SentinelBoot

A demonstrative, maximally memory safe, cryptographically secure bootloader for RISC-V written in Rust

The University of Manchester

Department of Computer Science

Lawrence Hunter

Supervisor: Dr. Pierre Olivier

Abstract

Memory safety is a persistent issue in software, especially system software such as bootloaders, due to stringent performance overheads necessitating minimal runtime checks. Exploiting the vulnerabilities that arise from a lack of memory safety leads to myriad issues, including data leaks, denial-of-service, and arbitrary code execution. Therefore, this thesis proposes SentinelBoot. SentinelBoot is a demonstrative cryptographically secure bootloader aimed at enhancing boot flow safety of RISC-V through memory-safe principles, predominantly leveraging the Rust programming language with its ownership, borrowing, and lifetime constraints. Additionally, SentinelBoot employs public-key cryptography to verify a kernel's hash (digital signature), augmented by the RISC-V vector cryptography extension, thereby establishing secure boot functionality. SentinelBoot achieves these objectives with a 20.1% hashing overhead (approximately 0.27s additional runtime) and produces a resulting binary one-tenth the size of an example U-Boot binary with half the memory footprint.

Acknowledgements

I express my sincerest gratitude to my family, partner, and friends for their unwavering support during the peak of these challenges. I would also like to extend my gratitude to my supervisor, Dr. Pierre Olivier, for his expert knowledge and continuous support. Finally, I express my deepest gratitude to Codethink for sponsoring this thesis as part of an industrial partnership with The University of Manchester.

Contents

Figures

Tables

Chapter 1

Introduction

The problem SentinelBoot aims to solve is contextualised, explained, and motivated throughout this chapter. Additionally, the success criteria, work realised, and challenges are outlined.

1.1 Context

System software such as bootloaders, operating systems, and device drivers are predominantly written in C [1], an example being U-Boot (Universal Bootloader), a widely used open source bootloader which is 94.5% C [2]. In recent years, the headache of memory safety vulnerabilities, including financial losses and reputational damage from security breaches, has driven development away from these unsafe languages [3]. The high performance requirements of system software has prohibited alternatives such as Java, Go, or Python from being viable [4]. With the recent introduction of Rust, it has become possible to phase out C by putting tighter constraints on the source code.

1.2 Problem

System software is one of the final frontiers which has not taken advantage of memory safe languages due to performance overheads; the Rust programming language presents itself as a viable option [4]. Rust has already been explored for device drivers and unikernel operating systems, including Rust UDP [5] and RustyHermit [6] respectively, though few projects target bootloaders, which are a cornerstone of a computer system's security. Vulnerabilities which are reduced by Rust include data races, null pointers, and use-after-free [7], though Rust does not guarantee eliminating these vulnerabilities. Rust also makes use of explicit unsafe lines of code which encourage safe practices.

1.3 Motivation

Memory safety is a persistent issue: a huge number of security bugs are caused by memory safety errors, and continue to arise during development in unsafe languages such as C/C++ [8]; Heartbleed, which resulted from a flaw in the OpenSSL's heartbeat extension, caused a buffer over-read, which allowed attackers to extract sensitive data from server memory [9]. While there is significant focus on preventing memory safety issues through means such as static code analysis or ISO26262 standardisation, these means can only go so far [10, 11], and especially struggle with low-level dedicated microcontrollers [12]. While it can be reasoned whether unsafe code passes these checks, as the number of unsafe lines increases it becomes more challenging to reason about the safety [13], and as such problems begin to arise in being able to definitively reason about the memory safety of larger Linux/SoC systems [14].

It is very difficult to reason about the safety of a system if the process which initialises it is unsafe and susceptible to exploitation [15]. Therefore, this thesis will focus on the earliest stages of the bootflow before an OS/kernel is loaded: these stages are overwhelmingly written in unsafe languages, mostly C and assembly. RISC-V will be chosen as the target architecture as its open source nature allows for supporting an OSS community. The thesis will form a demonstration of limiting unsafe code lines without affecting functionality.

1.4 Aims and Objectives

SentinelBoot is written solely in Rust and assembly, with assembly only used where no alternative is possible. Therefore, using a limited number of unsafe lines of code, SentinelBoot should be able to:

A1 Boot Linux produced by a standard toolchain.

A2 Cryptographically verify a kernel at runtime.

A3 Cryptographically verify a kernel at runtime using vector cryptography.

A4 Communicate current operations through a serial console.

A5 Produce a binary which is of a sensible size proportional to the subset of features to ensure efficient and practical deployment on a target system.

A6 Produce a project structure that employs good practices for scalability and maintainability.

1.5 Work Realised

SentinelBoot replicates U-Boot's functionality to boot a standard Linux kernel [16] (A1). Firstly, to accomplish the objectives outlined, the development of a Rust binary able to execute from an entry point with no operating system support, including the implementation of a serial driver (A4), was performed.

Finally, SentinelBoot's implementation diverges from U-Boot's functionality to implement a secure boot mechanism, which cryptographically verifies a loaded kernel by performing a digital signature check, providing a root of trust [17] (A2). This involved hashing the kernel and combining the result with the server's public key to ensure a hash match, and also performing a signature check to ensure that the kernel was not tampered in transit. Additionally, SentinelBoot accelerates the hashing operation by utilising the RISC-V vector cryptography extension (A3). Next, support was added to meet the kernel's booting requirements before handing execution to a trusted kernel; an onboard ROM chip storing a cryptographic key is not utilised, instead a remote server is used as the root of trust. Finally, throughout the development process SentinelBoot's structure focused on maintainability to add additional board support (A6) and minimise unnecessary external crate dependencies to minimise bloat (A5).

1.6 Challenges

SentinelBoot focuses on a niche part of system software which is not often discussed or mainstream. This results in the concepts used being poorly documented, exacerbating the difficulty of reaching a level of understanding such that they could be implemented. Further, the embedded nature means there is nothing 'watching' the execution; it is not possible to debug without specialised JTAG interfaces on hardware, and significant work must be undertaken before any sign of life is observed via a serial console. Overall, the high learning curve and lack of tooling maturity prevents standard test driven development approaches.

1.7 Evaluation Strategy

The binary will be inspected to determine the composition and relative sizes of crates and libraries that form it compared to similar OSS implementations, specifically U-Boot. This will ensure the ability to operate under the same constrained conditions.

The binary will be compared against U-Boot to determine booting overhead, particularly with verification; further, the data will be analysed to determine standard deviation in results (performed both in emulation and on hardware) to ensure the ability to perform its task within a reasonable time difference.

Finally, to determine the overhead from hashing, the size of the kernel will be varied by compiling in modules.

1.8 Success Criteria

SentinelBoot should be able to successfully boot the current long term support (LTS) Linux (5.10). The resulting binary should be less than 700 kB in size which is the approximate size of a U-Boot binary (shown in Section 4.2) and able to boot Linux in less than 3 seconds, which is approximately twice the boot time with U-Boot (shown in Section 4.1). SentinelBoot should be able to compile in less time than the U-Boot binary, approximately 11 seconds (shown in Section 4.3). Additionally, SentinelBoot should be successfully able to execute on both hardware and under emulation. Finally, SentinelBoot should

minimise unsafe line count the degree to which will be judged subjectively as it is disingenuous to treat U-Boot as 100% unsafe.

1.9 Outline

The background outlines the required concepts to motivate the development work and related work. The methodology is split into three sections: describing and discussing the development work to transition from machine code to unsafe Rust, unsafe Rust to safe Rust, and booting a signature-checked kernel. The evaluation assesses to what extent SentinelBoot achieves the success criteria. Finally, the contributions are summarised and final remarks given on the limitations within SentinelBoot and what future work could improve them, with an ending reflection on the development work.

Chapter 2

Background

This chapter will outline the necessary background information to further contextualise the thesis, and provide specific accounts of the described problem. These topics include:

1. Memory safety principles and vulnerabilities, in order to identify the vectors SentinelBoot will guard against.
2. The Rust programming language, to detail and explore the advancements offered.
3. RISC-V, to explore the architecture and why its infancy is important.
4. Bootflow, to analyse the process which SentinelBoot must adhere to.
5. How existing codebases are written and vulnerabilities within them, in order to motivate the need for change.
6. A look at direct alternatives, to motivate how SentinelBoot is different.

2.1 Memory Safety

Memory safety is a fundamental programming concept that focuses on the secure and reliable management of memory resources, addressing the potential risks and vulnerabilities of memory operations, aiming to prevent unauthorised memory accesses, leaks, and corruptions that can produce crashes and security breaches. The following key principles and associated common vulnerabilities are based upon Hui Xu *et al.*'s 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs' [18].

2.1.1 Key Principles

Valid Memory Access

'Valid memory access' involves guaranteeing that a program only accesses memory locations that have been properly allocated to it. Invalid memory accesses, such as reading from or writing to unallocated or deallocated memory regions, can lead to unpredictable behaviour including segmentation faults or access violations. Memory access can be categorised into privileged and unprivileged access.

Privileged access refers to operations that require elevated privileges, such as accessing kernel memory or hardware registers. Unprivileged access is usually more straightforward to control, as the operating system isolates user-space processes from one another, preventing direct access to other processes' memory. However, ensuring valid privileged memory access becomes more challenging as it is not possible to simply isolate memory spaces, and resulting errors can have severe consequences, including system crashes and security breaches. Proper validation and sanitisation of memory addresses is essential in both privileged and unprivileged contexts to prevent unintended memory access and potential exploits.

Memory Leak Prevention

A memory leak occurs when a program fails to release allocated memory after it is no longer needed. Over time, these leaked memory blocks can accumulate, gradually depleting available memory and potentially causing performance degradation. Memory leaks not only lead to inefficient memory usage but also pose security risks. An unbounded memory leak could be exploited by malicious attackers to exhaust system resources, leading to denial-of-service (DoS) attacks. If the leaked memory contains sensitive data or cryptographic keys, it can be accessed maliciously, compromising the confidentiality and integrity of the application.

Locks and Mutexes

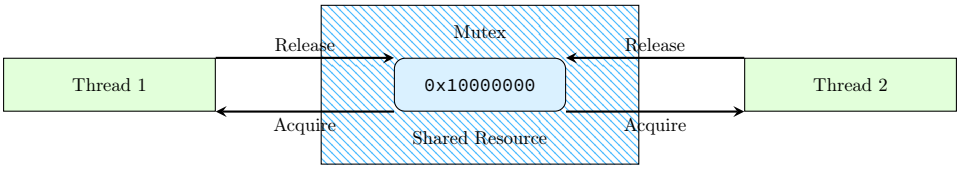

Figure 2.1: Shared resource example

Locks (or mutexes) provide mutual exclusion: they allow only one thread at a time to access shared resources by acquiring a lock before accessing shared data and releasing it afterward. Threads can coordinate their activities and avoid concurrent modifications. Figure 2.1 illustrates a theoretical mutex around a shared memory location.

Atomic Operations

Atomic operations guarantee that an operation on shared data is indivisible and therefore their execution cannot be affected by context switches. They ensure that no other thread can access the data during the atomic operation, thereby preventing data races (see Section 2.1.2).

Null Pointer Handling

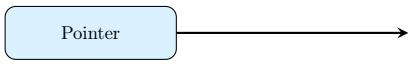

Figure 2.2: Null pointer example

Handling null pointers is another essential aspect of memory safety. A null pointer is a pointer that does not point to any valid objects (in C, address 0). Dereferencing a null pointer can lead to crashes or undefined behaviour. Memory safety necessitates careful handling of null pointers to prevent unexpected program termination or undefined behaviour. Figure 2.2 illustrates a theoretical null pointer vulnerability.

Dangling Pointers

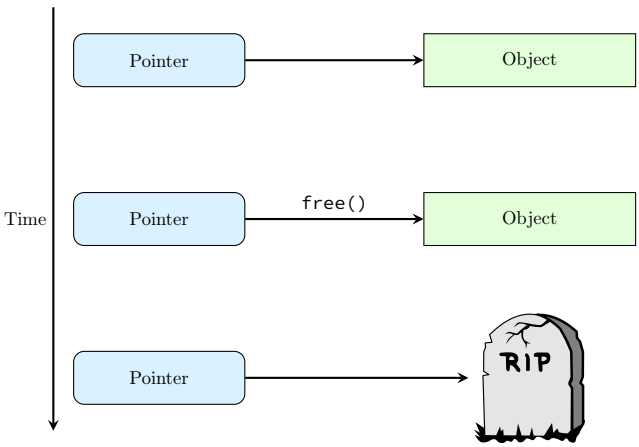

Figure 2.3: Dangling pointer example

A dangling pointer is a pointer that points to a memory location that has been deallocated or which no longer holds valid data. Accessing data through a dangling pointer can lead to undefined behaviour as the memory it points to may have been reallocated for other purposes. Figure 2.3 illustrates a theoretical dangling pointer vulnerability.

2.1.2 Common Vulnerabilities

Memory safety vulnerabilities such as buffer overflows, use-after-free, and data races can lead to exploits.

Buffer Overflows

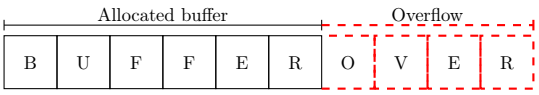

Figure 2.4: Buffer overflow example

Buffer overflows are a type of memory safety vulnerability that occurs when a program writes more data into a fixed-size memory region than it can hold. The excess data spills into adjacent memory regions, potentially overwriting critical data or even function pointers. This can lead to data corruption, crashes, and unauthorised access to sensitive information. In the worst-case scenario, attackers can exploit buffer overflows to inject malicious code into the program's execution flow, enabling arbitrary code execution and compromising the system's security. Figure 2.4 illustrates a theoretical buffer overflow vulnerability.

Use-After-Free

Use-after-free is a vulnerability that occurs when a program continues to use a pointer after the memory it points to has been deallocated. This can happen if a program frees the memory but still retains references to the now-invalid pointer. When the program later attempts to access the memory through the stale pointer it can lead to unpredictable behaviour or security breaches. Figure 2.5 presents an example use-after-free vulnerability written in C.

Use After Free — C

```c
#include <stdlib.h>
#include <string,h>

int main(int argc, char* argv[]) {
    char* ptr = (char*) malloc((40)*sizeof(char))
    if (ptr == NULL) { return 1; }
    free(ptr);
    strcpy(ptr, "Use After Free");
    return 0;
}
```

Figure 2.5: Use After Free

Detecting and debugging a use-after-free vulnerability can be challenging due to its non-deterministic nature: that is, even if the inputs are known, the outcome cannot be predicted with certainty. The program may appear to function correctly until the reused memory is reallocated to another object. Consequently, accessing the memory through the old pointer can lead to unintended consequences and unpredictable behaviour ranging from program crashes and data corruption to more severe security exploits.

Data Races

Data races are a type of memory safety vulnerability that arise in concurrent programs where multiple threads access shared data without proper synchronisation. A data race occurs when at least one thread modifies shared data while other threads are reading from or writing to it simultaneously. This lack of synchronisation can lead to unpredictable behaviour and non-deterministic outcomes, making debugging and maintaining concurrent programs challenging. Data races can result in corrupted data, crashes, or other unexpected behaviours.

Data races occur due to non-atomic shared memory operations in concurrent environments. When multiple threads concurrently access shared data, the order of execution becomes indeterminate, and the final outcome may depend on the relative timing of thread executions. In the absence of proper synchronisation mechanisms, data races can manifest as race conditions.

Race conditions occur when the outcome of a program depends on the interleaving of thread executions, and different interleaving orders can produce different results. This non-determinism can be particularly problematic in critical sections of code where data integrity is crucial. Figure 2.6 illustrates a theoretical race condition vulnerability.

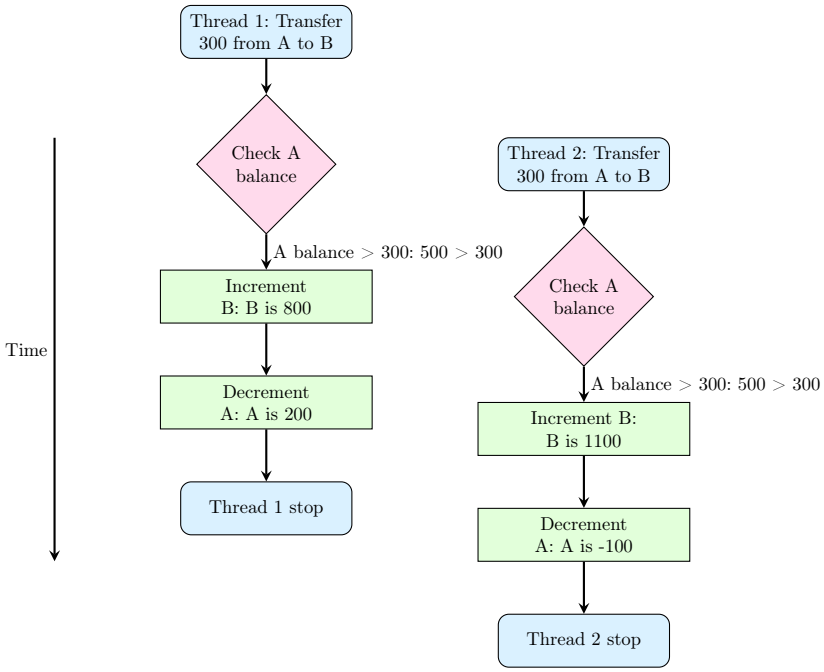

Figure 2.6: Race condition example

2.1.3 Memory Integrity Protection Measures

Ensuring memory integrity involves protecting allocated memory regions from unintended modifications or corruptions, which can compromise program stability and security. Based upon Oscar Llorente-Vazques *et al.*'s 'Detection, exploitation and mitigation of memory errors' [19] and Jinyu Gu *et al.*'s 'EPK: Scalable and Efficient Memory Protection Keys' [20].

