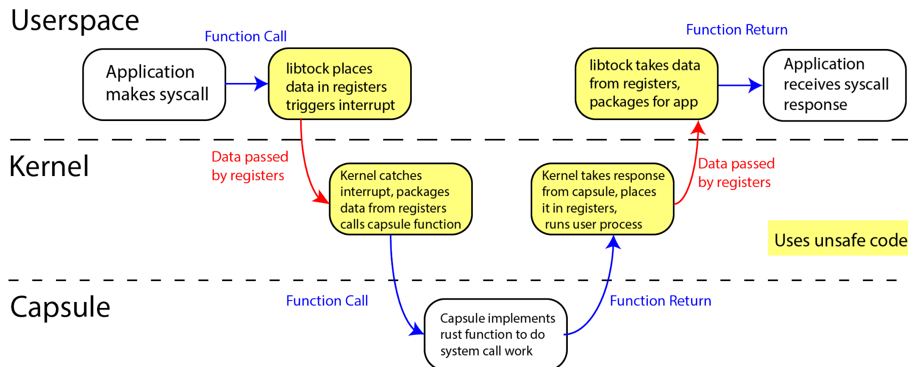

Figure 6: System Calls in Tock

Notice that in step 4, the kernel takes untyped data without compiler-guaranteed bounds, and packages it up in an `AppSlice` structure that can be used in safe rust. The bounds it uses are not provided by the compiler and are not guaranteed to match those on the data that was in userland; herein lies the problem.

3 Related Work

Many researchers have been developing secure operating systems for years. We will first discuss some sytems that directly inspire this work, then we will make some generalizations about what approaches have been taken to the system call boundary.

3.1 SAFE

SAFE is a system based on a *tagged architecture* and metadata checks around that architecture. In a tagged architecture, all data has *tags* stored in hardware; tags contain metadata about the memory they are attached to, such as type information, pointer metadata, "secrecy" data, or whether the data is executable. SAFE was designed as a clean-slate solution to the secure OS problem. SAFE also includes a formally verified kernel and new programming languages designed for safety [11].

SAFE consists of three layers, which the authors call the hardware layer, the "concreteware" layer, and the "userware" layer. We will discuss relevant properties of each layer.

First, the userware layer - "userware" is what SAFE calls the part of the system consisting of OS services that do not need to be priveleged and user programs. The SAFE developers put the drivers, network stack, and some other traditionally "kernel" code in an unpriveleged "userware" execution mode, in order to minimize the priveleged threat surfaces. Breeze, the language in which userware is written, is a type-safe language with *information-flow control* built in; that is to say, data is tagged with a level of *secrecy*, and secret data may only be accessed from a proper *authority context*. The goal is to isolate the userware and not let it do anything that will impact other userware or concreteware, primarily through tags and hardware that verifies tags before

allowing most operations [6].

SAFE's concreteware is made up of its memory manager, scheduler, and a few other small components. The concreteware is written in a subset of Breeze called Tempest, and it is formally verified - SAFE takes formal verification as an integral part of how they satisfy their security model.

Finally, the hardware includes several new or unusual components: for one, each word of memory is tagged with a pointer to arbitrary data that provides the "semantics" of the tag - this allows for a wide range of uses for tags, from the information-flow control mentioned above to Write XOR Execute. It also includes a *Fat Pointer Unit*, a piece of hardware specifically responsible for preventing spatial violations on pointers by incorporating into each pointer a *base and bound* (the region that pointer is allowed to access) while allowing them to fit in 64-bit words to minimize the impact on the memory footprint [12].

The SAFE project offers substantial security features for "userware" and "concreteware". Additionally, guarantees at the boundary are provided by run-time checks on the metadata encoded in the tags, using hardware mechanisms for efficient enforcement. Unlike the SAFE project, we avoid having to use a garbage-collector or other large runtime that manages user memory. We use a pre-existing programming language and are compatible with legacy hardware.

3.2 CHERI

CHERI (Capability Hardware Enhanced RISC Instructions) is a system designed to use capability-based addressing as an augmentation to traditional memory-protection systems [41]. It is a RISC[2]-based system. CHERI uses capabilities supported in hardware to improve the granularity of memory protection.

[2]Reduces Instruction Set Computer, an open instruction set standard

CHERI uses an augmented pointer the authors call a *memory capability* to manage permissions at a granular level. A memory capability contains a pointer, a length, and permissions on that pointer. CHERI also supports *object capabilities* - an object capability consists of permissions data packaged in a similar way to a memory capability. Any data tagged as a capability in hardware cannot be easily fabricated or moved around, requiring specific hardware instructions, some of which are priveleged.

CHERI places these protections on top of a traditional memory protection system, the *Memory Management Unit* (MMU). The MMU maps each process to a separate memory space, simplifying addressing within each process and preventing processes from being able to access data internal to other processes. The combination of the page-level protections of the MMU with more granular protections provided by capabilities allows CHERI to have very comprehensive control over which code can access what memory. Notably, however, the kernel has no MMU protection and it also has full capability access to all memory.

CHERI uses object capabilities to protect its system-call interface on the userspace side - applications must have a specific object capability to make a system call [37], which limits the ability of malicious code to make an unsanctioned system call. Data passed into a system call takes the form of a memory capability, so memory accesses across the system-call boundary are equally protected as memory accesses within an application.

CHERI is a general system that mixes novel hardware support and existing protections, and their system-call interface is well-protected. However, CHERI uses custom hardware to a larger extent than this project. Their processor has substantially new components in order to be able to manage capabilities and require their use in place of most pointers. Though we hope to eventually have hardware support for some of our protections, we aim to support an existing

instruction set (e.g., ARM or RISC-V).

3.3 Singularity

Singularity is a memory-safe operating system with software-based process isolation [17]. Singularity forbids shared memory between processes. Singularity is a microkernel where the drivers and most other kernel components are implemented as userspace processes.

Like Tock, Singularity does not use an MMU to isolate processes, but does so through software mechanisms and by limiting pointer arithmetic. Singularity keeps what the authors call a *memory independence invariant*: compile-time and runtime checks make sure no process can access regions of memory belonging to any other process. These checks use a mechanism simliar to those used by Rust.

The most relevant aspect of Singularity is the mechanism by which processes (including drivers, which are just special processes) communicate. They are set up with *channels* through which data (of a specific type) can be sent. Channels have types on both ends. In order to enforce that processes do not gain pointers into areas they should not access, Singularity implements channels by placing data into a special region called the *exchange heap*. Channels are a safe way to transfer data between processes, though they can cause deadlocks [33].

A few other aspects of Singularity are relevant to this work. The garbage collector, which is one of the features allowing memory safety in their language Sing#, includes approximately half the unsafe code in the Singularity kernel. All processes using hardware (i.e., drivers) must declare which hardware they are using in a manifest to prevent conflicts. The compiler compiles code to "Typed Assembly Language", which can be verified to be safe, in case the complicated Sing# compiler has bugs.

Singularity is relevant to this work because it has a working mechanism for process-kernel communication that checks types at the boundary. However, Singularity relies on a proprietary language and a large (garbage-collecting) runtime to solve these problems, both of which we avoid by using Rust.

3.4 System-call Approaches

Several different approaches to the system-call interface have been taken by recent secure OS work. Some systems [6, 28, 40] do not address the possibility of a vulnerability in the system call interface at all and just trust (applications and) the kernel not to misuse the pointers they get.

Some systems [17, 38] use a large runtime or the fact that code is running in a VM in order to check types on data being passed between processing units at runtime. Singularity [17] additionally places any data that needs to be read by multiple processes in a designated memory region, so that the kernel does not need to hold pointers to application regions.

In Reenix [27], the userspace applications and the kernel must be compiled together, and there is no less-priveleged mode for the userspace applications to run in, so safety is guaranteed at the system-call boundary through Rust's compiler. This is an excellent use of compiler guarantees but it is too restrictive for our work.

Many systems [22, 26, 32] allow the programmer to specify a size of a buffer to pass across an interprocess or system-call boundary, but do not check that the length specified corresponds to the data in the buffer.

4 Threat Model

Our threat model (See Figure 7) trusts hardware entirely, the intent of the kernel author, and the fact that user applicaiton code has been compiled by a bona

fide compiler.

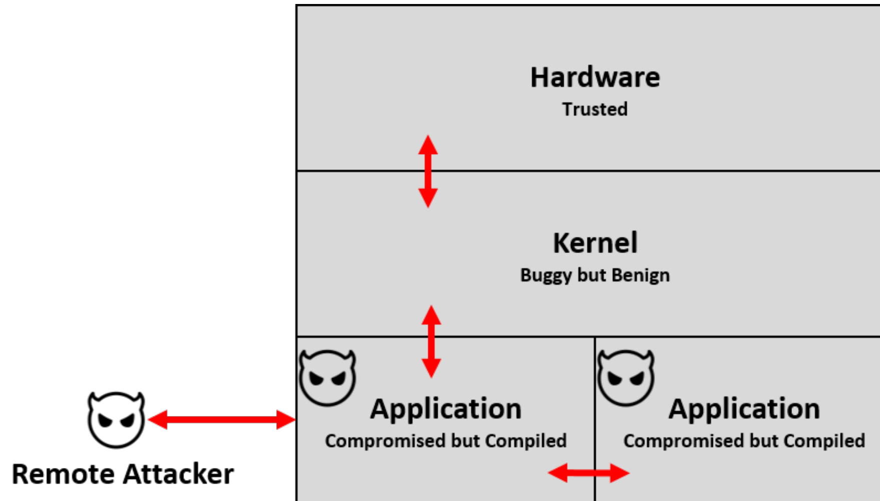

Figure 7: Our Threat Model

First, we trust the hardware on which the software is running. This is a decision we made because we need to trust something in order to maintain a basic amount of usability, and hardware has longer development cycles. Although this is imperfect [20], we must concentrate on part of the threat space.

Running on the trusted hardware is a "benign but buggy" kernel, which can be trusted to attempt to implement its specification, but it may have bugs. We do not consider supply-chain attacks on the kernel, where an adversary contributes malicious code to the kernel code; we trust that the kernel implementers act in good faith.

Finally, we have the application running under the operating system - we consider it to be "compromised but compiled" - that is to say, it must be compiled with our toolchain, but otherwise is free to do its best to interfere with other applications. However, a user application is only allowed to use unsafe code as far as calling libraries that use unsafe code; it is not allowed to implement its own unsafe sections. We aim to prevent this application from crashing

or compromising the system or other applications. The user application may collaborate with a remote attacker to attempt to break the security of the system.

5 Example Vulnerability

Tock supports userland code in either C or Rust; we will assume a Rust-based userland; with C userland code, it is less meaningful to discuss the types of the data going to the kernel, because C has a less strict type system than Rust.

Because both userland and kernel code are written in Rust, type safety and memory safety should be guaranteed at all times; however, this is not actually the case if data is passed between them unchecked, as happens with system calls. As discussed in Section 1, data is passed between the kernel and userland by putting it in certain registers or by putting (untyped) pointers to it in registers; this means that it loses its type, owner, and any other rust abstractions when passed across the boundary. In Listing 4, we present an attack taking advantage of that vulnerability, motivating this work.

Listing 4: Buffer overflow in Rust under Tock

```
fn main() {
    let mut console = Console::new();

    // if _buffer overflows, _mod will be overwritten
    let mut _buffer: [u8; 6] = [2; 6];
    let mut _mod: [u8; 6] = [1; 6];

    // initially _mod contains all ones
    console.write(String::from("Contents of _mod before:\n\n"));
    for x in _mod.iter() {
        console.write(fmt::u32_as_hex((*x).into()));
        console.write(String::from(" "));
    }

```

```
15      // share buffer to kernel
16      console.write(String::from("\n\nSharing␣_buffer␣to␣kernel\n\n"));
17      const CAPSULE_RNG: usize = 0x40001;
18      let buf_len = 16;
19      let _allow_res = syscalls::allow(CAPSULE_RNG,0,buf,buf_len);
20
21      // allocate an indicator for the kernel to tell us when it's done
22      let is_written = Cell::new(false);
23      let mut is_written_alarm = |_, _, _| is_written.set(true);
24      let _sub_res = rng_set_callback(&mut is_written_alarm);
25
26
27      console.write(String::from("\n\nFilling␣_buffer␣randomly\n\n"));
28      let num_bytes = 16;
29      let _cmd_res = syscalls::command(CAPSULE_RNG, 1, num_bytes, 0);
30
31      // wait for the kernel to finish
32      syscalls::yieldk_for(|| is_written.get());
33
34      // after RNG is truggered on _buffer,
35      // _mod is populated with random numbers
36      console.write(String::from("Contents␣of␣_mod␣after:\n\n"));
37      for x in _mod.iter() {
38          console.write(fmt::u32_as_hex((*x).into()));
39          console.write(String::from("␣"));
40      }
41
42      loop {
43          syscalls::yieldk();
44      }
45  }
```

We will describe what the code in Listing 4 is doing, step-by-step. First, we allocate the buffers in sequence (line 5), so that they are adjacent on the stack. The next 10 lines print out the initial values of the buffers and set up for the system call. Next, we call the `ALLOW` system call (line 19), sharing the buffer `_buffer` with the RNG capsule. Note that the real length of `_buffer` (line 5 and 6) is less than the provided length (line 18), but there are no mechanisms in

place to prevent the user program from supplying the incorrect length. [3]

Thus, instead of sharing just the 6 bytes of `_buffer`, we have also shared the memory containing `_mod`. The next 10 lines set up notification so that the user process can know when the kernel is done providing random data. On line 29, the user process requests that the kernel fill the memory region it has shared with random bytes. When we print out `_mod` on line 38, we find that it has been replaced with random data.