Read-Only Memory (ROM): Memory regions are marked as non-writeable, preventing data modifications. This is commonly used for constants and program instructions.

Memory Protection Keys: Some architectures support hardware-based memory protection keys, associating each region with a unique key to control access and prevent unauthorised writes.

Address Space Layout Randomisation (ASLR): Randomising memory layout makes it harder for attackers to predict critical data or code locations, mitigating certain attacks.

Stack Canaries: Defend against buffer overflows by placing a value between stack variables and return addresses. Modification of the stack canaries triggers graceful termination of the program.

Address Sanitiser: A dynamic analysis tool detecting memory bugs such as out-of-bounds accesses and use-after-free vulnerabilities during execution.

2.1.4 Summary

Memory safety vulnerabilities, if exploited, can result in serious consequences, such as critical data leakage and tampering, arbitrary code execution, privilege escalation, and more. It's worth noting that in the environment where a bootloader operates, the focus is primarily on avoiding memory corruption within the region where the kernel will be loaded. Any concerns regarding memory corruption outside this specific area are generally considered less relevant or less severe.

2.2 Rust

The Rust programming language is syntactically similar to C and can likely be understood by a C programmer. While syntactically similar, Rust has additional features and rules, including rules around ownership, borrowing, lifetimes, and a more advanced type system (with optional data types, error data types, and fixed length immutable arrays). In this section, the similarities and differences of Rust and C are explored.

2.2.1 Syntax

Hello, World!

Hello, World! — C

```c
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Hello, World! — Rust

```rust
fn main() {
    println!("Hello, World!");
}
```

Figure 2.7: 'Hello, World!' C and Rust code comparison.

The similarities in the languages are evident in Figure 2.7, with most C developers likely able to intuitively understand Rust code.

Fibonacci

Fibonacci — C

```c
#include <stdio.h>

void calc_fib(int* fib, int target) {
    int temp;
    for (int i = 1; i < target; i++) {
        temp = fib[1];
        fib[1] = fib[0] + fib[1];
        fib[0] = temp;
    }
}

int main() {
    int fib[2] = {0, 1};
    int target = 30;

    calc_fib(fib, target);
    printf("%d\n", fib[1]);
    return 0;
}
```

Fibonacci — Rust

```rust
fn calc_fib(fib: &mut [i32; 2], target: i32)
↪  {
    let mut temp;
    for _ in 1..target {
        temp = fib[1];
        fib[1] = fib[0] + fib[1];
        fib[0] = temp;
    }
}

fn main() {
    let mut fib: [i32; 2] = [0, 1];
    let target = 30;

    calc_fib(&mut fib, target);
    println!("{}", fib[1]);
}
```

Figure 2.8: Fibonacci sequence C and Rust code comparison.

By comparing the two languages in Figure 2.8, it is evident that they exhibit differing approaches to pointers. In Rust, it is possible to be more precise when specifying the type of pointer, for example only accepting a mutable borrow of an i32 array consisting of 2 elements, while in C any integer array is accepted. Despite this distinction, the syntax remains very similar between the two languages.

2.2.2 Memory Safety

Unsafe

For the exploration of unsafe languages, C is used as an example due to its nature as one of the most widely used unsafe languages.

C is a programming language renowned for its lack of memory safety features, granting developers the freedom to manipulate memory as they see fit without constraints imposed by the compiler. While this

flexibility simplifies development, it often results in complex memory safety bugs that require significant debugging efforts. To mitigate these risks, developers adopt practices such as careful code design, static code analysis, and dynamic runtime checks.

This liberty is fundamental to the success of C. An illustrative example is the concept of the 'null garbage collector', where the costs and efforts associated with ensuring memory safety may outweigh the potential risks of memory leaks over the system's runtime [21], such as in the case of a missile guidance system.

Rust

Rust is a memory-safe language which includes an unsafe subset. Developing a program solely in the safe Rust subset guarantees full memory-safety with minimised risk of use-after-free, dangling pointers, or any other kind of memory safety vulnerability [18].

The compiler achieves this through three primary mechanisms: ownership, borrowing, and lifetimes. Ownership ensures that a piece of memory has a single owner at a time, preventing memory leaks by automatically deallocating memory when the owner goes out of scope, e.g. a temporary variable within a function. Borrowing allows multiple pieces of code to have read access to a memory location at any one time, however, if any piece of code writes to the memory location, only one mutable (edit) access can exist and read-only accesses cannot, preventing data races through strict access rules. Lifetimes define the duration of memory, either static for the entire runtime or constrained to specific scopes, thus allowing Rust to avoid using a garbage collector, as when data is determined at compile to be no longer used it can be deallocated.

While these concepts enhance memory safety, developers may encounter challenges while working with the compiler. Rather than compiling and debugging incorrect types, such as using a pointer where an integer was expected, the Rust compiler detects and reports such errors, ensuring code correctness and reliability. In exceptional cases, developers may override the compiler's safety checks by using the `unsafe` keyword, however this approach should be used sparingly and only when absolutely necessary to minimise undermining the protections offered by the Rust language.

No technique is perfect against direct attempts to confuse and bypass them. This is demonstrated in Speykious' `cve-rs` repository, which introduces common memory vulnerabilities written purely in safe Rust [22].

2.3 RISC-V

RISC and CISC are both fundamental CPU architectures: RISC adheres to the principle of simplicity and regularity in its instruction set design; CISC focuses on providing a diverse and feature-rich set of complex instructions.

RISC-V is an open-source Instruction Set Architecture (ISA) based on RISC principles utilising extensions such as vector and multiplication operations. RISC-V has gained adoption in industry and academia due to the alternatives x86-64/ARM requiring licences and royalty payments [23].

RISC-V is an important factor in demonstrating Rust's capacity to replace C as the system software programming language of choice. This is due to its infancy and the relatively fewer Rust system software projects which utilise it, and therefore there are fewer projects to take inspiration from.

The x86-64 assembly code in Figure 2.9 showcases the ability to manipulate the stack in just 3 instructions, demonstrating its more complex nature compared to the RISC-V assembly code in Figure 2.10 which requires 7 instructions for the same task.

Squaring an integer — x86-64 Assembly

```
square:
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rbp-4], edi
    mov eax, DWORD PTR [rbp-4]
    imul eax, eax
    pop rbp
    ret
```

Figure 2.9: x86-64 Assembly for squaring an integer

Squaring an integer — RISC-V Assembly

```
square:
    addi sp, sp, -32
    sd ra, 24(sp)
    sd s0, 16(sp)
    addi s0, sp, 32
    sw a0, -20(s0)
    lw a0, -20(s0)
    mulw a0, a0, a0
    ld ra, 24(sp)
    ld s0, 16(sp)
    addi sp, sp, 32
    ret
```

Figure 2.10: RISC-V Assembly for squaring an integer

2.4 Bootflow

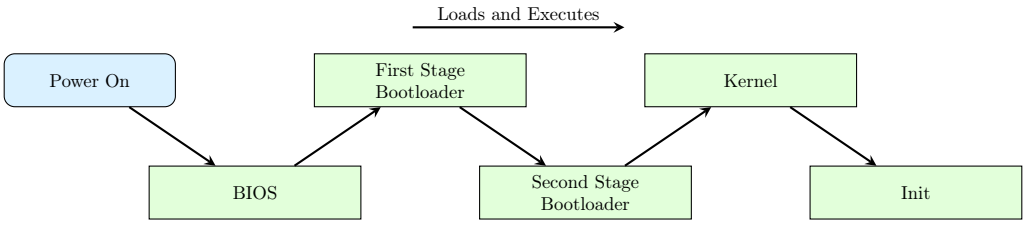

Figure 2.11: Linux bootflow example

Figure 2.11 illustrates the Linux boot process which involves:

1. Initialising hardware components
2. Loading of the bootloader
3. Loading of the initial ramdisk into memory
4. Executing the bootloader to transfer control to the kernel
5. Kernel then initialises the system's core functionalities before transitioning to the user-space environment

2.4.1 First-stage bootloader

First-stage bootloaders reside on fixed disks or removable drives and must fit within the first 446 bytes of the Master Boot Record [24]. Due to the highly restricted nature and limited scope resulting from the byte constraint, first-stage bootloaders are not the primary focus of this thesis. Examples of first-stage bootloaders include BIOS, coreboot, and Libreboot.

2.4.2 Second-stage Bootloaders

Second-stage bootloaders are not themselves operating systems, rather they load an operating system and transfer execution to it [24]. These bootloaders may offer additional functionality such as allowing users to select which operating systems to boot, entering safe mode, or operating without an operating system such as the GNU GRUB shell. Given the broader scope of second-stage bootloaders, this is the target of this thesis.

The boot process is considered to be complete when the system is ready to interact with the user, or when the operating system is capable of running system and application programs. Examples of second-stage bootloaders include GNU GRUB, BOOTMGR, or iBoot.

2.5 Breakdown of the Linux Booting Process

The Linux boot process proceeds through the following sequential steps [25], with provided examples of existing codebases:

BIOS: The Basic Input/Output System (BIOS) conducts integrity checks on the HDD/SSD, and then locates, loads, and executes the bootloader program, which can be found in the MBR. Once the bootloader program is identified and loaded into memory, the BIOS hands over control of the system.

First-stage bootloader - MBR: Situated in the first sector of the bootable disk, the MBR is responsible for loading and executing the second-stage bootloader, thereby initiating the boot process.

Second-stage bootloader - GNU GRUB: The GRUB splash screen typically appears first during boot, facilitating the selection of kernel; GRUB utilises kernel parameters to determine the kernel's location.

Kernel: The kernel first mounts the root file system and then executes the `/sbin/init` program. The kernel establishes a temporary root file system using the initial RAM file system, `initrd`, until the actual file system is mounted.

Init: At this stage, the system executes run-level programs; depending on the Linux distribution, various services such as `sendmail` may start up.

2.6 How are these Tools Written?

- BIOS - coreboot [26]:
  - Roughly 562,389 lines of code[1]
  - C: 94.0%
  - C++: 0.6%
  - Assembly: 0.5%
  - Other (e.g. markdown or make): 4.9%
- MBR - MBR boot manager [27]:
  - Roughly 5,523 lines of code[2]
  - Assembly: 99.4%
  - Other: 0.6%
- Second-stage bootloader - GNU GRUB [28]:
  - Roughly 487,057 lines of code[2]
  - C: 91.9%
  - Assembly: 4.2%
  - Other: 3.9%

It is evident that these tools primarily utilise Assembly, C, and C++, all of which are memory unsafe languages. Despite significant efforts towards memory safety in these projects [24], the inherent unsafety of these languages makes it highly likely that bugs either already exist or bugs will arise during development, for instance the Heap-Based Buffer Overflow in `Sudo` allowed unprivileged users to gain root privileges on the vulnerable host due to out-of-bound characters not being included in its size calculation of the `user_args` buffer [29]. This bug, introduced during development, went unnoticed for nearly 10 years.

[1] Calculated by `git ls-files | xargs wc -l`

2.7 Memory Safety Exploits

Memory safety exploits are specific instances where a memory safety vulnerability is leveraged to accomplish an attacker's objectives. It's crucial to recognize that, while such attacks are feasible, they are not simple, and exploiting a memory safety vulnerability requires expert knowledge and skill.

2.7.1 BIOS Memory Exploits

BIOS rootkits pose significant threats due to their difficulty in detection and resilience against security measures, as evidenced by the National Security Agency's (NSA) use of `DEITYBOUNCE` [30].

For instance, LoJax employed a multistep process for BIOS exploitation [31]:

1. Circumventing a platform's safeguards against unauthorised firmware updates, involving system data gathering due to platform specificity.
2. Extracting the contents of the system's SPI flash memory (where BIOS/UEFI is located) to a file.
3. Injecting the malicious module into the firmware image and rewriting it to the SPI flash memory.

The method of BIOS patching in the third step varies, ranging from exploiting misconfigured platforms to bypassing SPI flash memory write protection, sometimes involving race conditions. While updating the BIOS can mitigate known exploits, it's a non-trivial task requiring specific procedures, such as using a USB drive for updating, beyond the capability of most users. Thus, prioritizing memory safety practices can significantly reduce the likelihood of such exploits, considering the severe ramifications of bugs such as LoJax, which allowed attackers remote access to compromised systems.

2.7.2 First-stage Bootloader Memory Exploits

The MBR itself lacks specific documented direct memory safety exploits, however it remains vulnerable to modification by userspace malware such as `MBR-ONI`, which modifies the MBR to inject malicious code into the boot flow [32].

2.7.3 Second-stage Bootloader Memory Exploits

The `CVE-2020-10713` vulnerability represents the most serve vulnerability found within the GNU GRUB project to date [33]. This buffer overflow exploit affected all versions of GNU GRUB prior to 2.06 and occurred during the parsing of the `grub.cfg` file. Exploiting this vulnerability enabled attackers to bypass the secure boot protections, and so facilitated the loading of untrusted or modified kernels, and granted them high-privilege and persistent access to compromised systems.

The exploit could be executed by direct access, manipulation of `pxe-boot`[2], or remote access to networked systems with root privileges. Once exploited, attackers could perform arbitrary commands, including altering the boot process, patching the OS kernel, and other malicious activities.

2.8 Existing Codebases

2.8.1 U-Boot

U-Boot is an open-source bootloader widely used in embedded systems [16]. Known for its adaptability, U-Boot can operate on various architectures, as well as being open source, making it a popular choice.

2.8.2 Trusted Firmware-A (TF-A)

TF-A is an open-source bootloader and firmware framework designed for ARM-based systems, especially those requiring security and trustworthiness in their boot process [34]. It plays a pivotal role in establishing a secure boot chain and delivering a trusted execution environment. TF-A is engineered to kickstart a secure boot process, guaranteeing that only authenticated and trusted code is executed during system initialisation. It also verifies the authenticity of subsequent bootloader stages and the operating system. TF-A is responsible for maintaining the security state of the system, which encompasses managing cryptographic keys, secure storage, and ensuring the integrity of the boot process at each stage.

[2] A set of standards that enables a computer to load an operating system (OS) over a network connection

2.8.3 Other

Bootloaders are developed in various programming languages to target different hardware platforms, although only a few gain mainstream adoption. One such notable project is Rust OSDev's experimental pure-Rust x86 bootloader [35].

Chapter 3

Methodology and Design

This section introduces SentinelBoot's threat model, including the use case, particularly threat vectors, and why existing preventions do not go far enough. Secondly, the tools used during development and the rationale behind their selection are explored. Thirdly, the development process for implementing functionality in SentinelBoot to execute from an assembly entry point to a Rust function is described. Fourthly, the development of serial drivers and a global memory allocator to enhance functionality is explored. Finally, the development of kernel integrity and authenticity verification techniques, as well as execution hand off to the kernel, are explored. Figure 3.1 illustrates the development structure.

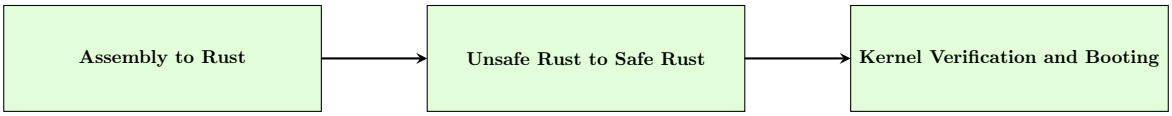

Figure 3.1: Development structure

3.1 Threat Model

Xiong Wenjun and Robert Lagerström in 'Threat modeling - A systematic literature review' state "threat modeling is a process that can be used to analyze potential attacks or threats, and can also be supported by threat libraries or attack taxonomies" and "provides a structured way to secure software design, which involves understanding an adversary's goal in attacking a system based on system's assets of interest" [36].

3.1.1 Our Use Case

SentinelBoot targets thin client devices that do not store their own kernel at boot time, instead obtaining one from a trusted server. Whilst this may not be the operation of this specific device, it is feasible for a device related to Google's Chromebooks to load their kernel from a trusted server upon boot. The kernel pulled from the trusted server is verified using a digitally signed hash, ensuring its integrity and authenticity, thus providing secure boot.

3.1.2 Attacks

Evil Maid and Social Engineering

Both 'evil maid' and 'social engineering' attack vectors utilise direct access to the client hardware to modify it in a way that is undetectable to the end user. Whilst the compromise is not obvious to the user, malicious goals may include implanting malware, installing keyloggers, or stealing sensitive data. The evil maid specifically focuses on a malicious actor having direct access to an unattended device for a brief period of time, during which they can perform the exploit [37]. On the other hand, social engineering focuses on directly manipulating the user to compromise their own system unknowingly [38].

Man-in-the-middle (MITM)

A MITM (Man-in-the-Middle) attack refers to a malicious technique where an unauthorised third party intercepts and potentially alters communication between two parties [39]. An attacker positions themselves between the legitimate sender and receiver, allowing them to eavesdrop on, manipulate, or inject malicious content into the exchanged data, as illustrated by Figure 3.2. The goal of a MITM attack is to secretly control or disrupt communication by undermining the confidentiality, integrity, and authenticity of the data exchanged between the legitimate parties.

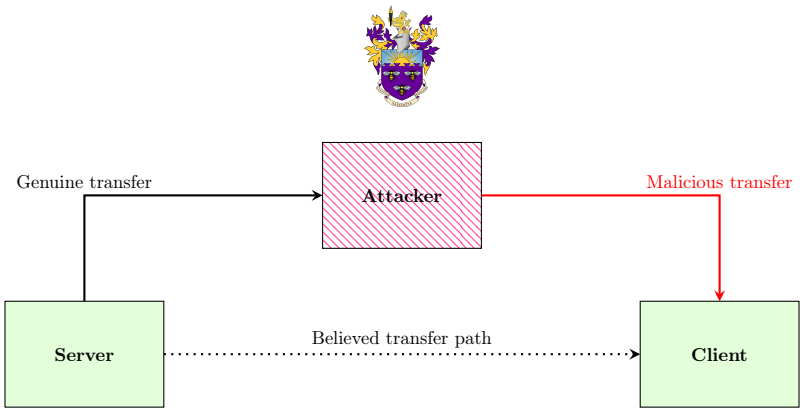

Figure 3.2: Example man-in-the-middle attack scenario

Critical Memory Exploit

Through a memory vulnerability, an attacker is able to perform an exploit. This exploit can range in severity from full remote code execution to a read primitive.

3.1.3 Preventions

Trusted Platform and Trusted Execution

Trusted Platform Modules (TPMs) are specialised hardware modules that provide a range of security functions, including cryptographic key storage and cryptographic functions [40]. Trusted Execution Environments (TEEs) are secure sections of the processor that isolate execution of sensitive code. Both TPMs and TEEs can be utilised to implement secure boot functionality and attestation, forming a root of trust. Although RISC-V supports the implementation of both of these measures, they are not widely available and are a subject of research [41]. Unfortunately, this renders them non-viable for use in SentinelBoot.

Compilers

The Rust compiler can detect many memory safety vulnerabilities through its employment of ownership, borrowing, and lifetime rules; however, as shown in Peykious' `cve-rs` repository, the compiler is not able to catch all vulnerabilities [22].

Summary

Given the nature of a bootloader, if an attacker gains physical access, such as an evil maid attack or through social engineering, the binary can simply be swapped out for the attacker's. Since TEEs and TPMs are not available, this represents a significant security risk. Instead, the attack vector SentinelBoot focuses on defending against is a physical attack that does not involve replacing the binary, instead modifying the server by which SentinelBoot derives its root of trust. Additionally, given the kernel binary's transfer through the network, MITM attacks pose a serious threat vector that needs to be mitigated. Finally, it is imperative to adhere to good memory safety practices to minimise the probability of vulnerabilities in the source code, as the compiler cannot catch them all.

3.1.4 Goals

An attacker would seek to circumvent the protections provided by SentinelBoot to enable the loading of a modified kernel by the thin client. The specific actions of this modified kernel could range from granting full backdoor access to the network with arbitrary code execution provided to the adversary, to the implementation of a well-intentioned patch by a third party.

3.2 Inspiration

The project structure of SentinelBoot drew inspiration from Rust Embedded's Operating System development tutorials in Rust on the Raspberry Pi [42]. This tutorial was followed prior to the start of SentinelBoot's development and proved useful. Consequently, the Raspberry Pi (ARM64) specific project structure was modified to target RISC-V, incorporating adjustments in drivers, assembly code, the build system, and linking scripts.

3.3 Continuous Integration (CI)

Continuous integration (CI) is a development methodology focused on enhancing the speed, efficiency, and dependability of software delivery processes [43]. Automated workflow practices enforce code quality standards, conduct static code analysis, and verify compliance with coding conventions. Additionally, workflows can automate the creation and publication of software releases, reducing the manual effort required for versioning and distribution. Overall the use of automated workflows improve SentinelBoot's code quality and readability.

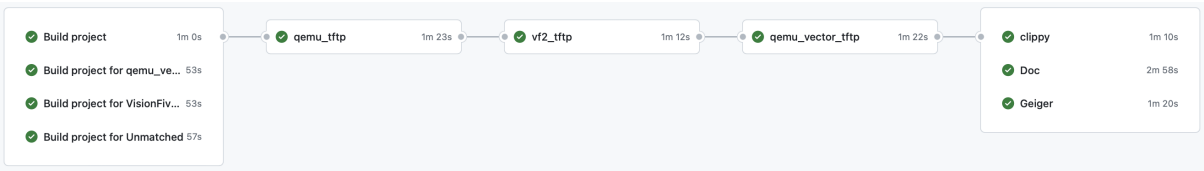

Figure 3.3: Example of GitHub Workflows

Workflows included in SentinelBoot's continuous integration pipeline are depicted in Figure 3.3 and can be summarised as follows:

Build: Compiles the entire project for all targets, ensuring that broken code is not merged into the main branch.

Booting: Uses the compiled binaries to verify that, on each respective target, the binary functions as expected and successfully boots Linux.

Quality Control: Validates that the source code adheres to coding standards, ensures successful compilation of the documentation, and reports the state of unsafe line usage.

3.4 Tools of the Trade

3.4.1 QEMU

QEMU (Quick EMUlator) serves as a highly customisable emulator capable of translating binary code to the host's Instruction Set Architecture (ISA) from a range of different hardware and device models [44]. It supports running a wide range of operating systems and allows interfacing with physical host hardware including network cards, USB devices, and hard disks; further, QEMU can emulate its own hardware including network cards, serial ports, and CAN interfaces.

In the context of this project, QEMU is utilised to emulate a full RISC-V system, allowing for significantly shorter development cycles. Booting a virtual RISC-V board and verifying its output can all be included within the build script. Moreover, QEMU supports remote GDB debugging, simplifying SentinelBoot's debugging process.

QEMU was preferred over alternatives such as VirtualBox due to being considerably easier to manipulate through bash scripts, drastically speeding up testing cycle times. Additionally, QEMU supports more RISC-V extensions, including vector cryptography, which is used to accelerate cryptographic operations within SentinelBoot [45, 46]. Finally, from personal experience, QEMU achieves high performance.

3.4.2 Raspberry Pi

Raspberry Pis are affordable, credit-card sized computers which are capable of running full Linux systems, particularly Raspbian [47]. As a cheap, small, and Linux capable computer with a range of ports including USB and Ethernet, it is ideal for a rig controller to interact with RISC-V boards.

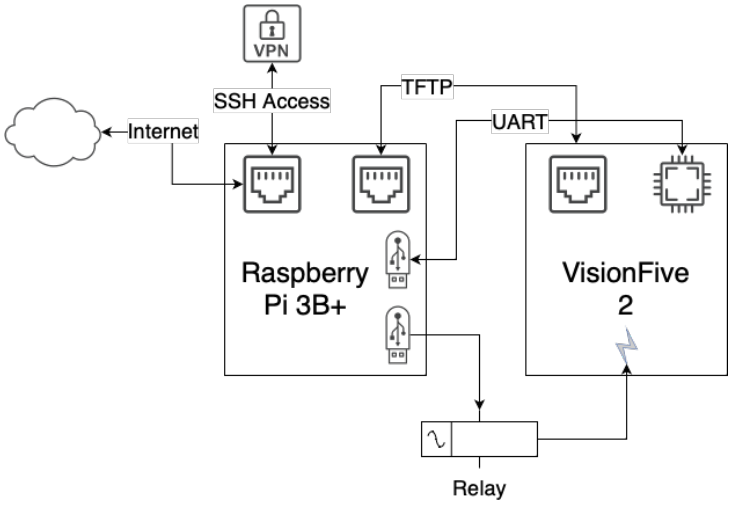

Figure 3.4: Raspberry Pi rig controller setup

The Raspberry Pi serves as a rig controller for the VF2 (VisionFive 2) RISC-V board, chosen due to its availability and good performance. As depicted in Figure 3.4, the Raspberry Pi connects to the VF2 board via Ethernet, serial (UART), and through a relay. This setup enables remote power cycling, booting, and serial communication with the VF2 board. Setting the Raspberry Pi to be remotely accessed outside the local network allows for its use in continuous integration workflows for testing SentinelBoot on hardware.

3.4.3 Docker

Docker is an open-source containerisation platform that enables developers to package applications and their dependencies into lightweight, portable containers [48]. These containers are isolated, lightweight units that bundle an application along with its runtime libraries and dependencies, ensuring consistent and reliable deployments across different environments. Docker was chosen over alternatives such as Podman due to familiarity and better documentation.

Docker is used to create a consistent build environment for SentinelBoot across the multiple machines which build it, including macOS, Arch Linux, and Raspbian.

3.5 Assembly to Rust

Reaching Rust code from the boot process is not as straightforward as branching to it, therefore it is necessary to examine how SentinelBoot's binary is formed, and the setup required beforehand, including generating the binary, understanding the control flow from the entry point, and configuring the relevant control registers.

3.5.1 Linking

Linking is the final stage of compilation [49]. It involves amalgamating a set of binaries, arranged in a defined address space, into a single unified binary. Executable and Linkable Format (ELF) is a common standard file format for executable files; Figure 3.5 illustrates a simplified linking process.

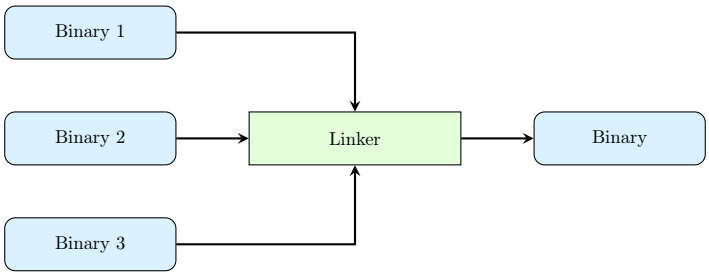

Figure 3.5: Simplified linker combining binaries

How the final SentinelBoot binary is linked is specified by a linker script (LD script). A linker script is written in the linker command language, outlined in Figure 3.6, and its main purpose is to describe how the sections in the input file should be mapped to the output file and control the memory layout of the output file. The result forms an ELF binary with the structure illustrated in Figure 3.7.

Basic LD script — text

```
SECTIONS
{
. = 0x10000;
.text : { *(.text) }
. = 0x8000000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

Figure 3.6: Basic LD script

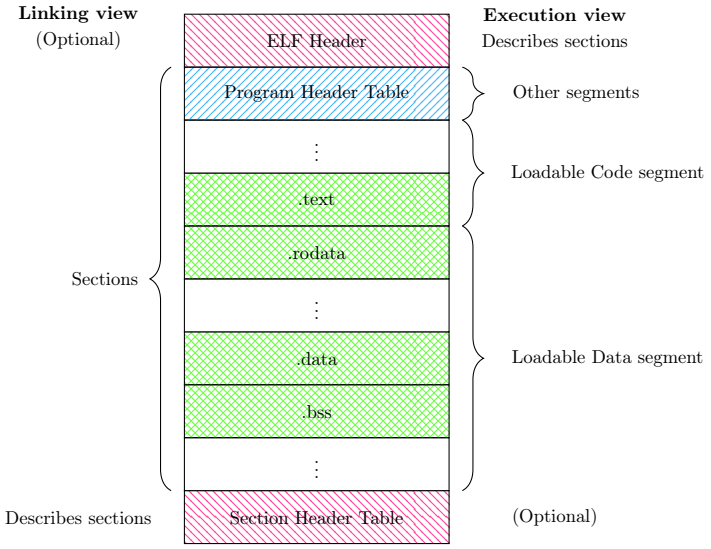

Figure 3.7: Common ELF binary structure

3.5.2 Assembly

Before reaching SentinelBoot's Rust code, the hardware must be in a state able to execute it [50]. Due to the linker script, the SentinelBoot assembly begins executing from the `_start` label. Moving from assembly to Rust requires handling the multiple hardware threads (HARTs) including the interrupt status and operating mode of each HART. To initialise the hardware state, SentinelBoot needs to perform the following operations:

1. Load the global pointer for accessing global variables and data structures.
2. For all but one HART, branch to splitting the stack amongst HARTs, enabling interrupts, setting the trap vector, setting the return address, and setting the Rust entry point for non-main HARTs.
3. For the other HART, initialise the block starting symbol (BSS) section to zero, load the stack pointer, set the status to machine mode, disable interrupts, and load the Rust entry point.
4. All HARTs then branch to Rust in supervisor mode. Reaching Rust in supervisor mode is important as it limits the level of control over the system, reducing risk from exploits [51].

Control status registers (CSRs) serve a wide range of functions, including interrupt enabling for each privilege level, HART information, and floating point handling [52]. CSR registers that are important for setting the hardware state in machine mode include `mtvec`, `mepc`, `mie`, `mstatus`, and `satp`. Additionally, the CSR registers which are important for setting the hardware state in supervisor mode include `stvec`, `sepc`, `sie`, `sstatus`, and `satp`

{m,s}tvec: Machine/Supervisor trap-vector (a vector is essentially a function pointer). The base-address register holds two values: the `BASE`, which is the function pointer to the trap handler (an assembly function to handle some event), and `MODE`, which specifies whether to use direct or vectored interrupts. Direct interrupt mode means all traps go to the exact same function; vectored will call different functions based on the trap, i.e. an offset in a branch table.

{m,s}epc: Machine/Supervisor exception program counter. When a trap is taken into {M,S}-mode, the corresponding `{m,s}epc` is set to the virtual address of the instruction that triggered the exception, used for returning once exception is handled. It can also be explicitly written to.

{m,s}ie: Machine/Supervisor interrupt enable register. This can be written to setting which interrupts are enabled.

{m,s}status: Machine/Supervisor status register. Each HART has one of these, and each keeps track of and controls its respective HART's current operating state.

satp: Supervisor address translation and protection. This holds the physical page number (`PPN`) of the root page table, an address space identifier (`ASID`), and `MODE` field. `MODE` sets the current address-translation scheme, which SentinelBoot does not use; thus, it is set to zero.

Figure 3.8 illustrates SentinelBoot's control flow from machine mode entry point to reaching Rust. The illustration includes the branch for all but one HART and its control flow, as well as any context switches, illustrated with dotted lines.

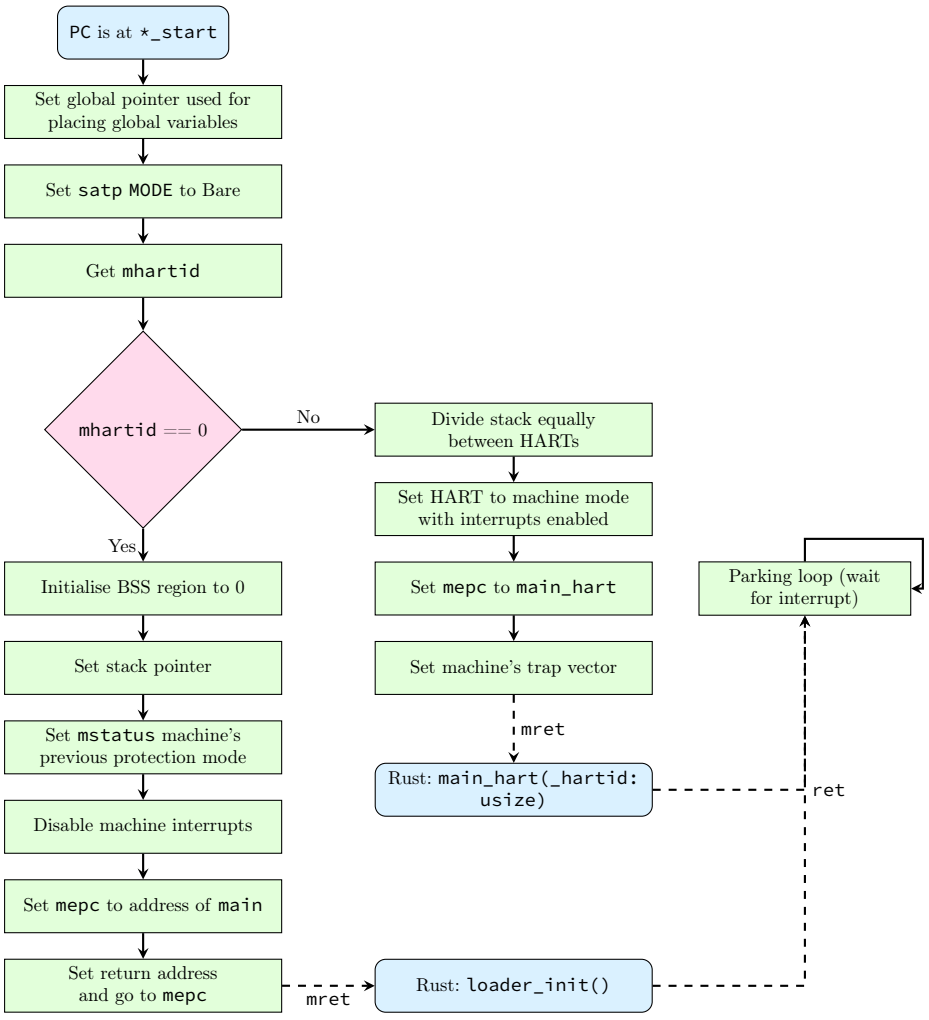

Figure 3.8: Raw RISC-V assembly to Rust

It is necessary to be able to initialise the hardware state in both machine and supervisor mode due to a development decision to avoid writing an additional driver for Ethernet, with the intention of maintaining

the goals of SentinelBoot to be achievable within the time frame. Therefore, U-Boot was used to perform trivial file transfer protocol (TFTP) operations on SentinelBoot's behalf: these TFTP operations would communicate with the root of trust server to download the kernel into memory, though a caveat arises: U-Boot hands execution to SentinelBoot (once TFTP operations have been performed) in supervisor mode. Consequently, two sets of assembly code are needed depending on the target.

Figure 3.9 illustrates SentinelBoot's control flow from supervisor mode entry point to reaching Rust. Additionally, as U-Boot hands execution, it does so only on a single HART and as such it is not necessary to handle the multi-HART case.