Note the similarity between this example and Listing 1, a stack smash in C. We are essentially using the system-call interface provided by Tock as a way to circumvent the Rust features that would prevent us from accidentally indexing too far into `_buffer`. This thesis presents a system that prevents this vulnerability from being exploited.

Many approaches have been taken to building a system with built-in memory safety, and they have their benefits and drawbacks; but an all-too-common drawback is not considering the boundary between safe components in secure system design.

For example, the way that the processor communicates with input and output devices on most modern systems is through memory, or through mapping I/O devices to locations in memory. Even kernels written in memory-safe languages need to cast arbitrary memory addresses to typed values in order to deal with memory-mapped I/O, and it is not always clear that they can know in advance, for example, the length of these regions (e.g., the VGA buffer for video output). By augmenting the safe kernel with hardware support for type constructs, one can hope to make even cases like this safe.

This is not just a problem for Tock - it is a pervasive pattern in large systems,

[3] Here it should be noted that we are using a custom userland syscall implementation; the version that the Tock developers provide includes a check on this, but it's just in the userland library that Tock programs make system calls through, not inherent to how the kernel processes the system call.

and kernels are no exception. For example, there are a number of linux kernel bugs that resulted from poor bounds checking [7, 9], as well as Windows vulnerabilities that have been exploited by real-world malware on improper checking of system call arguments [10, 8].

In this thesis, we build a system that has built-in resistance to attacks on the boundary between the userspace and kernel.

6 Methods

The system described in this thesis is an improvement around the system-call interface of Tock. Our goal in designing this system was to preserve type metadata available at compile-time that is not normally available when the data crosses the system-call boundary to the kernel, and to pass that data to the kernel.

In summary, data passed through system calls to the kernel has its type metadata passed as well through a separate mechanism, allowing protection against the sort of attack described in Section 5 without requiring modification to either the userland or the capsule code. A high-level overview of the system is in Figure 8 (compare with Figure 6).

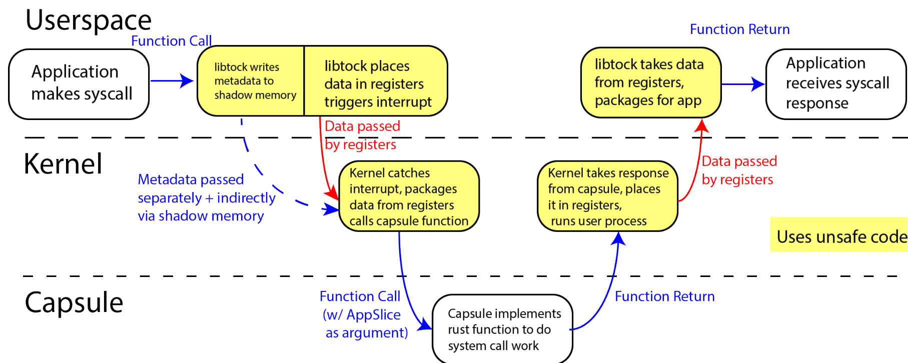

Figure 8: Syscall architecture in Modified Tock

The system is composed of three parts:

1. Whenever data from user-mode is passed to the kernel, its type and size are first written to a *shadow memory* region; i.e., a region invisible to the normal operation of the program.

2. When data is passed across the system-call boundary, the kernel component handling the sytem call retrieves the type data from the shadow memory and passes it to the capsule handling the system call, along with the data itself.

3. When a capsule tries to access data it has been passed from userspace, our system prevents access unless the data type matches the type expected by the system call, by truncating the data passed on or by returning an error to the userspace program requesting the operation.

6.1 Shadow Memory and Type Tracking

In order to expose the types of the user-mode data to the kernel, we track them in *shadow memory*. Shadow memory is memory that is not included in the memory the process is allowed to use for its heap, and it is used for tracking metadata. The shadow memory is only accessible to the process by using unsafe code. When allocating memory for processes, we split the memory for each process in half, allocating half of the space to shadow memory and using the other half for the regular process memory. As depicted in Figure 9, the top half of the memory for each process is the shadow memory and the bottom half has the memory layout that Tock normally uses. Metadata for each byte of memory is stored in the corresponding byte of shadow memory.

We need not track the full Rust type of every object in the shadow memory; there are three important pieces of data to store for each byte:

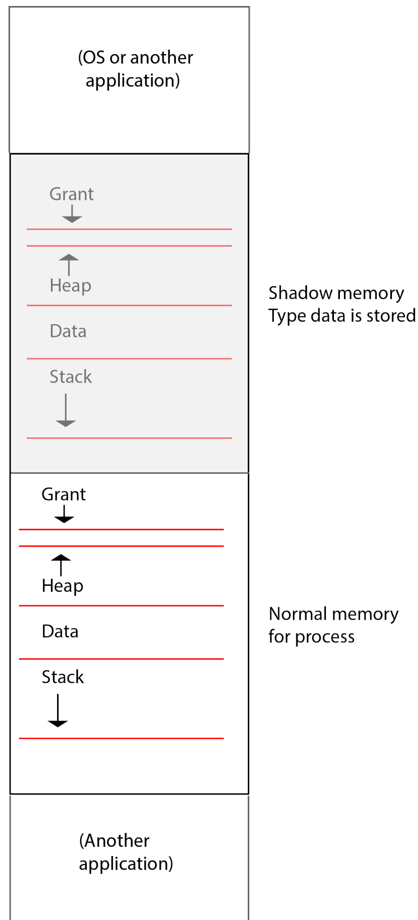

Figure 9: Memory layout modification

- *Type* - we really only need to store whether it is part of an array/slice of raw bytes, since all capsules expect shared user data to be in the form of raw bytes. Doing more complex checks would require significant modification to the capsule code, for little benefit, since most of the data required by Tock's system calls is not structured further than the byte array in which it is passed. As shown in Figure 10, the shadow memory regions corresponding to where `_buffer` and `_mod` are include the BYTE marker, but the region after them does not.

- *Mutable* - Whether the reference being shared is mutable - if it is not mutable, then the kernel should not be able to modify it. This data is not currently used by the kernel (see Section 8.4 for further discussion). As shown in Figure 10, the shadow memory regions corresponding to where both `_buffer` and `_mod` are, as well as the following region, are marked MUTABLE.

- *Start* - Whether this particular byte is the beginning of its particular array/slice being passed to the kernel. When our system-call handler is checking a piece of memory shared by a user program, if it reaches a byte that is the start of its slice when examining a piece of memory passed from userspace, the kernel does not share any bytes beyond that point to the capsule processing the system call. As shown in Figure 10, the shadow memory locations corresponding to the first byte of each of `_buffer` and `_mod` have the START marker.

We store this information in the lowest 3 bits of each byte of the shadow memory; the upper 5 bits are unused.

6.2 System-Call Handler Modifications

The system call handler in the Tock kernel takes the data passed from userspace and packages it in a slice-like structure called an `AppSlice`. This structure has an API similar to a Rust slice (variable-length array), but is slighlty more complicated internally.

We have modified the kernel system call handler in Tock to include a pointer to the type data in shadow memory in the packaged `AppSlice` structure that it passes to the capsules' system-call handlers. This modification allows the `AppSlice` to handle the type-checking, so that the capsule code does not need to be modified at all.

If the user process tries to share too many bytes to the kernel to output through the console, the `AppSlice` will simply ignore any bytes beyond the limit and show only the smaller set of bytes to the console capsule. If the chunk of memory the user tries to share is of the wrong type entirely, the kernel will return an error to the user process that tried to share the data to the kernel.

More concretely, if the user program passes a buffer of length 6 (e.g., `_buffer` in Figure 10), with claimed length 16 (as described in Section 5), the `AppSlice` will only expose the 6 bytes before `_mod` starts. If the user program shares a pointer to a not-byte-array with any length (e.g., `something_else` in Figure 10), the ALLOW call will return an error to the user program that tried to share the memory of the wrong type.

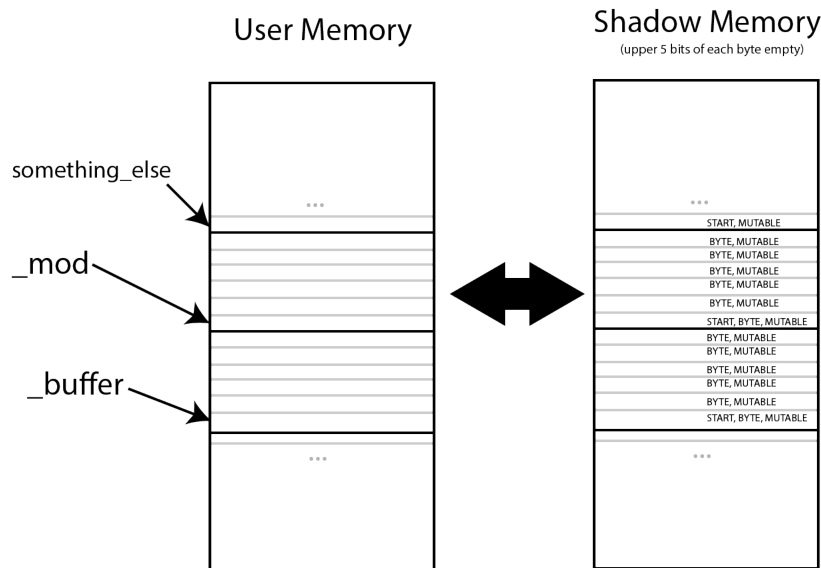

Figure 10: Example of Shadow Memory Contents

6.3 User-Mode Component

The user-mode component of the system is simpler than the kernel-mode component. Whenever a buffer is passed to our (Rust) system-call library, that buffer's length is checked and then the type is written to each byte in the corresponding shadow memory location. When the system call returns, the shadow memory region for that object is cleared so that future objects allocated in the same memory region are not pre-marked with the wrong metadata.

The shadow memory location is determined by calling a system call that returns the offset to the shadow memory. In the system as implemented, this extra system-call provides a substantial amount of overhead. That overhead is discussed further in Section 8.5.

The user-mode application (which is written in Rust) is not allowed to use unsafe code except by calling our libraries (recall our assumptions about Rust userland code in Section 4). As a result of this restriction, we can be confident that the metadata attached to the Rust object when it is passed to our library is correct.

Several conditions need to be met for the types of the userland data to be available to the kernel at runtime. For one, this data needs to exist past the time of compilation; normally the data is used for compiler analysis and not packaged into the binary in any way. For two, it needs to be placed into the shadow memory. If we did not change the compilation process by having our library inspect the types of the data it is passed, this data would not be available at runtime and no system would be able to distinguish between different data types in memory. Our solution is that our library forces the compiler to include the types in the binary, because they will be written to memory by our library which was present at compile-time. A different solution, discussed in Section 8, is a compiler extension that places type information somewhere else in the binary

for reference.

7 Evaluation

We evaluate this work in several dimensions.

- We consider performance of both specific system-call operations and general workloads, comparing to vanilla Tock. We additionally test its performance under synthetic loads. We look at both how much extra memory we use and how much slower this modification makes Tock.
- We show that this protection is able to defend against the vulnerabilities we set out to fix.
- We reason about what other vulnerabilities this system can prevent.

7.1 Performance

In order to evaluate the performance of the system, we timed tasks on both vanilla Tock and our modified system. We used the CPU clock to determine the number of CPU ticks[4] tasks took to complete. In order to make the CPU clock accessible to our userland tests, we relaxed some of the MPU protections in Tock while performing our tests.

We ran two types of benchmarks: microbenchmarks and macrobenchmarks. The microbenchmarks (described in Section 7.1.1) did very little work in addition to the system-call task we were measuring or were subsets of that system call. The goal of these tests was to compute the overhead on the system-calls themselves, and thus place a theoretical upper-bound as to how much

[4]We had difficulty accessing the actual CPU clock; the way the board we used for testing (NRF52dk) is set up in Tock seems to only allow access to a 32 MHz clock also on the CPU, as opposed to the 64 MHz CPU clock. All the measurements in this paper refer to the 32 MHz clock.

our changes could effect the system performance. The macrobenchmarks (Section 7.1.2) attempt to gauge the impact of these modifications on more realistic systems, by including not just system-calls but waiting for the full execution of the system call, sending data over I/O, and so on.

7.1.1 Microbenchmarks

We will describe the microbenchmarks that are the components of the system call, then we will describe the microbenchmarks that consist of an ALLOW system call and a little more computation.

Table 1 and Figure 11 show the different processes that take time in an ALLOW system call, which is where our overhead is: the system call starts with `libtock` interpreting the parameters given by the user program (A-B). Then it gets the offset to the shadow memory using a MEMOP syscall (B-C). Next, it writes the metadata to the shadow memory (C-D). The kernel checks the shadow memory and packages the pointer into an `AppSlice` (D-E), and then the capsule stores the `AppSlice` for later (E-F). Finally, `libtock` clears the shadow memory region (F-G).

Context switching overhead (the time from the first B to the second B, for example) was too small to reliably measure, however it is included in the overall A-G measurements and in the measurements that include a system-call boundary traversal (B-C and D-F).

We measured the timing of each of these regions under both vanilla Tock and our modified Tock with a 100 byte buffer; the results of these measurements are in Table 2. Results are averaged over 100 runs of the system call.