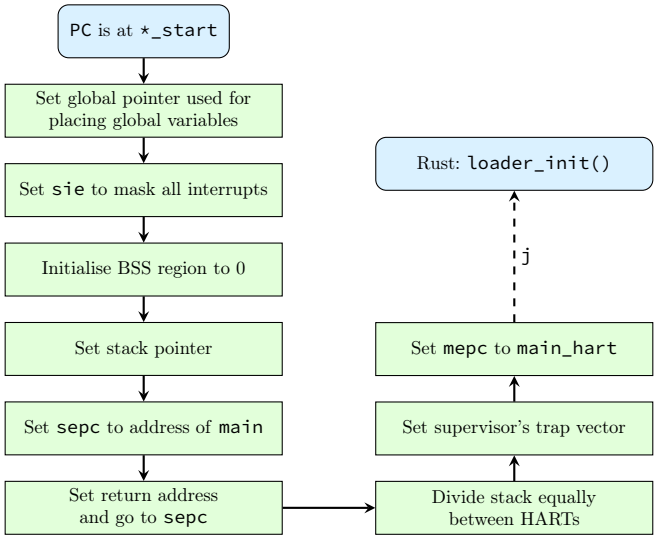

Figure 3.9: Supervisor RISC-V assembly to Rust (U-Boot use case)

3.5.3 The Rust Branch

After the execution of the assembly code, SentinelBoot's control flow is now dictated by Rust. While this transition is symbolic, there are no drivers or a memory allocator initialised, therefore additional specifically unsafe work is needed to enable full operation.

3.6 Unsafe Rust to Safe Rust

SentinelBoot's control flow is now dictated by Rust; drivers and a memory allocator need to be initialised to enable full functionality. The current state is not strictly unsafe nor entirely composed of unsafe lines of code, however, all the following setup requires unsafe memory access to initialise the wrappers, therefore pre- and post-initialisation phases will be symbolically classified as an unsafe-to-safe switch.

3.6.1 The Serial Driver

The implementation of a serial driver marks a distinctive milestone in SentinelBoot: it represents the first instance where debugging SentinelBoot becomes feasible without relying on external tools such as GDB and OpenOCD. This achievement significantly eases the debugging process and accelerates development speed.

Universal Asynchronous Receiver/Transmitter (UART) defines a simple protocol for exchanging serial data between two devices [53]. One of the main advantages of UART is its asynchronous nature, meaning there is no need for a shared clock signal. To allow communication without a shared clock signal, the protocol uses a predefined signal rate such as 9600 bits/s or 115200 bits/s. The two devices must also use the same frame structure and parameters. For example, Figure 3.10 illustrates an expected `7o1` frame where 7 data bits are used with one bit used for odd parity, and Figure 3.11 presents a real world `7o1` frame obtained from a serial analyser.

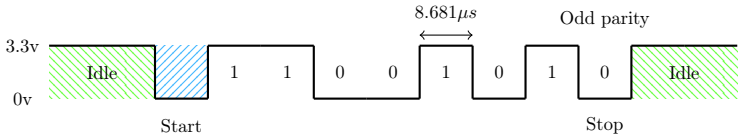

Figure 3.10: UART frame 115200 `7o1`

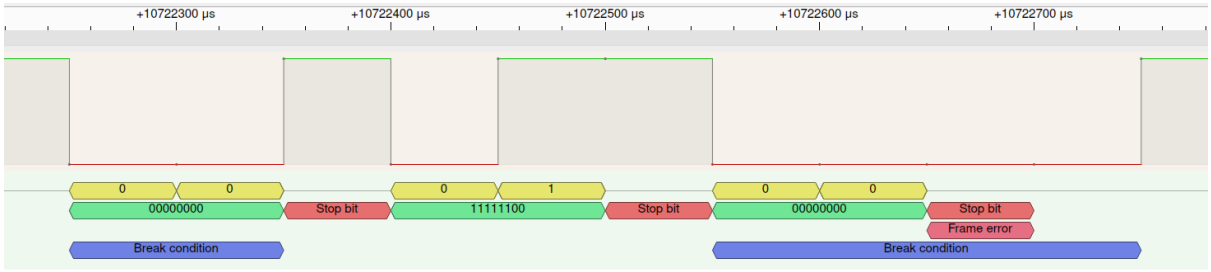

Figure 3.11: UART frames 115200 `7o1` captured using a serial analyser

To accommodate all three of SentinelBoot's targets, serial drivers for `VIRT16550A`, `DW8250UART`, and `FU740-C000` are required. All three are implementations of the same protocol, there are only slight differences between them, therefore only `VIRT16550A` is used for the example. `VIRT16550A` is memory mapped to address `0x10000000` and contains seven registers as shown in Table 3.1.

Address	Register	Access Type	Reset Value	Description
`0x00`	`RBR`	Read only	`0x00`	Receive Buffer Register
`0x00`	`THR`	Write only	`0x00`	Transmitter Holding Register
`0x01`	`IER`	Read/Write	`0x00`	Enable/Disable interrupts
`0x02`	`IIR`	Read only	`0x01`	Information on the interrupt that occurred
`0x02`	`FCR`	Write only	`0x00`	Control behaviour of FIFOs
`0x03`	`LCR`	Read/Write	`0x00`	Only bit used is bit 7 and specifies divisor latch
`0x05`	`LSR`	Read only	`0x60`	UART state

Table 3.1: `VIRT16550A` registers

Modifying the values stored in these registers alters the behaviour of the chip: changes include setting the UART frame, setting the baud rate, and buffer behaviour. A simplified initialisation of the register values can be described by:

1. Set `LCR` to desired bits for UART frame
2. Enable FIFOs
3. Enable receiver buffer interrupts
4. Set divisor latch access bit (multiplexing bit)
5. Set divisor latch to 115200 i.e. `0x0001`
6. Unset divisor latch access bit

To prevent data races from HARTs writing to the same UART driver, a mutual exclusion (mutex) is implemented. Before a HART can read or write from SentinelBoot's serial driver it must acquire the

mutex lock: the lock specifies which HART has control of the driver and blocks all other HARTs from acquiring the lock until the locking HART has finished.

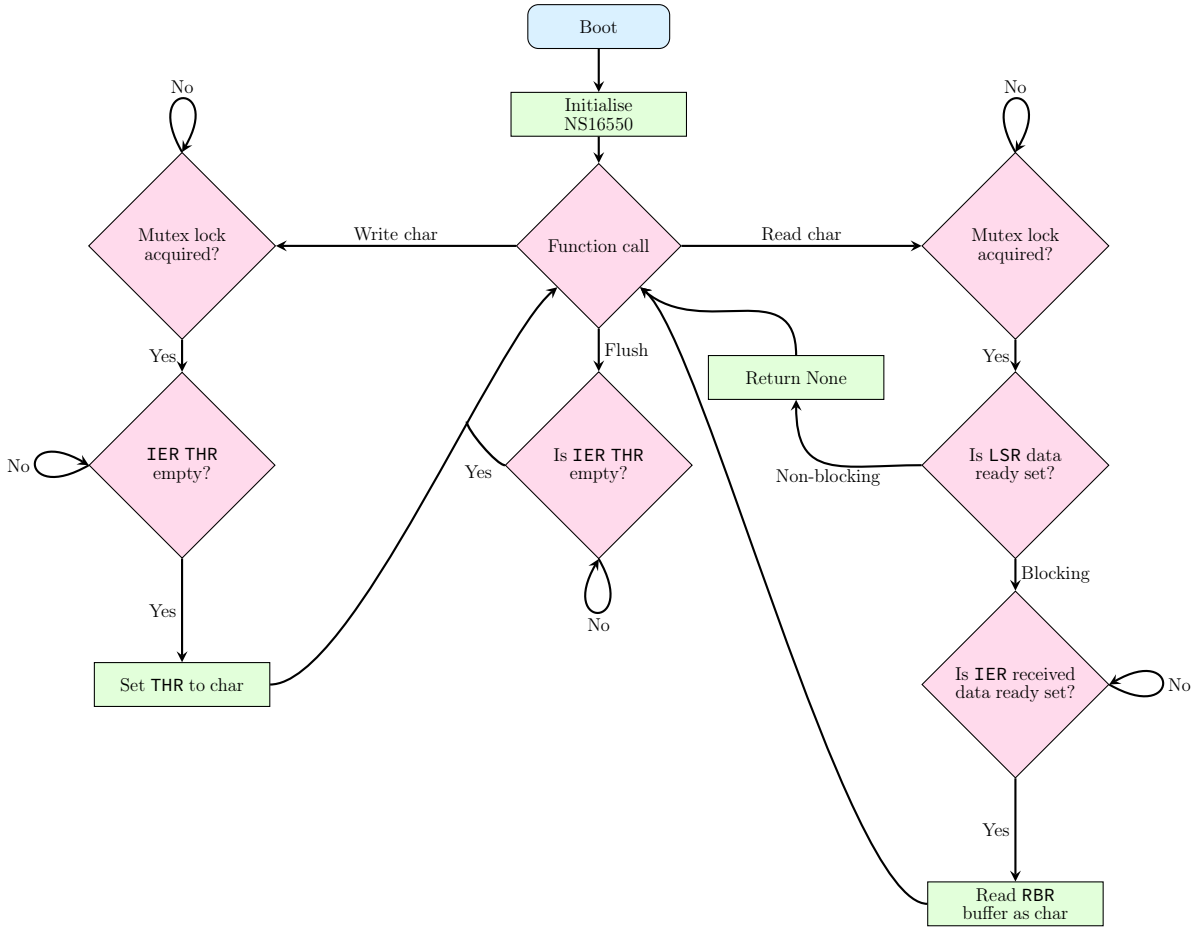

Figure 3.12: `VIRT16550A` UART Driver

3.6.2 The Global Memory Allocator

Within a program there are two types of allocated memory: static and dynamic [54].

Statically allocated memory is known and assigned at compile time by the linker, and it is encoded directly into the binary. These allocations exist for the entire runtime of the program, are located at fixed memory locations, can always be referenced from local variables due to the `'static` lifetime.

Dynamically allocated memory is created and destroyed during the runtime of the program. Each dynamic allocation has an associated lifetime restricting its scope. Additionally, all dynamic allocations must fit into a fixed-size memory region known as the heap and can be modified, including the ability to expand in size, allowing the use of data structures such as vectors. This additional functionality is utilised by crates used within SentinelBoot.

Figure 3.13 illustrates the locations of static and dynamic memory allocations within the ELF binary format. While it is not essential to be able to utilise dynamic memory allocation for SentinelBoot's functionality, doing so allows the use of more Rust features.

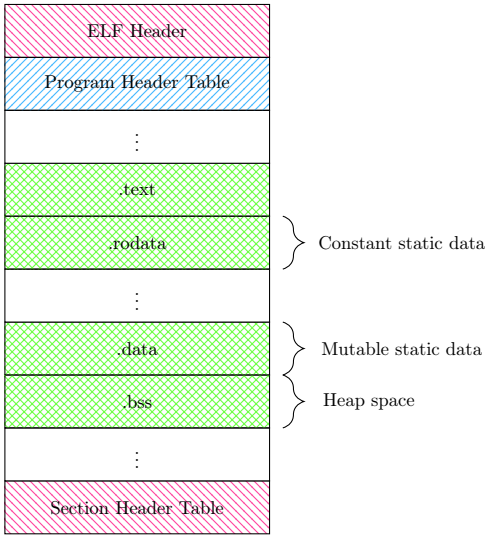

Figure 3.13: Static dynamic allocations in ELF binary structure

Unlike statically allocated memory, dynamic memory allocations cannot be handled by the linker. An implementation of the `GlobalAlloc` trait is necessary to provide allocation functions: `alloc`, `alloc_zeroed`, `dealloc`, and `realloc`.

alloc**:** Create a new allocation of size x and return the pointer to it.

alloc_zeroed**:** Create a new allocation of size x, set all memory locations within the allocation to the value 0, and return the pointer to it.

dealloc**:** Take a pointer to an allocation and mark it as deallocated.

realloc**:** Take a pointer to an allocation and a new size y and return a pointer the new allocation.

The implementation of each function within SentinelBoot requires carefully managing the heap region to separate, log, and satisfying the constraints of each allocation. Figure 3.14 illustrates how allocations can be organised in memory. The need to manage an unknown number of allocations at any point in time without corruption necessitates an additional data structure to store the allocations.

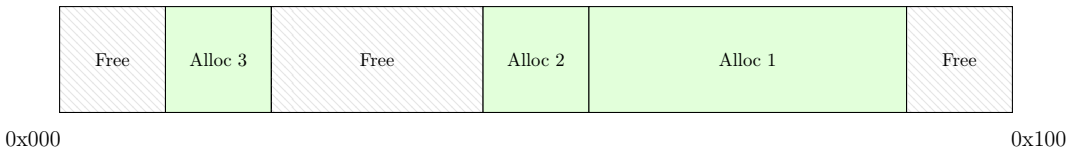

Figure 3.14: Example allocations in a memory region

Many implementations for a dynamic memory allocator exist with varying degrees of optimisation. One of the simplest but still performant implementations uses a doubly linked list[1] of allocation structures to maintain the heap space. The allocation structures, as such, are required to store the two pointers to the next and previous structures, the size of the allocation (for efficiency), and a flag indicating the type, i.e., allocated, free, dead, and root. Figure 3.15 illustrates a possible doubly linked list structure showing the pointers between structures as arrows.

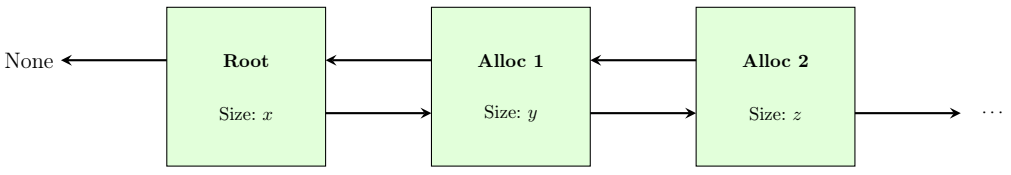

Figure 3.15: Example doubly linked list allocation structure

[1] A doubly linked list is a data structure in which each item contains a pointer to the item before and after it.

As SentinelBoot is written in Rust, it is important to discuss the doubly linked list data type. Since each structure is mutably pointed to by the two structures either side, the list violates Rust's borrowing rules. To implement the doubly linked list, it is necessary to bypass the compiler's borrowing checks. However, it is possible to do so safely by creating a mutex wrapper around each pointer and storing the immutable pointer to this instead, as illustrated in Figure 3.16. This prohibits more than one access to the actual mutable pointer at any point in time, ensuring thread safety.

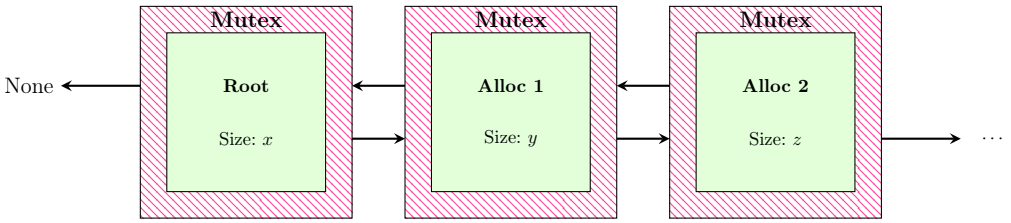

Figure 3.16: Example doubly linked list allocation structure with mutex pointer wrappers

The advantage of a doubly linked list for a dynamic memory allocator is that it allows bidirectional traversal of the data structure. The bidirectional nature makes it computationally efficient to amalgamate allocations, improving the utilisation of memory, as illustrated by Figure 3.17. This is primarily a factor when reallocation is performed, as it is possible to maintain the same alloc structure.

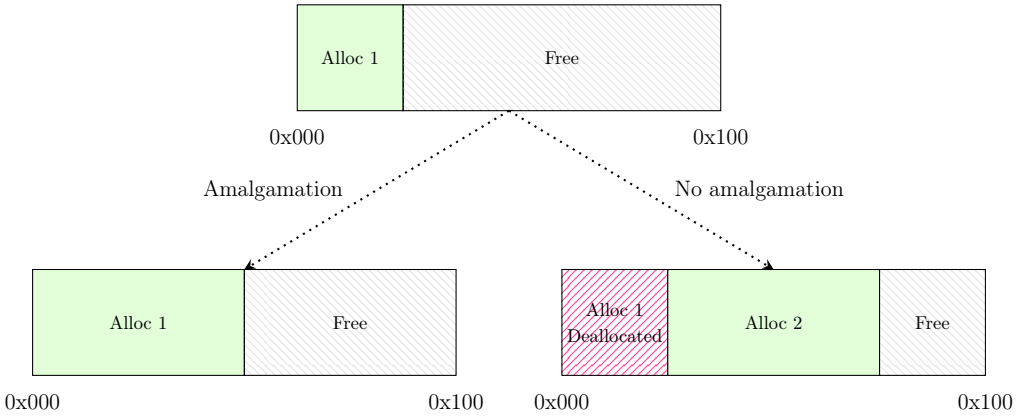

Figure 3.17: Example of memory utilisation with and without realloc amalgamation

3.6.3 GDB, JTAG, and OpenOCD

As no drivers are initialised, it is not possible to print any text from SentinelBoot at this point, making debugging significantly harder. The tools used to enable reaching a working implementation included GDB for debugging and OpenOCD to communicate with the HiFive Unmatched's JTAG.

GDB (GNU Debugger) is a command-line utility and a fundamental tool for debugging and analysing programs [55]. It provides features such as setting breakpoints at specific code locations, stepping through code instruction by instruction, and inspecting the values of variables and registers. These features were utilised to inspect correct values at set points in the control flow to determine correct operation and locating the erroneous instructions otherwise. By itself GDB allowed the debugging of QEMU, and successful output was achieved as demonstrated by Figure 3.18.