We ran two other microbenchmarks in addition to those described in Table 1: SimpleRNG, which measures the time to start a random number generator operation, and SimpleBLE, which measures the time to start a simple bluetooth advertisement. These operations were chosen because they are representative

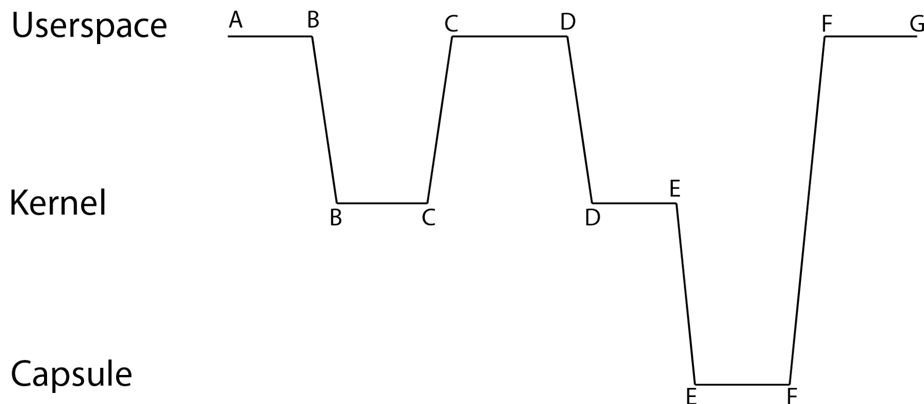

Figure 11: Parts of a ALLOW system call under our system

Chunk	Where	Description
A-B	libtock	Get length from buffer
B-C	kernel	**Get offset to shadow memory**
C-D	libtock	**Write data to shadow memory**
D-E	kernel	**Validate data in shadow memory**
E-F	capsule	Store `AppSlice` for later use
F-G	libtock	**Unmark shadow memory**
A-G	overall	Total

Table 1: Parts of an ALLOW Syscall under our system. Bolded steps take more substantial time.

of the system calls available in Tock. Average timing of these benchmarks is shown in Table 3. The reason that the AllowSyscall benchmark shown here and in the CDF graph below takes longer than the ALLOW syscall in vanilla Tock as timed in Table 2 is that we had to turn off some compiler optimizations to prevent the compiler from optimizing away our measurement ability and thus got slightly slower code.

In order to check that our system call modifications take a consistent amount of time and not a long tail, we timed an ALLOW systemcall on 100 bytes 10000 times. Figure 12 shows the probability of completion over time based on 10000 iterations of the ALLOW syscall on both systems. Our system provides a noticeable but consistent amount of overhead.

Chunk	Vanilla	Modified
A-B	0	0
B-C	0	2.51
C-D	0	0.18
D-E	(not measured)	(not measured)
E-F	(not measured)	(not measured)
D-F	2.91	6.20
F-G	0	0.17
A-G	2.91	9.06

Table 2: Timing for parts of an ALLOW syscall, all numbers in ticks

Test	Vanilla Tock	Modified Tock	Description
AllowSyscall	3.16 ticks	8.92 ticks	Pass a buffer to the kernel with 24 bytes of data
SimpleRNG	20.7 ticks	21.7 ticks	Pass a buffer to the kernel, ask the kernel to generate 24 bytes of random data in the buffer. Do not wait for the data to be generated.
SimpleBLE	17.0 ticks	21.0 ticks	Pass a 17-byte buffer to the kernel, ask the kernel to advertise that data over Bluetooth. Do not wait for the data to be advertised.

Table 3: Timing on Microbenchmarks of modified Tock syscall interface

We applied the same technique to the SimpleRNG test, and found that although our system does not have high variance in its timing, the system random number generator does, and so the test occasionally takes a lot longer. Running this test 400 times on both systems produces the CDF shown in Figure 13 for the SimpleRNG test. The SimpleBLE test has overhead simliar to AllowSyscall so we do not produce a CDF for it here.

7.1.2 Macrobenchmarks

We measured 3 larger operations as well, to get an idea of what impact our changes have on the overall speed of the system.

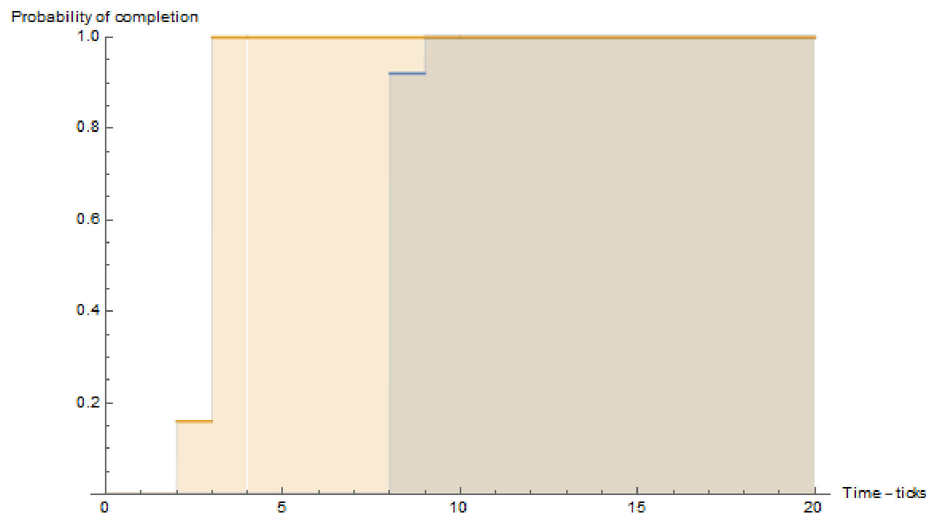

Figure 12: CDF of times for an ALLOW syscall on 100 bytes of data. Yellow is vanilla Tock, blue is our sytem.

First, as a baseline, we ran a pseudo-random number generator in userspace. It does not make any system calls, so we did not expect any overhead. Timing varied slightly based on system load, but the times fluctuated in similar ranges under modified and vanilla Tock. The average time on vanilla Tock was one tick faster, but we do not believe this was due to anything other than random chance.

Next, as a task fairly representative of the things the Tock systems do, we timed an program that scans for BLE devices using the BLE driver. It sends the data back from the kernel over a buffer created with an ALLOW system call and makes frequent use of the system-call interface. We found that it has approximately 3% overhead when run under modified Tock.

Finally, as an attempt to find a task that was heavy in ALLOW system calls, we timed a program that prints 100 lines of text to the console. This program had approximately 6% overhead when run under modified Tock. Because very little computation was performed on the data in this test, its performance is

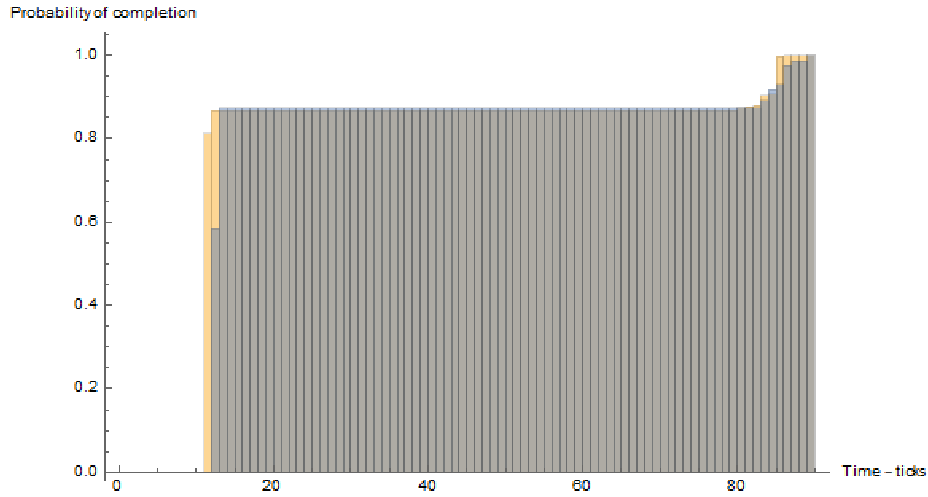

Figure 13: CDF of times for setting up an RNG transaction on 24 bytes of data. Yellow is vanilla Tock, blue is our sytem.

close to the worst-case performance in a realistic scenario for our system.