Figure 3.18: UART output in QEMU

JTAG (Joint Test Action Group) is a hardware interface standard primarily used for testing and debugging integrated circuits and electronic systems [56]. JTAG provides a standardised interface for accessing internal circuitry, allowing direct access to the chip's pins and registers for software-controlled testing and manipulation of device components.

OpenOCD (Open On-Chip Debugger) is an open-source tool designed to provide an efficient and flexible solution for debugging and programming microcontrollers as well as embedded systems [57]. It serves as a bridge between the development environment and the hardware, enabling interaction with - and control of - on-chip debugging and programming features.

When coupled with OpenOCD, it is possible to use GDB to interact with the hardware running SentinelBoot rather than just with QEMU. Successful output was achieved as demonstrated by Figure 3.19. Debugging both QEMU and hardware is necessary due to the differing drivers required and QEMU's tendency to be forgiving with errors compared to hardware.

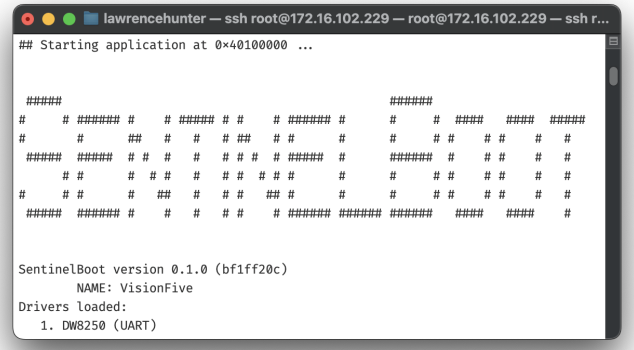

Figure 3.19: UART output on hardware

3.6.4 Unsafe Rust to Safe Rust Branch

With the serial driver and global memory allocator initialised within SentinelBoot, it is possible to make the symbolic branch to the main function from the Rust entry point and continue execution.

3.7 Verification and Booting of The Linux Kernel

For RISC-V, the kernel's boot requirements are not extensive, only expecting the registers `a0` and `a1` to contain the address of the device tree binary (DTB) in memory and the `hartid` respectively [58]. Additionally, SentinelBoot and the kernel need to cooperate in setting up the memory translation and paging mechanisms to establish a consistent memory mapping and virtual memory environment for the operating system.

3.7.1 Kernel Configuration

The kernel is configured by the DTB [59], which is a data structure used in the boot process of many embedded systems, serving as a description of the hardware components present in the system. This allows the operating system to identify and initialise these components during the boot sequence.

The DTB is a compiled data structure originating from a Device Tree Source (DTS) file, which specifies the requirements as a plain text tree structure. To modify the kernel's boot arguments during boot time would necessitate decompiling, modifying, and recompiling the DTB. While it would be ideal for the bootloader to perform this task, there is an alternative approach: the arguments can be compiled directly into the kernel, allowing us to avoid the process at boot time. Given the significant technical effort required to reimplement the device tree compiler in Rust (where it is typically written in C), it is outside the scope of SentinelBoot, however, as the bootloader's kernel verification requires modification of the kernel binary prior to boot in order to digitally sign it with its hash, this limitation is deemed acceptable for its intended use case.

3.7.2 Kernel Hashing

Hashing is a cryptographic technique that converts data of arbitrary length into a fixed length literal [60]. Essential properties of any cryptographic hashing algorithm function h, given an output set Y and input set X, are as follows:

Deterministic: $\forall x \in X. \forall (y_1, y_2) \in Y \times Y. (h(x) = y_1) \wedge (h(x) = y_2) \Rightarrow (y_1 = y_2)$, that is, for an input x, the corresponding output will always be the same hash literal.

Uniform: $\forall x \in X. \forall y \in Y. P(h(x) = y) \approx \frac{1}{|Y|}$, that is every output is equally likely considering every possible input.

Avalanche effect: given an input x, even a small modification will result in a massively different result.

Pre-image resistance: $\forall y \in Y$ finding $x \in X. h(x) = y$ is computationally hard, that is, given a hash value y, it should be difficult to find the associated input x.

Second pre-image resistance: $\forall x_1 \in X$ finding $x_2 \in X. (x_1 \neq x_2) \wedge (h(x_1) = h(x_2))$ is computationally hard, that is, given an input x_1, it should be hard to find an x_2 such that their hash literals are the same.

Collision resistance: Finding $(x_1, x_2) \in X \times X. (x_1 \neq x_2) \wedge (h(x_1) = h(x_2))$ is computationally hard, that is it should be hard to find an x_1 and x_2 such that their hash literals are the same.

Given a hashing function that satisfies these properties, it is possible to verify the integrity of a binary file. If a binary is hashed with this function and its hash is stored, at a later date, it can be verified (with extremely high confidence) that it has not been modified by recomputing the hash and comparing it. Therefore, by extending this verification process to the Linux kernel, by comparing the hash of a given kernel with a stored hash, it is possible to determine if the kernel has been modified before booting, forming part of SentinelBoot's secure boot functionality.

SHA256

Secure Hash Algorithm 2 (SHA-2) is a set of six cryptographic functions, of which SHA256 is one [61]. SHA256 satisfies the aforementioned required properties, is widely used, standardised, efficient, and widely compatible. SHA256 uses a series of bitwise operations, rotations, and additions. The other five functions within SHA-2 offer varying levels of security and performance overheads, though SHA256 strikes the optimal balance for SentinelBoot.

Other hashing functions exist outside SHA-2, including MD5, SHA-1, and SHA-3. SHA-2 offers stronger security than MD5 and SHA-1, which are both known to be vulnerable to collision attacks. SHA-3 offers better security and potentially faster performance than SHA-2, but it is newer and less compatible. Overall, SHA-2 is the optimal hash function set to be used in SentinelBoot's functions.

SHA256 Steps

Given:

$$\texttt{ROTR}^n(x) = \text{Circular right rotation of } x \text{ by } n \text{ bits}$$
$$\texttt{SHR}^n(x) = \text{Right shift of } x \text{ by } n \text{ bits}$$

1. Store an array, k, of 64 32-bit constant values.
2. Take an arbitrary-length input, and pad with 0s until the data is a multiple of 512, with the final 64 bits set to a big endian integer of the original input length
3. Initialise the hash values with eight constants representing the fractional parts of the square roots of the first eight primes, forming an array of eight 32-bit values, H.
4. For each 512 bit chunk:
   (a) Generate message schedule:
      i. Copy the chunk into an array of 32 bit entries
      ii. Append an additional 48 32-bit entries of value 0 forming W
      iii. For each entry $W[i]$ perform, where i is the index $16 \ldots 63$:

$$\begin{aligned}
\sigma^0 &= \texttt{ROTR}^7(W[i-15])\ \texttt{XOR}\ \texttt{ROTR}^{18}(W[i-15])\ \texttt{XOR}\ \texttt{SHR}^3(W[i-15]) \\
\sigma^1 &= \texttt{ROTR}^{17}(W[i-2])\ \texttt{XOR}\ \texttt{ROTR}^{19}(W[i-2])\ \texttt{XOR}\ \texttt{SHR}^{10}(W[i-2]) \\
W[i] &= W[i-16] + \sigma^0 + W[i-7] + \sigma^1
\end{aligned}$$

   (b) Perform compression:
      i. Using the initialised hash values H, initialise eight variables a, b, c, d, e, f, g, h where $a = H[0], b = H[1], \ldots, h = H[7]$
      ii. Perform the following 64 times, where i is the loop count $0 \ldots 63$:

$$\begin{aligned}
S_1 &= \texttt{ROTR}^6(e)\ \texttt{XOR}\ \texttt{ROTR}^{11}(e)\ \texttt{XOR}\ \texttt{ROTR}^{25}(e) \\
Ch &= (e\ \&\ f)\ \texttt{XOR}\ (!e\ \&\ g) \\
t_1 &= h + S_1 + Ch + k[i] + W[i] \\
S_0 &= \texttt{ROTR}^2(a)\ \texttt{XOR}\ \texttt{ROTR}^{13}(a)\ \texttt{XOR}\ \texttt{ROTR}^{22}(a) \\
Maj &= (a\ \&\ b)\ \texttt{XOR}\ (a\ \&\ c)\ \texttt{XOR}\ (b\ \&\ c) \\
t_2 &= S_0 + Maj \\
\{h, g, f, e, d, c, b, a\} &= \{g, f, e, d + t_1, c, b, a, t_1 + t_2\}
\end{aligned}$$

   (c) Update hash values

$$H = \{H[0] + a, H[1] + b, \ldots, H[7] + h\}$$

5. Produce the final hash by concatenating the hash values

$$\texttt{digest} = \texttt{concat}(H[0], H[1], H[2], H[3], H[4], H[5], H[6], H[7])$$

Determining Runtime Kernel Size

Having found SHA256 to be a suitable hashing function for SentinelBoot, it is necessary to find the input. As the kernel binary will be located in memory, it can be viewed as being bordered by an infinite number of arbitrary values, where including even a single extra digit not part of the binary would produce the Avalanche effect. Therefore, it is necessary to precisely and reliably determine the size of the kernel binary.

The Linux kernel is an ELF (Executable and Linkable Format) binary, therefore, given a start address, it is possible to parse the ELF header and read the addresses of the sections within the binary. Utilising section addresses, it is possible to calculate the total size of the kernel binary. Figure 3.20 illustrates summing the section sizes of an ELF binary.

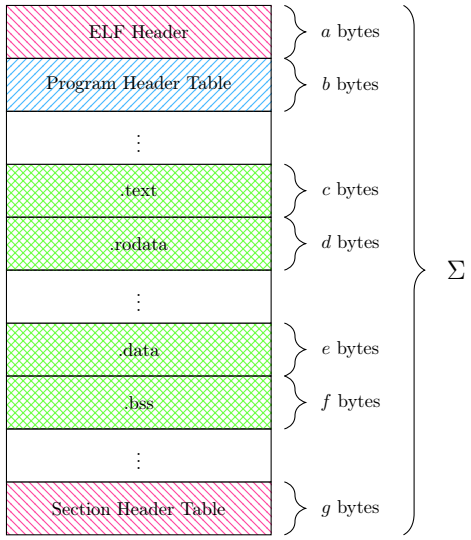

Figure 3.20: Example size of ELF binary calculation

With the kernel size known and the start address known, as it is fixed, it is possible to feed the kernel data into the SHA256 function, producing a hash literal.

Public Key Cryptography

Public key cryptography consists of one or more entities each possessing a private key, which is never shared, and a public key, which is publicly known [62]. The public and private keys are mathematically linked such that one can decrypt messages encrypted with the other. Additionally, the certificate is stored by a trusted certificate authority (CA), an external 3rd party used to verify the public key's authenticity. Without the CA, an attacker could just use their own public key while pretending to be the intended recipient.

Public key cryptography operates on similar principles to hashing but with the additional use of keys.

- It should be computationally infeasible to find the plaintext given the associated ciphertext without the proper key.
- It should be computationally infeasible to determine the private key from the public key.

Typically, for communication between two entities, one entity encrypts the plaintext with the other's public key verified by a certificate obtained from a CA, producing the ciphertext. The ciphertext is then sent to the other entity, which uses its private key to decrypt it, Figure 3.21 illustrates this system.

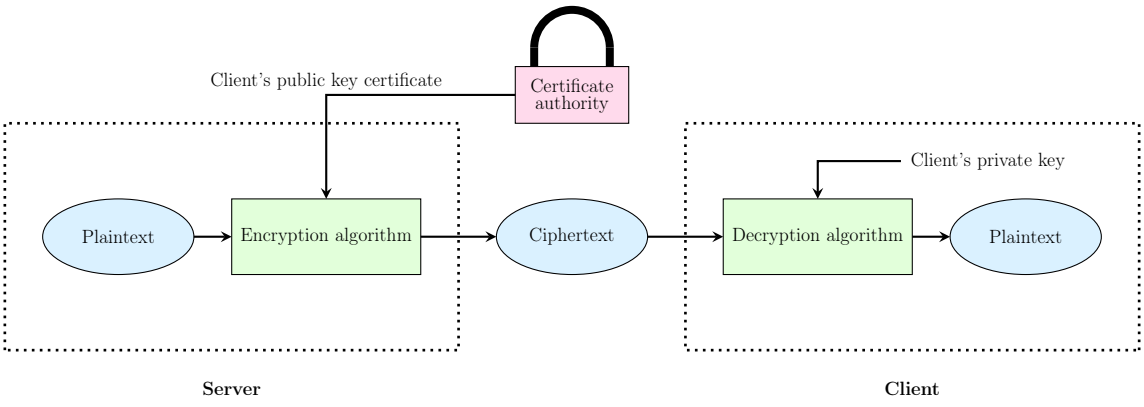

Figure 3.21: Example public key cryptography data exchange

The operations of a popular example of public key cryptography, RSA, can be described as follows:

- Two very large prime numbers are selected, p and q.

- Compute $n = p \times q$
- Compute $\phi(n)$, Euler's totient of n, which is the number of positive integers less than n that are co-prime to n (their only common divisor is 1).
- Choose a number e that is less than and co-prime to n
- Compute the private key, d, which satisfies $(d \times e) \mod \phi(n) \equiv 1$
- The public key is (n, e), and the private key is (n, d)
- To encrypt a message M using the public key perform $C \equiv M^e \mod n$
- To decrypt the encrypted message perform $M \equiv C^d \mod n$

The discrete logarithm problem involves finding the exponent x when given $g^x \mod p$. This problem is believed to be computationally difficult particularly for large prime numbers. This belief is the basis for the security behind public key cryptography.

Digital Signature

The current implementation of a raw hash and a kernel binary being sent to SentinelBoot has left a significant attack vector unaddressed: that being a man-in-the-middle (MITM) attack. An attacker could intercept the hash and kernel during transmission, modify the kernel, rehash it, replace the original hash, and then forward the new modified hash and kernel to SentinelBoot. As the hash and kernel would still match, SentinelBoot would be fooled into booting an untrusted kernel. Additionally, as public key cryptography is too slow, only operating on fixed size data chunks (245 bytes for RSA), it is not feasible to transfer the kernel fully encrypted.

By slightly modifying the use of public key cryptography, it is possible to generate a digital signature. This involves sending the plaintext as well as a hash of the plaintext encrypted with the entity's private key [62]. Any receiving entity can hash the plaintext and decrypt the encrypted hash using the original entity's public key obtained from a certificate authority. Finally, by comparing the two hashes, it can verify that the plaintext has not been modified and is from the original entity. Figure 3.22 illustrates this process.

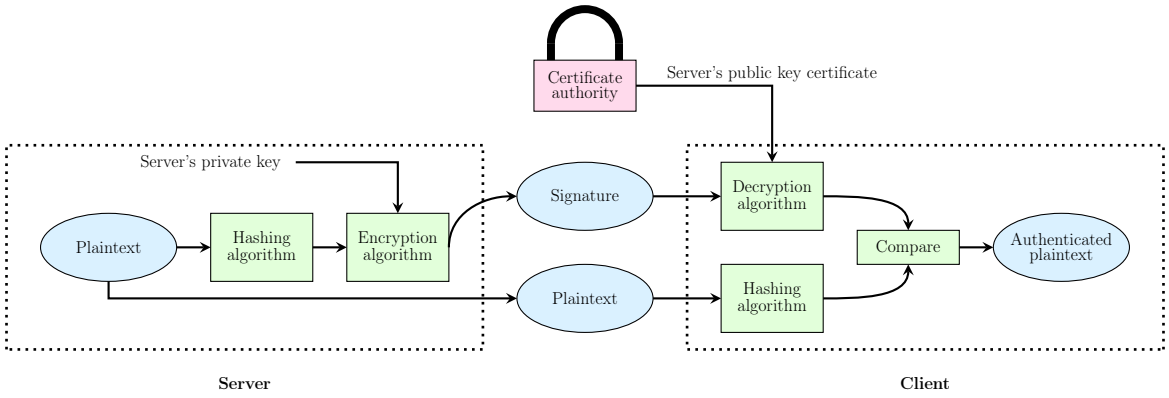

Figure 3.22: Example public key cryptography signature exchange

Instead of sending the hash and kernel binary, sending the digital signature of the kernel binary along with the kernel binary mitigates the MITM attack. This is because it is computationally infeasible for the attacker to fake the digital signature.

The algorithm chosen for the digital signature was ED25519, this is due to being faster and more secure than alternatives such as RSA, DSA, and ECDSA [63].

Summary

The transmission of a signed hash (digital signature) generated from a known-size Linux kernel alongside said kernel allows, with very high confidence, for the verification of the authenticity and integrity of the kernel. This high confidence facilitates the formation of SentinelBoot's root of trust through a known server.

3.7.3 Vector Cryptography

Vector cryptography is a RISC-V ISA extension that aims to accelerate common cryptographic operations [64]. This acceleration is achieved through Single Instruction Multiple Data (SIMD) operations, where the same instruction is applied in parallel to multiple chunks of data [65]. For instance, using eight 128-bit vector registers, it is possible to process 1024 bits per instruction. In comparison, Single Instruction Single Data (SISD) operations would only be able to process a single 128-bit register per instruction, resulting in an eighth of the throughput, Figures 3.23 and 3.24 illustrate this limitation.

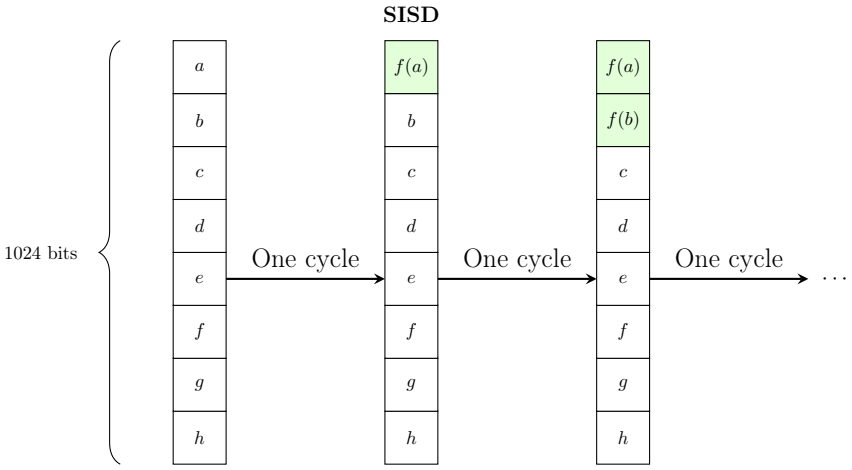

Figure 3.23: SISD operation

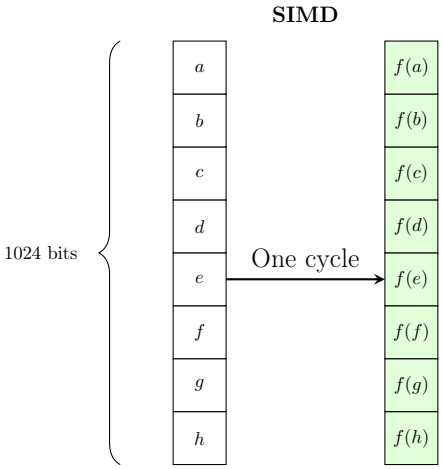

Figure 3.24: SIMD operation

The vector cryptography extension covers a wide range of cryptographic functions including GCM/GMAC, SM3, SM4, AES, SHA-2, and supporting bit manipulation operations. The SHA-2 extension, denoted as `Zvknh[ab]`, specifically consists of three assembly instructions:

`vsha2ms.vv`: This instruction performs two rounds of the SHA-256 message schedule update operation. It takes three vector registers containing four current message schedule values each and outputs one vector register containing the updated four message schedule values.

`vsha2ch.vv`: This instruction performs two rounds of the SHA-256 compression operation. It takes three vector registers, two containing the current state data, and one containing the four message schedule values (although it only uses the two most significant values), then outputs one vector register containing the next state data.

`vsha2cl.vv`: This instruction performs an identical operation to `vsha2ch.vv` but uses the two least significant message schedule values.

By combining these three assembly instructions with supporting vector operations, such as load and store, it is possible to perform the full SHA256 hashing function purely in assembly, with goal of accelerating the SHA256 operation within SentinelBoot. The assembly instruction control flow is illustrated by Figure 3.25.

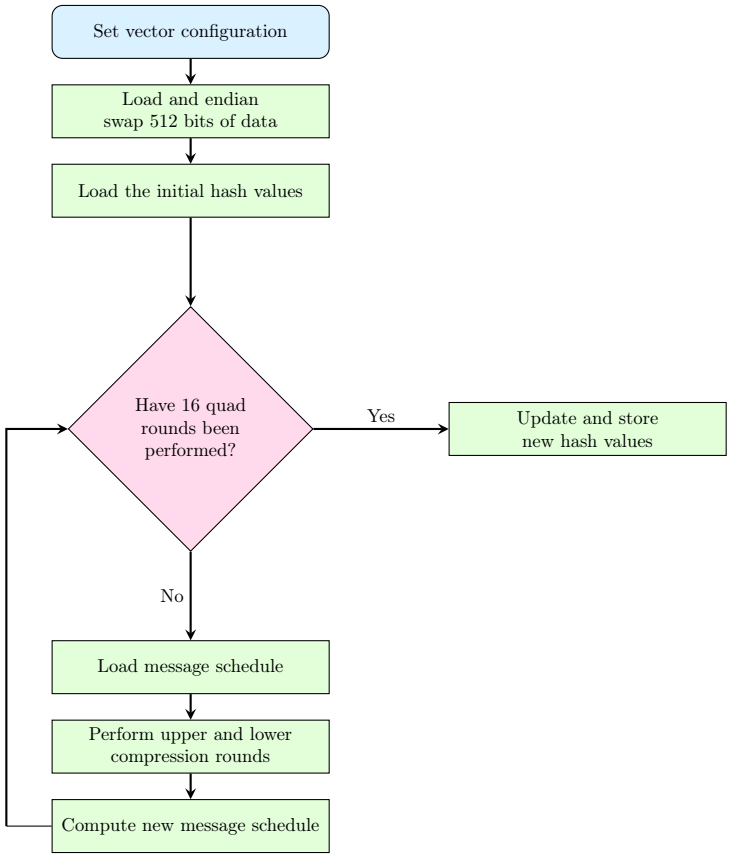

Figure 3.25: Assembly SHA-2 control flow

Currently, no commercially available RISC-V silicon implements the vector cryptography extension, as it was only ratified in September 2023. Therefore, aside from being difficult to reason about the true acceleration achieved, it is only possible to execute vector cryptography operations under emulation, thanks to QEMU supporting the extension fully.

As mentioned, vector cryptography was only recently ratified, and alongside hardware infancy, there is also tooling infancy. This results in the inability to assemble the vector cryptography and vector extension operations with the Rust toolchain. However, it is possible to bypass the need for assembling by pre-assembling the instructions and including the raw machine code within SentinelBoot. For instance, `vsha2ms.vv v10, v14, v13` translates to `0xb6e6a577` or `0b10110110111001101010010101110111`, this process is illustrated by Figure 3.26.

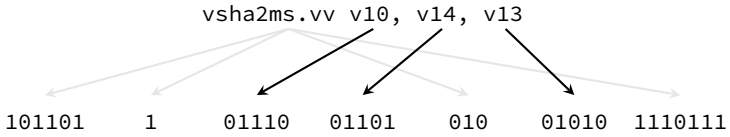

Figure 3.26: Hand assembling example `vsha2ms.vv` instruction

Due to bypassing the need for the Rust toolchain to assemble these instructions, it is possible for SentinelBoot to utilise the acceleration offered by vector cryptography to hash the Linux kernel in order to verify its integrity before booting.

3.7.4 Handing Execution

Once all checks are complete, the correct values can be loaded into `a0` and `a1`, then jump to the kernel. This control flow is demonstrated in Figure 3.27.

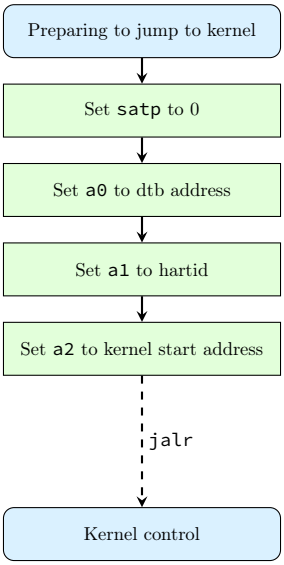

Figure 3.27: Rust jump to kernel control flow

3.7.5 Ghidra

Ghidra is a powerful software reverse engineering (SRE) tool developed by the NSA that aids in analysing and understanding binary executables [66]. It includes a disassembler, decompiler, and a variety of analysis tools. Ghidra is able to disassemble binaries into assembly code, and then decompile them into a high-level language representation, such as C. Ghidra was used over alternatives such as IDA due to familiarity with the tool.

Debugging Kernel Boot

Combining the binary exploration offered by Ghidra with the control flow of GDB, it was possible to monitor the processor throughout the kernel bootflow, identifying assembly instructions (using GDB) which either sent the processor to an infinite loop or were illegal. Working backwards from the assembly code shown in Figure 3.28 using Ghidra, it was possible to view the equivalent check in C code that failed. This process facilitated a vastly reduced cognitive load approach to deciphering runtime errors, as demonstrated in Figure 3.29.

Failed boot check using GDB — Bash

```
0x80202000: csrw sie, zero
0x80202004: csrw sip, zero
0x80202008: auipc gp, 0x12f2
0x8020200c: addi gp, gp, 1384
0x80202010: lui t0, 0x6 (valid compressed)
0x80202012: csrc status, t0
0x80202016: li t0, 8 (valid compressed)
0x80202018: blt a0, t0, 0x80202020
0x8020201c: j 0x802010cc
...
0x802010cc: wfi
0x802010d0: j 0x802010cc
```

Figure 3.28: Failed kernel check assembly code

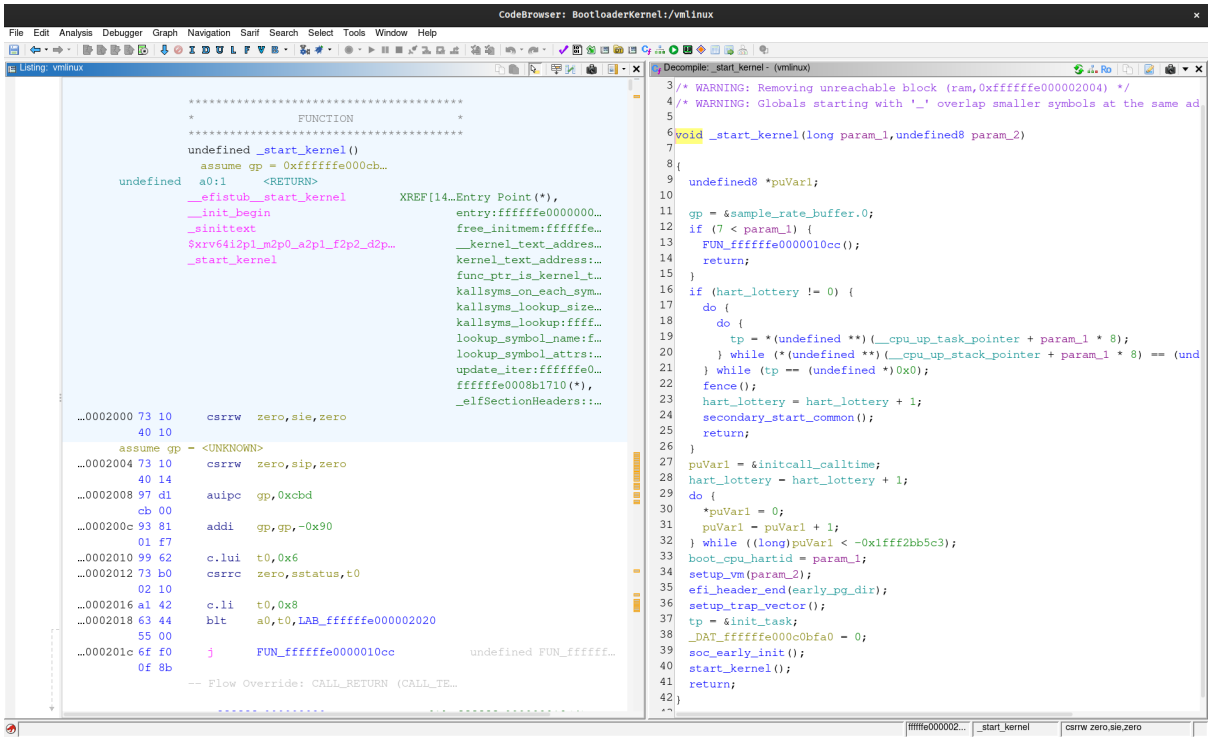

Figure 3.29: Equivalent assembly code address and C code in Ghidra

This approach ultimately led to achieving full kernel booting, as shown in Figure 3.30.

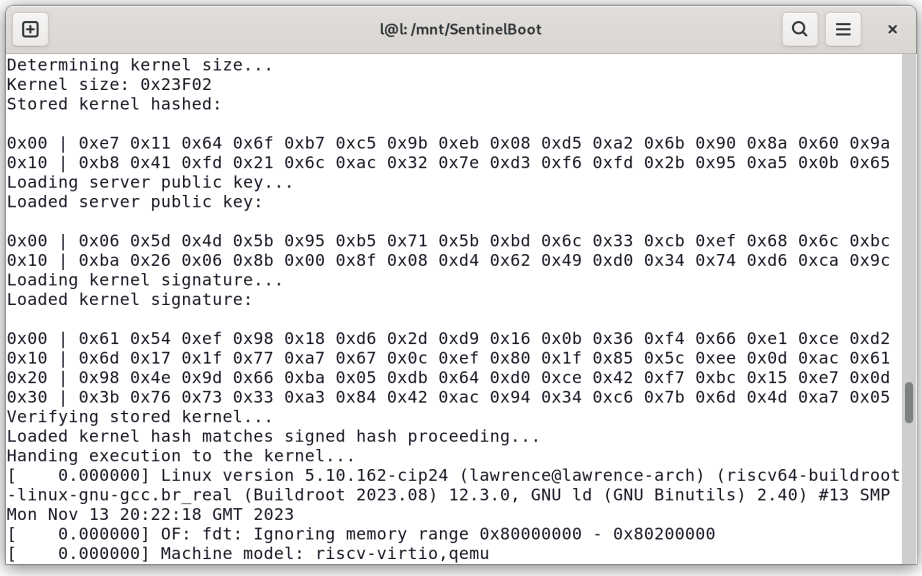

Figure 3.30: Serial output of the kernel hand off

Chapter 4

Evaluation

This evaluation assesses to what extent SentinelBoot achieves the success criteria: being able to cryptographically verify and boot Linux in less than 3 seconds, produce a resulting binary of less than 700 kB, compile in less than 11 seconds, present a strong security model, and minimise unsafe line count as much as possible. Testing was conducted on both emulation and hardware (specifically the VisionFive 2) platforms.

4.1 Boot Time

SentinelBoot aimed to cryptographically verify and boot Linux in less than 3 seconds, i.e. approximately double the execution time of a standard U-Boot binary[1].

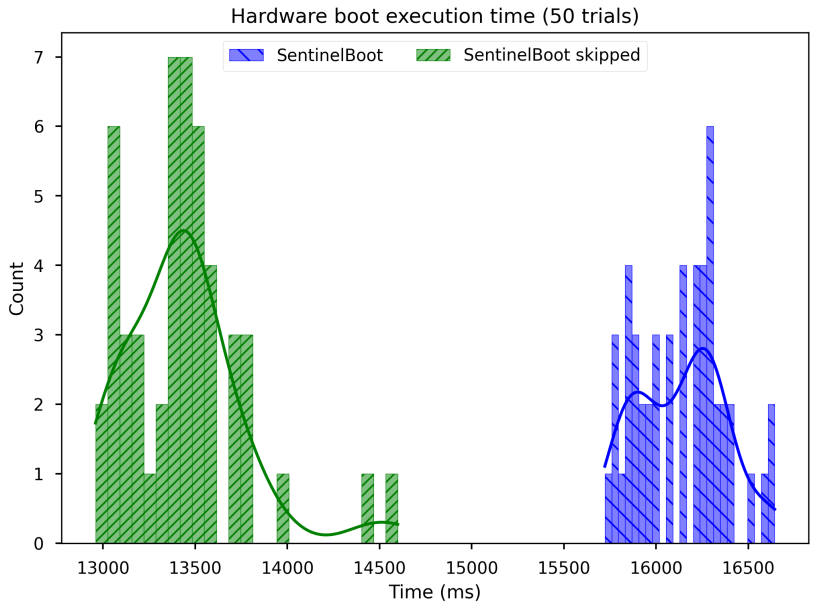

Figure 4.1: Hardware boot time with and without SentinelBoot

Target	Mean (ms)	Standard deviation (ms)
SentinelBoot	16133	239.49
U-Boot	13432	324.79

Table 4.1: Hardware boot time with and without SentinelBoot mean and standard deviation

Figure 4.1 and Table 4.1 show, on hardware, a standard U-Boot binary average execution time of approximately 13400 ms, and a SentinelBoot average execution time of approximately 16100 ms, and a slightly lower standard deviation for SentinelBoot execution times compared to U-Boot directly. The higher execution time of SentinelBoot is significant with a mean difference of approximately 2700 ms,

[1]Standard U-Boot binary is the result of using the target specific default defconfig i.e. `qemu-riscv64_defconfig` and `starfive_visionfive2_defconfig`.

or a 20.1% execution overhead. The slightly lower standard deviation of SentinelBoot's execution time is likely due to the longer execution time balancing throughput fluctuations for U-Boot's tftp transfers. SentinelBoot is therefore able to, on hardware, execute within the threshold laid out in the success criterion.

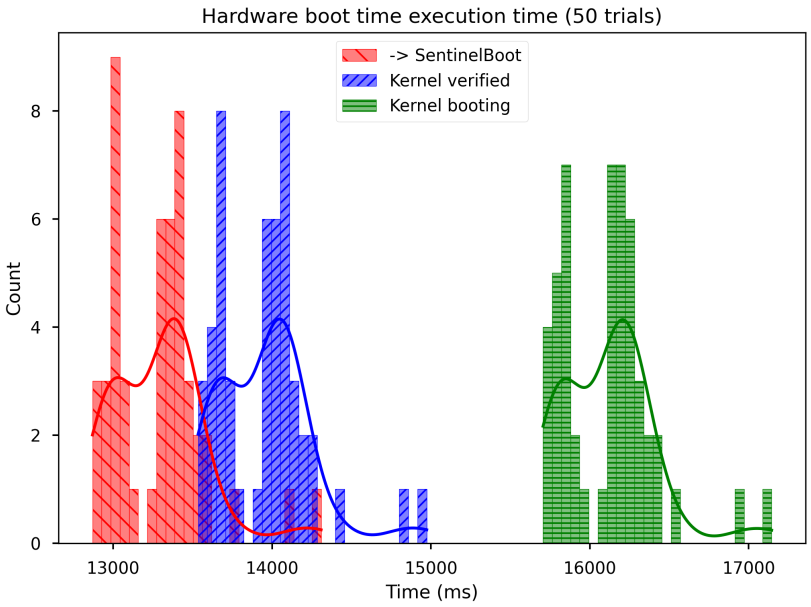

Figure 4.2: Hardware SentinelBoot execution time stages

Stage	**Mean (ms)**	**Standard deviation (ms)**
Reaching SentinelBoot	13289	293.94
SentinelBoot verification complete	13954	294.30
Kernel serial output	16114	294.44

Table 4.2: Hardware SentinelBoot execution time stages mean and standard deviation

Figure 4.2 and Table 4.2 illustrate, on hardware, the execution of SentinelBoot's control flow per stage, from being handed execution by U-Boot, to verifying the kernel, and finally the kernel booting. The additional operations offered by SentinelBoot to cryptographically verify the kernel only takes, on average, approximately 550 ms whereas receiving serial output from the kernel takes four times as long, at an average of approximately 2150 ms. The higher execution time of the kernel verification stage is significant and lower than expected when looking at Figure 4.1, as the overall overhead was 20.1%, with the kernel serial output taking approximately three times as long to execute than the verification. If the kernel performs any realignment or other significant operations prior to initialising the serial driver the significant delay is probable. It can therefore be viewed that the most significant component that determines SentinelBoot's execution time is the kernel's own setup.