These results are compiled in Table 4.

Test	Vanilla Tock	Modified Tock	Description
Cryptography	609 ticks	610 ticks	Compute a PRNG in userspace. No system calls involved so should not expect overhead.
BLE_scan	6871 ticks	7061 ticks	Scan for BLE advertisements, return the information received to the user process.
PrintData	19757 ticks	20613 ticks	Write 100 lines of text to the console.

Table 4: Timing on Macrobenchmarks of modified Tock syscall interface

7.1.3 Synthetic Benchmark

In order to determine how our system's performance depends on the size of the ALLOWed buffer, we tested how long an ALLOW call takes on each size of buffer from 1 byte to 500 bytes. Figure 14 shows the timing of an ALLOW

system call vs. the length of the buffer being shared. As expected, the call takes linearly longer depending on the size of the buffer for our system, but is constant for vanilla Tock. This difference is because our system checks the whole length of the memory passed to verify that it is of the correct type.

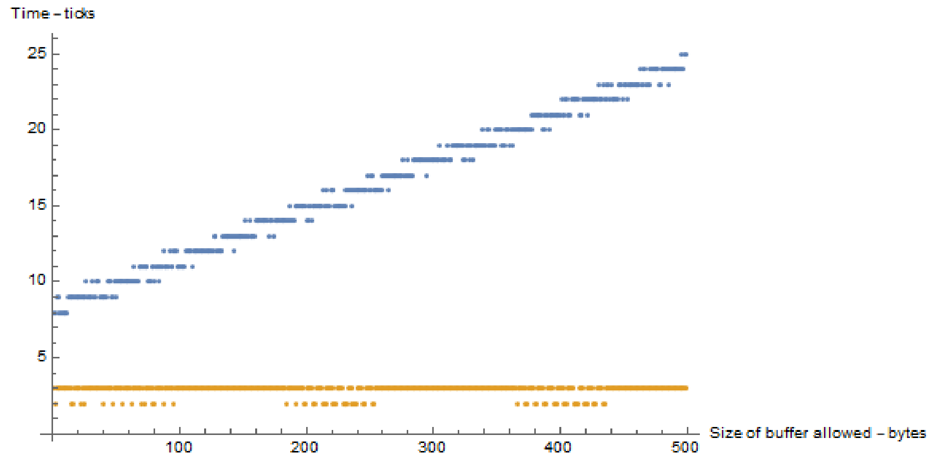

Figure 14: Time for an ALLOW operation vs size of buffer. Blue is modified tock, Yellow is vanilla.

7.1.4 Memory use

The modified system uses substantially more memory for each process; in particular, it doubles each process's memory usage. We chose to use a direct mapping of shadow memory to process memory to facilitate development while still evaluating the security provided by such a system. This choice is further discussed in Section 8.

7.2 Security Evaluation

In order to evaluate the security of our system, we ran the attack described in Section 5 on our modified system. As described in Subsection 6.2, the kernel receives a system call with a pointer and a length that do not match; in vanilla

Tock, the kernel generates random numbers over the entire length of the passed buffer, as shown in the output in Listing 5.

Listing 5: Cleaned up output from the example exploit under original Tock

```
1 Contents of _mod before: 0x01 0x01 0x01 0x01 0x01 0x01
2 Contents of _buffer before: 0x02 0x02 0x02 0x02 0x02 0x02
3 Generating a random number in _buffer...
4 Contents of _mod after: 0x3a 0x54 0x6c 0x96 0x92 0xeb
5 Contents of _buffer after: 0x7d 0xb0 0xfe 0xa8 0xb1 0x0a
```

Our system call-handler truncates the slice passed to the kernel so that only the variable passed to the kernel can be modified, and other variables that happen to share the same memory space (i.e., `_mod`, here) cannot be modified (i.e., there cannot be a spatial violation). Listing 6 shows the output from running the attack under modified Tock. As you can see, the data in `_mod` is not overwritten, even though the system call requests that data be written beyond the length of `_buffer`.

Listing 6: Cleaned up output from the example exploit under modified Tock

```
1 Contents of _mod before: 0x01 0x01 0x01 0x01 0x01 0x01
2 Contents of _buffer before: 0x02 0x02 0x02 0x02 0x02 0x02
3 Generating a random number in _buffer...
4 Contents of _mod after: 0x01 0x01 0x01 0x01 0x01 0x01
5 Contents of _buffer after: 0x93 0xe4 0xa0 0xbb 0x17 0x8c
```

7.3 Broader Effects

The technique demonstrated in this thesis is not applicable solely to Tock, and its impact if applied more widely could be substantial. Although the performance hit caused by the particulars of this technique and the need to write all the code in Rust make this impractical for large-scale application to existing systems, the benefits could be real. Any sort of memory corruption caused by metadata lost between the application and the operating system could be prevented, including [7, 10, 8, 9].

In order to apply this technique to another operating system, we would need several criteria to be satisfied. First, we need to annotate the type of the data going into each of the system calls for the OS; next, we need the user processes to be written in a typed language. If these criteria are satisfied, we would need the compiler that compiles the user processes to instrument the code with instructions to write the types to shadow memory. If implemented in Windows, for example, this technique would prevent any vulnerabilities based on the system misinterpreting data sent to it by a malicious user program, such as [8], where the user process can convince the kernel to write to any address by telling it that that address is a certain data structure, because the address would not be in the user process's memory area.

In separately compiled cooperating modules, the benefits of compilation can be cancelled out because compiler aassumptions, such as knowing everything that accesses a given memory location, or knowing that an address is no longer in use, fail. A fundamental issue in compiler-guaranteed correctness is that it requires a global view, and a global view is not always available. Ignoring this fact creates opportunities for bugs and vulnerabilities.

The bugs and vulnerabilities demonstrated in this thesis come about despite Rust's guarantees because of this lack of a global view in the compiler. However, without excessive overhead we have closed such holes by specifically passing metadata at runtime; this principle should have broad applicability.

8 Discussion

There are a few other limtations of this work that do not fit into any of the axes on which we evaluated in Section 7. We will raise several issues, discussing for each the reasons for it and some possible ways to address it.

8.1 Performance Overhead

As discussed in Subsection 7.1, our technique has 100% memory overhead, which is significant, especially on the embedded systems Tock is designed to run on. This memory overhead can likely be reduced because of the sparsity and repetitiveness of the data being stored in the shadow memory. For example, we could compress the representation of the data in shadow memory by using one byte for each word of memory instead of per byte. We also need not have shadow memory allocated for each process's grant region, which is only used by capsules and not by the process itself.

As far as speed goes, we would like to implement some of this work in hardware becaue it will alleviate much of the time overhead. In order to do that, we would like to use custom hardware. In order to modify the hardware, we would need to have open source hardware to modify, which likely means running Tock under RISC V . The work to port Tock to RISC V is underway, but it is not yet ready, so we implemented this work under ARM and in software.

8.2 Compatibility

Our kernel is not compatible with the vanilla `libtock` provided by the Tock developers, because it expects data to be written to shadow memory by `libtock` in userland.

We chose to implement the userland functionality in `libtock` because it was simpler and did not require compiler modifications. But for this sytem to be viable to use, we would like to move the responsibility for writing type information to shadow memory from `libtock` to the compiler. This would involve the compiler emitting code to mark the types of variables when they are allocated and erasing the types when they are deallocated. This modification will make our kernel compatible with vanilla `libtock`. Another benefit of this work

would be the ability to provide temporal safety in addition to spatial safety, as described in the next subsection.

8.3 Temporal Safety

This technique does not provide temporal safety in Tock due to the unusual way Tock does system calls. Tock only has 5 system calls, which are used in a very specific protocol, involving four calls: an ALLOW to share the buffer on which to operate, a SUBSCRIBE to receive notification when the process has finished, and a COMMAND to request that the kernel perform the operation, and a YIELD to wait until the kernel has finished. All system services are provided using this protocol in `libtock`, but one could imagine a user attempting to be asynchronous to save time and accidentally deallocating memory before the kernel is done with it.

A Rust program can still expose a buffer to the kernel, triggering our memory checks, and then deallocate that buffer and allocate something else in its place after our checks but before making the system call to use the shared memory. This is not currently protected because memory is unmarked immediately upon making the ALLOW system call to share the memory with the kernel, and thus we only check the shadow memory upon ALLOW calls. Listing 7 demonstrates this attack.

Listing 7: Violation of Temporal Safety under our system

```
1 const DRIVER_NO = BLUETOOTH_DRIVER_NO;
2 const ALLOW_NO = BLUETOOTH_SEND_BUFFER;
3 const SUBSCRIBE_NO = BLUETOOTH_ON_SEND_FINSH;
4 const COMMAND_NO = BLUETOOTH_SEND_ASYNC;
5 fn do_one_thing() {
6     // allocate a buffer to send, fill it with zeroes
7     let mut data_to_send: [u8;100] = [0;100];
8     // share the buffer with the kernel
9     syscalls::allow(DRIVER_NO, ALLOW_NO, &mut data_to_send, 100);
```

```
10 } // data_to_send is deallocated here, but when it was shared,
11   // it pointed to a valid location in memory.
12
13 fn do_another_thing() {
14     // allocate a new buffer, fill it with ones
15     let mut data_in_the_same_place: [u8;100] = [1;100];
16
17     // tell the kernel to tell us when it's done
18     let done = Cell::new(false);
19     let mut done_alarm = |_, _, _| done.set(true);
20     syscalls::subscribe(DRIVER_NO, SUBSCRIBE_NO, &mut done_alarm);
21
22     // tell the kernel to do its thing
23     let bytes_to_use: usize = 100;
24     syscalls::command(DRIVER_NO, COMMAND_NO, bytes_to_use);
25
26     // wait for the kernel to finish
27     syscalls::yieldk_on(|| done.get());
28 }
29
30 fn send_some_data() {
31     do_one_thing();
32     do_another_thing();
33 }
```

In Listing 7, we demonstrate a bug that a programmer could easily fall victim to, leading to unexpected behavior. A sequence of system calls in Tock is split between multiple functions, and the data sent to the kernel (`data_to_send`) goes out of scope on line 10 before it is used. Because they are both allocated on the stack, `data_in_the_same_place` is allocated in the same place as `data_to_send` was, and it is now pointed to by the pointer we passed to the kernel. Thus, when the kernel actually uses the data on line 24, it uses the data from the wrong buffer. This problem exists in regular Tock, but it becomes more notable in our modified Tock because we have prevented the spatial version of this problem.

If the compiler instrumented userland code to write the type information to shadow memory at allocation and deallocation time, `AppSlice`s could be more

thorough when they check the type information, and could do so at every use, throwing an error whenever the types mismatched, instead of only doing so at creation time. This modification would also require some modifications to capsule code, because a pointer may be valid when a capsule receives it, but no longer valid when the capsule tries to use it, confusing the capsule and possibly causing a crash and a kernel panic if not implemented properly.

This change would incur additional overhead of approximately an additional 2.3 tick per 100 bytes on each access to an `AppSlice`, based on the data from the synthetic benchmark and the microbenchmarks. We think this would be a fair price to pay for the protection from this category of bugs; it would consititute an 86% increase in overhead when compared to just the context switching to the kernel, but would be fairly minor ($< 10\%$) when compared to the work actually done by the capsule accessing the data, based on our macrobenchmarks.

8.4 Mutability

In section 6, we discussed three pieces of information stored in shadow memory: type, mutable and start. We do not currently use the mutability data because doing so would require modifications to capsule code. We cannot provide a different amount of data when the capsule tries to mutate the data from when the capsule simply tries to read it, due to implementation limtations on the `AppSlice` structure gating capsule access to user memory. We cannot return an error upon trying to access immutable user memory mutably because the capsules do not check for errors upon accessing an `AppSlice`. Thus, using the mutability data would require internal modifications to all the capsules. Because of that requirement, we are not certain that enforcing mutability across the system call boundary is worthwhile.

8.5 Extra System-Call

The way in which we have implemented the transfer of metadata to shadow memory currently involves an extra MEMOP system call to retrieve the offset to shadow memory for each ALLOW system call (B-C in Figure 11). This decision was made so as to allow the `libtock` modifications not to retain extra state, but it comes at the cost of additional overhead on every ALLOW system call. We can tell from Table 2 that this overhead is a significant portion of our system's overhead - 86% of the overhead on an empty buffer, or 50% on a buffer of 100 bytes, getting less significant for larger buffers.

This overhead could be reduced from once per ALLOW call to once per user program by storing the offset in a static variable in user memory inside of the `libtock` module responsible for writing data to shadow memory. This is one of the first improvements we would make to decrease overhead.

9 Conclusion

Many operating systems designers have not focused their attention on memory safety at the system-call boundary; vulnerabilities exist in Windows[8], Linux[7], and even memory-safe operating systems like Tock (Section 5) because of this oversight. This thesis showed that such vulnerabilities exist. We designed and implemented a system to protect against this issue in Tock. We successfully built a system that defends Tock against these vulnerabilities, and we evaluated its performance and security. We also discussed what it would take to apply that system to other operating systems and how and why it might be applied.

References

[1] Martın Abadi et al. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM. 2005, pp. 340–353.