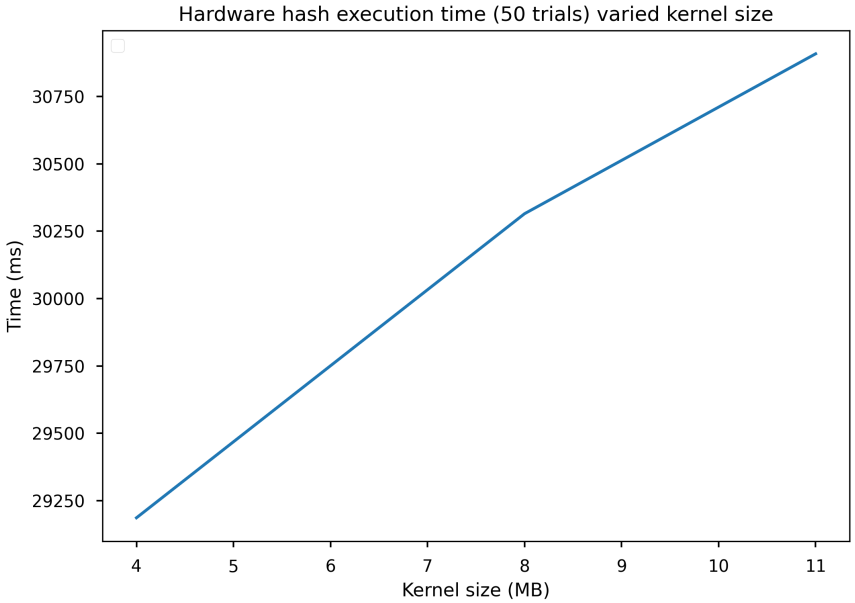

Figure 4.3: Hardware SentinelBoot execution time with varying kernel size

Kernel Size (MB)	**Mean (ms)**	**Standard deviation (ms)**
11	30908	0.12796
8	30315	3.1584
4	29187	0.19705

Table 4.3: Hardware SentinelBoot execution time with varying kernel size mean and standard deviation

Figure 4.3 and Table 4.3 illustrate, on hardware, SentinelBoot's execution time in regard to the size of the kernel binary to be hashed. SHA256 is an $O(n)$ operation and as such it is expected SentinelBoot's execution time would scale equivalently. This relationship can be seen with a seemingly linearly proportional increase in time for an increase in kernel size. However, it is important to note varying the kernel size involves changing configurations to compile in unnecessary modules to bloat the binary. Due to the limited way to vary the kernel size, only three different kernel sizes were obtained and therefore, while the relationship is shown, the strength of the claim is preliminary due to the number of data points.

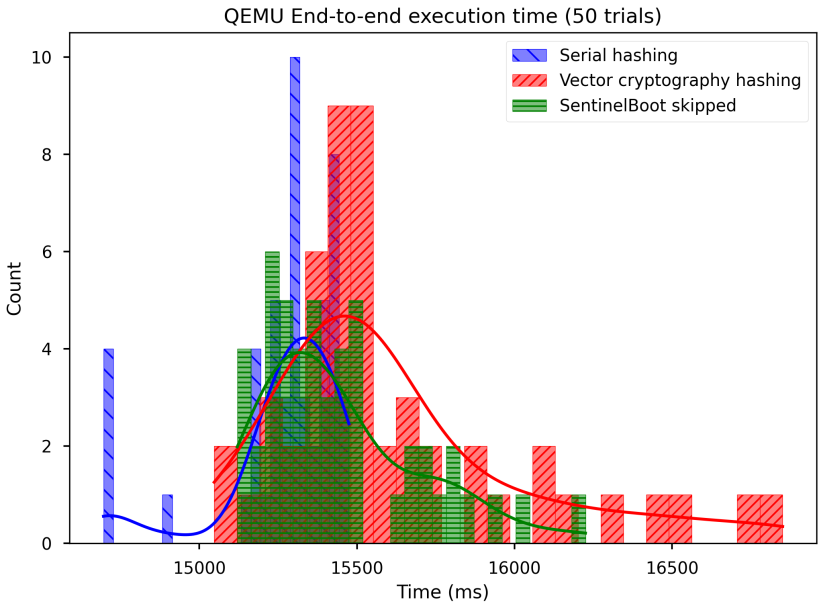

Figure 4.4: QEMU boot time with and without SentinelBoot including vector cryptography acceleration

Target	**Mean (ms)**	**Standard deviation (ms)**
SentinelBoot serial	15271	196.61
SentinelBoot vector cryptography	15621	403.77
U-Boot	15446	252.45

Table 4.4: QEMU boot time with and without SentinelBoot including vector cryptography acceleration mean and standard deviation

Figure 4.4 and Table 4.4 show, in QEMU, a standard U-Boot binary execution time of approximately 15400 ms, a SentinelBoot execution time of approximately 15300 ms without vector cryptography, and a SentinelBoot execution time of approximately 15600 ms with vector cryptography.

It should be noted that QEMU is not cycle accurate and, while performant, does not represent hardware performance such as bottlenecks and accelerations well, however, it is important to consider SentinelBoot's execution time under emulation for completeness. Nevertheless, SentinelBoot is able to execute in an equivalent time to U-Boot under emulation, and therefore within the success criterion.

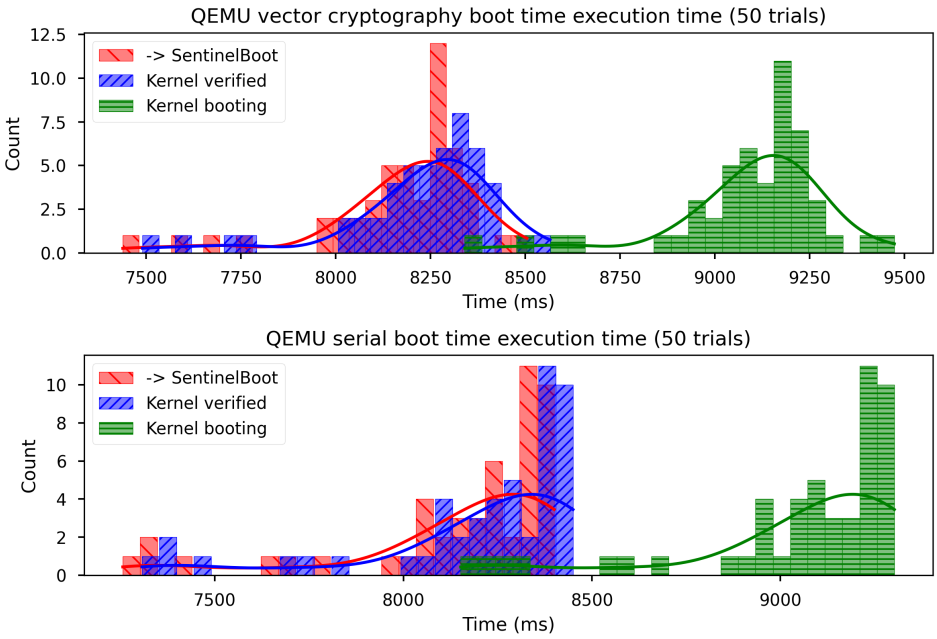

Figure 4.5: QEMU SentinelBoot execution time stages vector cryptography and serial

Stage	**Mean (ms)**	**Standard deviation (ms)**
Reaching SentinelBoot	8172.4	205.77
SentinelBoot verification complete	8225.2	206.02
Kernel serial output	9089.5	210.13

Table 4.5: QEMU SentinelBoot execution time stages vector cryptography mean and standard deviation

Stage	**Mean (ms)**	**Standard deviation (ms)**
Reaching SentinelBoot	8146.0	298.93
SentinelBoot verification complete	8195.9	298.91
Kernel serial output	9046.4	300.38

Table 4.6: QEMU SentinelBoot execution time stages serial mean and standard deviation

Figure 4.5, Table 4.5, and Table 4.6 illustrate, in QEMU, the execution time per stage of SentinelBoot's control flow. SentinelBoot cryptographically verifies the kernel in approximately 50 ms, whereas receiving serial output from the kernel takes seventeen times as long (at approximately 850 ms). Analogous to hardware, it is apparent that the kernel's own setup is the largest proportion of SentinelBoot's execution time.

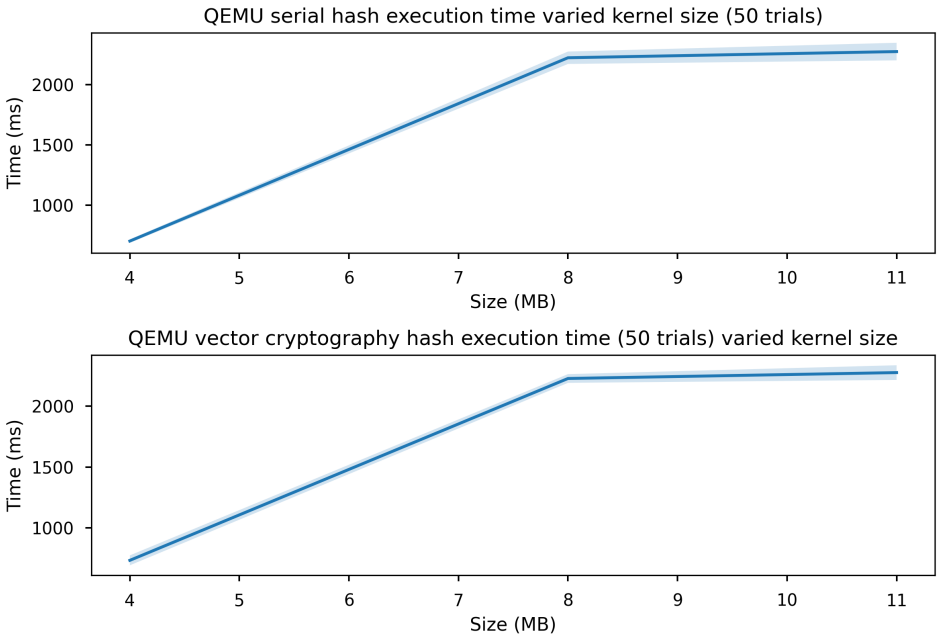

Figure 4.6: QEMU SentinelBoot execution time with varying kernel size using vector cryptography and serial

Kernel Size (MB)	**Mean (ms)**	**Standard deviation (ms)**
11	2274.4	60.181
8	2226.1	35.687
4	734.54	40.586

Table 4.7: QEMU SentinelBoot execution time with varying kernel size using vector cryptography mean and standard deviation

Kernel Size (MB)	**Mean (ms)**	**Standard deviation (ms)**
11	2274.5	72.697
8	2223.3	51.273
4	703.86	14.522

Table 4.8: QEMU SentinelBoot execution time with varying kernel size using serial mean and standard deviation

Figure 4.6, Table 4.7, and Table 4.8 illustrate SentinelBoot's execution time in regard to the size of the kernel binary to be hashed for both vector cryptography and serial in QEMU. The linear relationship expected is not seen with 4 MB, taking significantly less than 8 MB and 11 MB, which have no significant difference as they're within the confidence interval of each other's standard deviations. The cause of this behaviour is unknown but may relate to the hosts CPU cache sizes, or simply is the result of an artefact of the overhead caused by QEMU running within a virtual machine. Analogous to hardware, with only three data points the strength of the claim is weak, and with more data points the $O(n)$ relationship may have emerged.

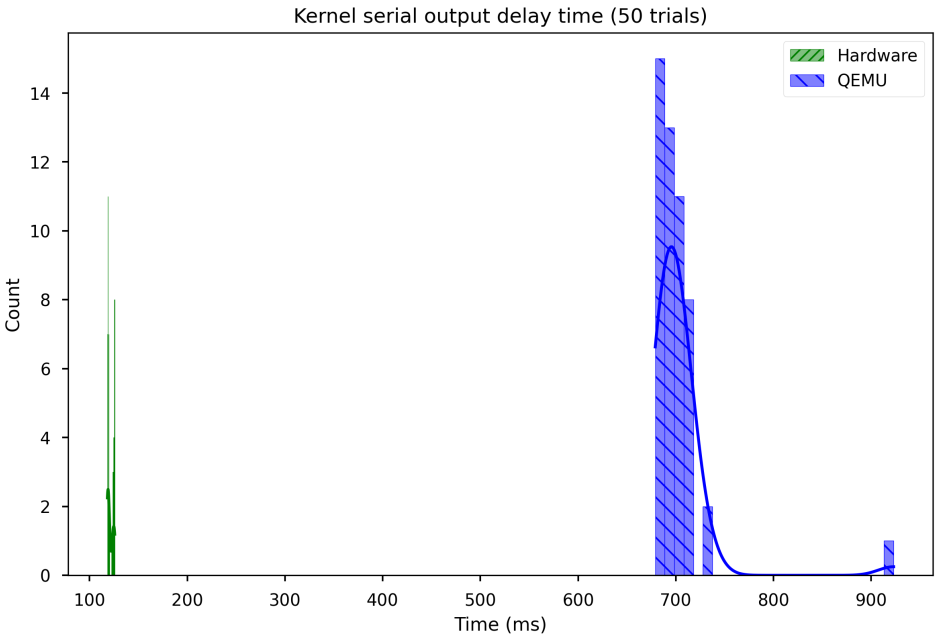

Figure 4.7: Kernel serial output delay on hardware and under emulation

Platform	Mean (ms)	Standard deviation (ms)
Hardware	121.21	2.8937
Emulation	701.94	34.263

Table 4.9: Kernel serial output delay on hardware and under emulation mean and standard deviation

Figure 4.7 and Table 4.9 demonstrate the kernel serial output delay for both hardware and QEMU. It is visible that QEMU encounters a significant delay, and while hardware does too, it is less significant. The primary component of the delay is still between the final output of U-Boot and the first output of the kernel.

It is reasonable to conclude that SentinelBoot has not hindered the kernel's initial boot process under emulation, however, executing on hardware, there is approximately 2000 ms of time unaccounted for. Adding `earlyprintk` or similar to the kernel command line arguments does not provide insight into the delay as the kernel does not report any additional operations prior to the first serial output. This, coupled with the fact it is not possible to obtain an alternative kernel binary means that it is difficult to theorise about the cause of the delay.

Events such as hardware UART buffers being full from SentinelBoot's execution would not cause such a large delay and, given only the jump from SentinelBoot to the kernel takes place after SentinelBoot's last print, the kernel itself is responsible for the delay, and given that it does not report any operations coupled with the VisionFive 2 not having a JTAG connector and not having access to a HiFive Unmatched board at evaluation time, it is not possible to profile the execution to determine the delay.

Overall, it is evident that SentinelBoot is able to meet the success criterion of booting Linux in less than 3 seconds, or approximately double a standard U-Boot binary execution time. Additionally, it is shown that SentinelBoot's execution time scales linearly with the size of the kernel binary to verify when executing on hardware.

4.2 Binary Size

SentinelBoot aimed to achieve its functionality in a compiled binary of less than 700 kB in size, or the approximate size of the U-Boot binary. The necessity for the resulting binary to be of an equivalent size is due to the constrained nature of the environment bootloaders operate; constraints include limited memory storage size, low memory bandwidth, or hardware cycle constrains.

Target	Binary Size (kB)
QEMU	73.824
QEMU Vector Cryptography	61.504
Visionfive 2	73.824
HiFive Unmatched	73.824
(U-Boot example)	(742.70)

Table 4.10: SentinelBoot target binary sizes

Table 4.10 presents the binary sizes for the four targets of SentinelBoot. Each one is significantly smaller than the U-Boot binary by around one order of magnitude, and therefore successfully meets the success criterion.