[2] Periklis Akritidis et al. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." In: *USENIX Security Symposium*. 2009, pp. 51–66.

[3] Brian Anderson et al. "Engineering the servo web browser engine using Rust". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM. 2016, pp. 81–89.

[4] A. Bensoussan, C. T. Clingen, and R. C. Daley. "The Multics Virtual Memory: Concepts and Design". In: *Commun. ACM* 15.5 (May 1972), pp. 308–318. ISSN: 0001-0782. DOI: 10.1145/355602.361306. URL: http://doi.acm.org/10.1145/355602.361306.

[5] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 147–160.

[6] S. Chiricescu et al. "SAFE: A clean-slate architecture for secure systems". In: *2013 IEEE International Conference on Technologies for Homeland Security (HST)*. Nov. 2013, pp. 570–576. DOI: 10.1109/THS.2013.6699066.

[7] *CVE-2003-0985*. Available from MITRE, CVE-ID CVE-2003-0985. Sept. 2004. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0985 (visited on 08/20/2018).

[8] *CVE-2016-7255*. Available from MITRE, CVE-ID CVE-2016-7255. Nov. 2016. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7255 (visited on 09/11/2018).

[9] *CVE-2018-10940*. Available from MITRE, CVE-ID CVE-2018-10940. Apr. 2018. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10940 (visited on 08/20/2018).

[10] *CVE-2018-8440*. Available from MITRE, CVE-ID CVE-2018-8440. Mar. 2018. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8440 (visited on 09/25/2018).

[11] André DeHon et al. "Preliminary design of the SAFE platform". In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. ACM. 2011, p. 4.

[12] U. Dhawan et al. "Hardware Support for Safety Interlocks and Introspection". In: *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Sept. 2012, pp. 1–8. DOI: 10.1109/SASOW.2012.11.

[13] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755. URL: http://doi.acm.org/10.1145/2663716.2663755.

[14] Isaac Evans et al. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity". In: *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'15)*. San Jose, CA, May 2015.

[15] Maxim Goncharov. *Heartbleed Vulnerability Affects 5% of Select Top Level Domains from Top 1M*. URL: http://blog.trendmicro.com/trendlabs-

security-intelligence/heartbleed-vulnerability-affects-5-of-top-1-million-websites/.

[16] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 336–345. ISSN: 0362-1340. DOI: 10.1145/2813885.2737979. URL: http://doi.acm.org/10.1145/2813885.2737979.

[17] Galen C Hunt and James R Larus. "Singularity: rethinking the software stack". In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2nd ed. 2018, Section 4. URL: https://doc.rust-lang.org/stable/book/second-edition/.

[19] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[20] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[21] D. R. Kuhn, M. S. Raunak, and R. Kacker. "An Analysis of Vulnerability Trends, 2008-2016". In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. July 2017, pp. 587–588. DOI: 10.1109/QRS-C.2017.106.

[22] Eugen Leontie et al. "Hardware-enforced Fine-grained Isolation of Untrusted Code". In: *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*. SecuCode '09. Chicago, Illinois, USA: ACM, 2009, pp. 11–18. ISBN: 978-1-60558-782-0. DOI: 10.1145/1655077.

1655082. URL: http://doi.acm.org.libproxy.mit.edu/10.1145/1655077.1655082.

[23] Amit Levy. *Design of Tock*. URL: https://www.tockos.org/documentation/design.

[24] Amit Levy et al. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 234–251.

[25] Amit Levy et al. "Ownership is Theft: Experiences Building an Embedded OS in Rust". In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS '15. Monterey, California: ACM, 2015, pp. 21–26. ISBN: 978-1-4503-3942-1. DOI: 10.1145/2818302.2818306. URL: http://doi.acm.org/10.1145/2818302.2818306.

[26] Amit Levy et al. "The Case for Writing a Kernel in Rust". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: ACM, 2017, 1:1–1:7. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124717. URL: http://doi.acm.org/10.1145/3124680.3124717.

[27] Alex Light. "Reenix: Implementing a unix-like operating system in rust". PhD thesis. Master's thesis, Brown University, Department of Computer Science, 2015.

[28] Toshiyuki Maeda and Akinori Yonezawa. "Formal to Practical Security". In: ed. by Véronique Cortier et al. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Writing an OS Kernel in a Strictly and Statically Typed Language, pp. 181–197. ISBN: 978-3-642-02001-8. DOI: 10.1007/978-3-642-02002-5_10. URL: http://dx.doi.org.libproxy.mit.edu/10.1007/978-3-642-02002-5_10.

[29] Nicholas D Matsakis and Felix S Klock II. "The rust language". In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[30] Santosh Nagarakatte et al. "CETS: Compiler Enforced Temporal Safety for C". In: *SIGPLAN Not.* 45.8 (June 2010), pp. 31–40. ISSN: 0362-1340. DOI: 10.1145/1837855.1806657. URL: http://doi.acm.org/10.1145/1837855.1806657.

[31] Santosh Nagarakatte et al. "SoftBound: Highly compatible and complete spatial memory safety for C". In: *ACM Sigplan Notices* 44.6 (2009), pp. 245–258.

[32] Weidong Shi, Chenghuai Lu, and Hsien-Hsin S. Lee. "Transactions on High-Performance Embedded Architectures and Compilers I". In: ed. by Per Stenström. Berlin, Heidelberg: Springer-Verlag, 2007. Chap. Memory-Centric Security Architecture, pp. 95–115. ISBN: 978-3-540-71527-6. DOI: 10.1007/978-3-540-71528-3_7. URL: http://dx.doi.org.libproxy.mit.edu/10.1007/978-3-540-71528-3_7.

[33] Zachary Stengel and Tevfik Bultan. "Analyzing Singularity Channel Contracts". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: ACM, 2009, pp. 13–24. ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572275. URL: http://doi.acm.org.libproxy.mit.edu/10.1145/1572272.1572275.

[34] L. Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[35] Jörg Thalheim et al. "Cntr: Lightweight OS Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association. 2018.

[36] Xi Wang et al. "Towards optimization-safe systems: Analyzing the impact of undefined behavior". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 260–275.

[37] R. N. M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[38] Michal Wegiel and Chandra Krintz. "XMem: Type-safe, Transparent, Shared Memory for Cross-runtime Communication and Coordination". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 327–338. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375621. URL: http://doi.acm.org.libproxy.mit.edu/10.1145/1375581.1375621.

[39] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. "Scale and performance in the Denali isolation kernel". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 195–209.

[40] Dan Williams et al. "Device Driver Safety Through a Reference Validation Mechanism". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 241–254. URL: http://dl.acm.org.libproxy.mit.edu/citation.cfm?id=1855741.1855758.

[41] J. Woodruff et al. "The CHERI capability model: Revisiting RISC in an age of risk". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201.

[42] Min Xu et al. "Towards a VMM-based usage control framework for OS kernel integrity protection". In: *Proceedings of the 12th ACM symposium on Access control models and technologies*. ACM. 2007, pp. 71–80.