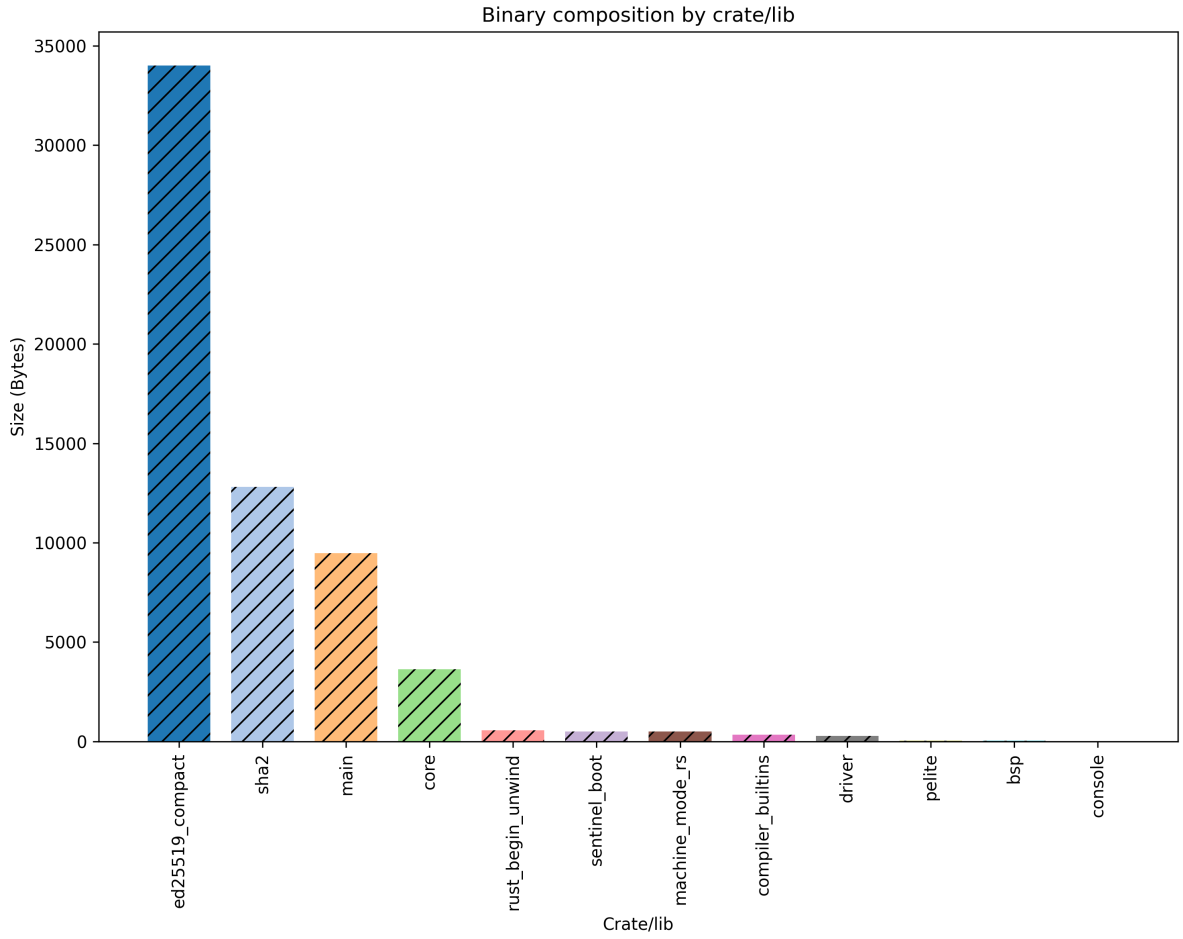

Figure 4.8: Cargo bloat binary composition

Target	Binary Size (kB)
ed25519_compact	34004
sha2	12804
main	9466
core	3640
rust_begin_unwind	560
sentinel_boot	510
machine_mode_rs	510
compiler_builtins	336
driver	286
pelite	40
bsp	36
console	18

Table 4.11: Cargo bloat binary composition

Figure 4.8 and Table 4.11 illustrate the binary composition as interpreted by `cargo bloat`. `ed25519_compact` constitutes the majority of the binary size: it is used only to verify the digital signature, as such it is very probable an equivalent crate could be employed which would compile to a smaller size, but given the ease of use of the API and the significantly smaller size of SentinelBoot, it is not a significant focus of future work.

Overall, it is evident that SentinelBoot is able to meet the success criterion of generating a binary less than 700 kB, or approximately a standard U-Boot binary size, however, it is evident that more careful consideration of crates may further optimise SentinelBoot's composition and therefore binary size.

4.3 Compile Time

SentinelBoot aimed to compile in less time than the U-Boot binary, approximately 11 seconds.

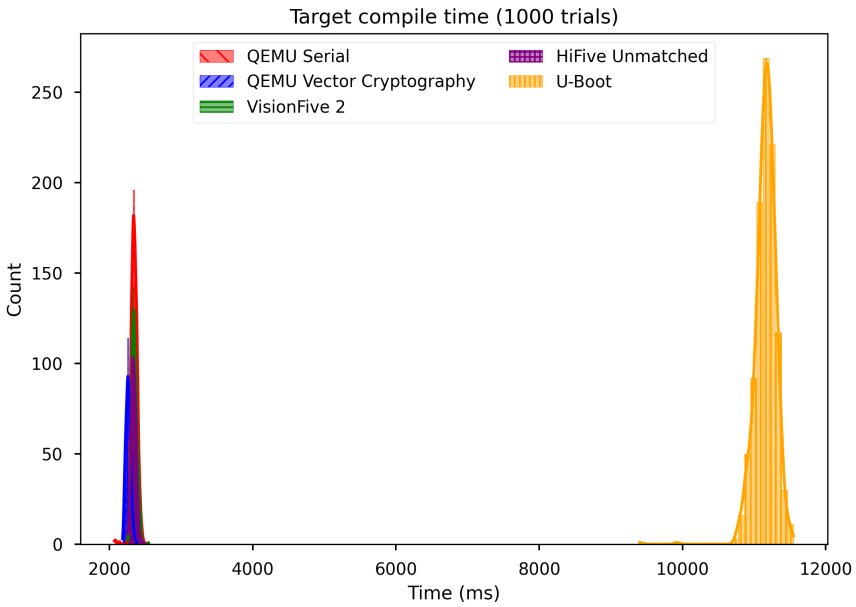

Figure 4.9: SentinelBoot target compile time compared with U-Boot

Target	Mean (ms)	Standard deviation (ms)
QEMU	23478	43.469
QEMU Vector Cryptography	22620	33.749
Visionfive 2	23475	37.015
HiFive Unmatched	23388	33.983
(U-Boot example)	11161	149.89

Table 4.12: SentinelBoot target compile times

Figure 4.9 and Table 4.12 illustrate the compile times for each of SentinelBoot's targets compared to U-Boot. It is visible that U-Boot, which takes advantage of parallelism (`make -j8`), is almost five times slower than SentinelBoot's slowest target. Therefore, it is possible to conclude SentinelBoot achieves this success criterion.

It is important to discuss Rust's compile time, as it does not natively support fully parallelised builds. The build time of large scale projects will be an important factor in Rust's adoption as slow builds waste developer time, ultimately raising software costs.

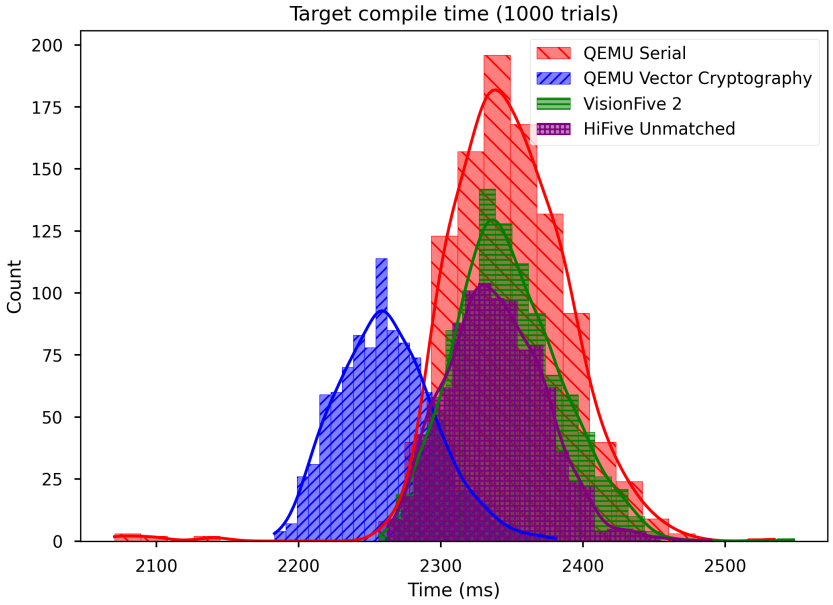

Figure 4.10: SentinelBoot target compile times

Figure 4.10 and Table 4.12 show almost no difference between build times apart from vector cryptography, which is significantly faster. The faster build in this case is due to not using the `sha-2` crate, and instead implementing the functionality in assembly. The code to achieve this operation is significantly simpler, at approximately 160 lines of code compared to the approximately 1800 in the `sha-2` crate[2], therefore given the fewer lines of code, the vector cryptography target compiles faster.

[2]Calculated by `git ls-files | xargs wc -l`

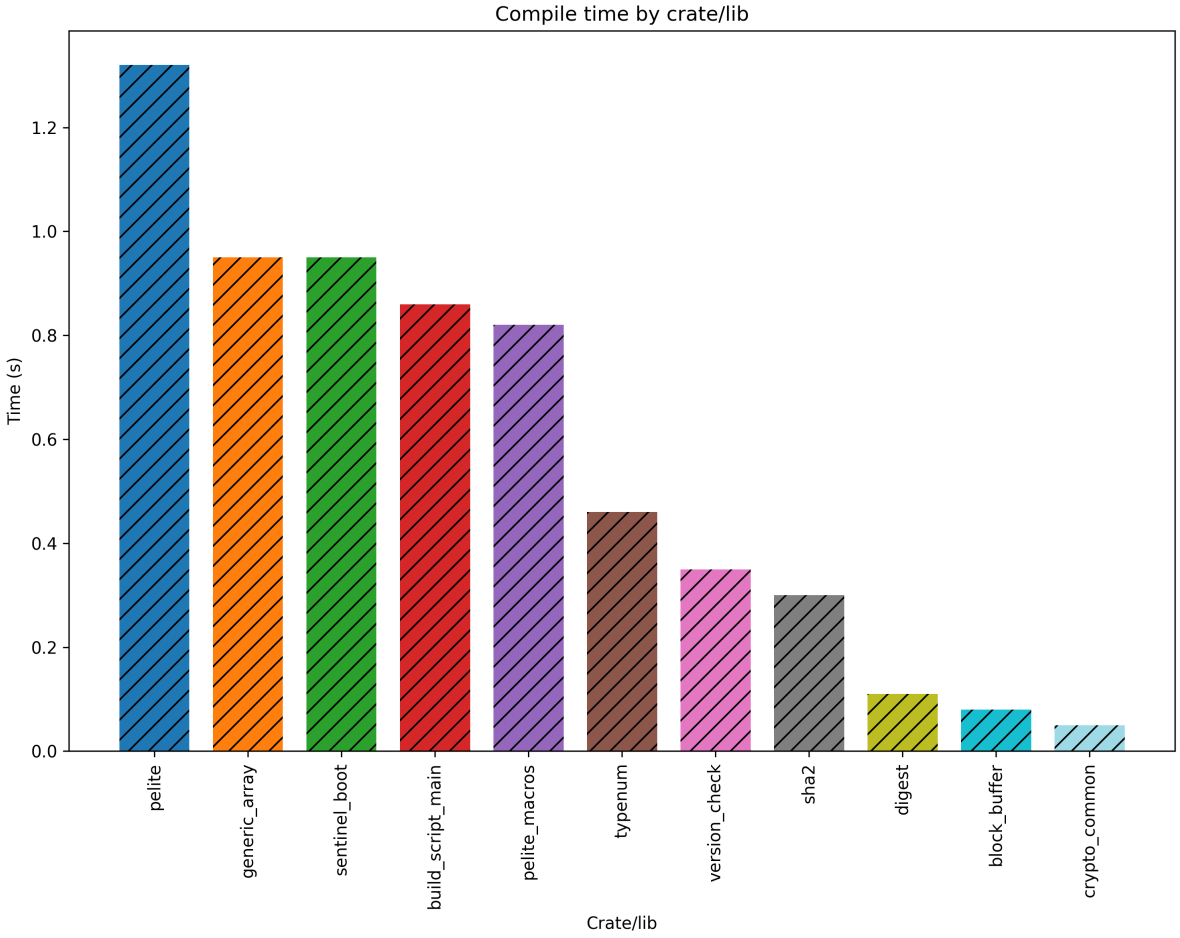

Figure 4.11: SentinelBoot target compile times cargo bloat output

Target	**Binary Size (kB)**
`pelite`	1.32
`generic_array`	0.95
`sentinel_boot`	0.95
`build_script_main`	0.86
`pelite_macros`	0.82
`typenum`	0.46
`version_check`	0.35
`sha2`	0.3
`digest`	0.11
`block_buffer`	0.08
`crypto_common`	0.05

Table 4.13: SentinelBoot target compile times cargo bloat output

Figure 4.11 and Table 4.13 illustrate the compile time data provided by cargo bloat. It is visible that the `pelite` crate is consuming a large amount of compile time, with `pelite` and `pelite_macros` consuming a total of 2.18 seconds. Additionally, the `sha2` crate, while itself not requiring a large amount of compile time depends on the `generic_array` crate, which consumes 0.95 seconds of compile time.

`pelite` makes poor use of feature flags: where SentinelBoot only requires `pelite::pe64` to parse the 64 bit ELF header, it is also required to build the 32 bit parser, all non ELF parsing structs, and support for digital signatures. `sha2` utilises the `generic_array` crate which is not necessary for SHA256, which

itself does not require dynamic memory allocation to be performed. Both crates illustrate the need for caution when choosing crates to utilise, due to their dependencies and potentially unnecessary features.

Overall, it is evident that SentinelBoot is able to meet the success criterion of compiling in less than 11 seconds, or approximately a standard U-Boot binary's compile time. However, it is evident that more careful consideration of crates used may further optimise SentinelBoot's compile time.

4.4 Memory Safety and Security

4.4.1 Memory Safety

SentinelBoot aimed to, as much as possible given development time and skill set, produce the required functionality with a minimal number of unsafe lines of code in order to demonstrate the advancements offered by the Rust programming language.

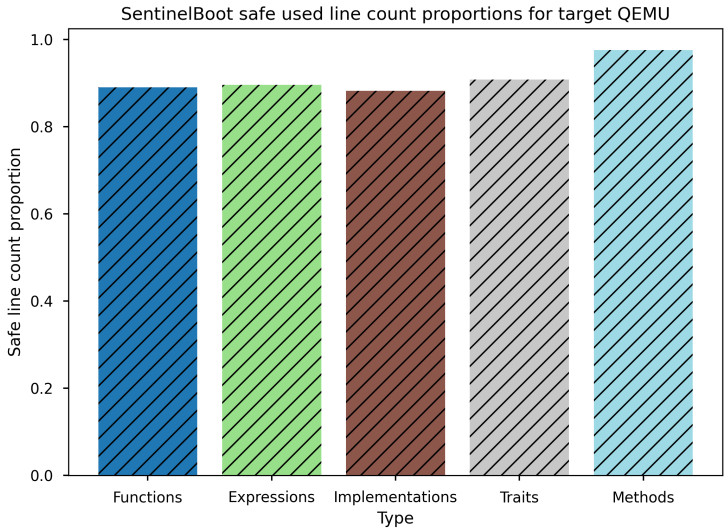

Figure 4.12: SentinelBoot used line safety proportions

Type	Safe Used Line Proportion
Functions	0.89063
Expressions	0.89598
Implementations	0.88266
Traits	0.90833
Methods	0.97632

Table 4.14: SentinelBoot used line safety proportions

Figure 4.12 and Table 4.14 illustrate the used safe line count data provided by `cargo geiger` when compiled for the QEMU target. This data only includes lines of code which are used in the final binary, and therefore does not include all lines from other targets.

Given the nature of a bootloader, it is unavoidable that certain parts of the control flow will require direct memory access, such as memory mapped serial devices. Despite this necessity, SentinelBoot achieves greater than 80% proportion of used lines being safe, with traits achieving approximately 90%, and methods approximately 98%.

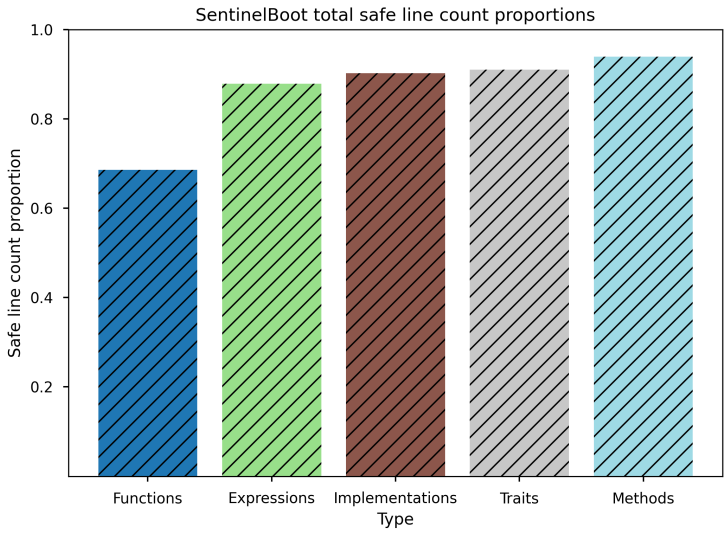

Figure 4.13: SentinelBoot total line safety proportions

Type	**Safe Used Line Proportion**
Functions	0.68571
Expressions	0.87908
Implementations	0.90254
Traits	0.91057
Methods	0.93974

Table 4.15: SentinelBoot total line safety proportions

Figure 4.13 and Table 4.15 illustrate the total safe line count data provided by `cargo geiger`. This data includes lines of code for all targets.

SentinelBoot achieves approximately 68% safe function proportion, 87% safe expression proportion, and greater than 90% for implementations, traits, and methods. The lower memory safety proportions are due to the vector cryptography target replacing the memory safe `sha-2` crate with an unsafe assembly implementation. As the vector cryptography extension utilises assembly instructions, it is not possible to execute them without the `unsafe` keyword, thus the lower safety proportions.

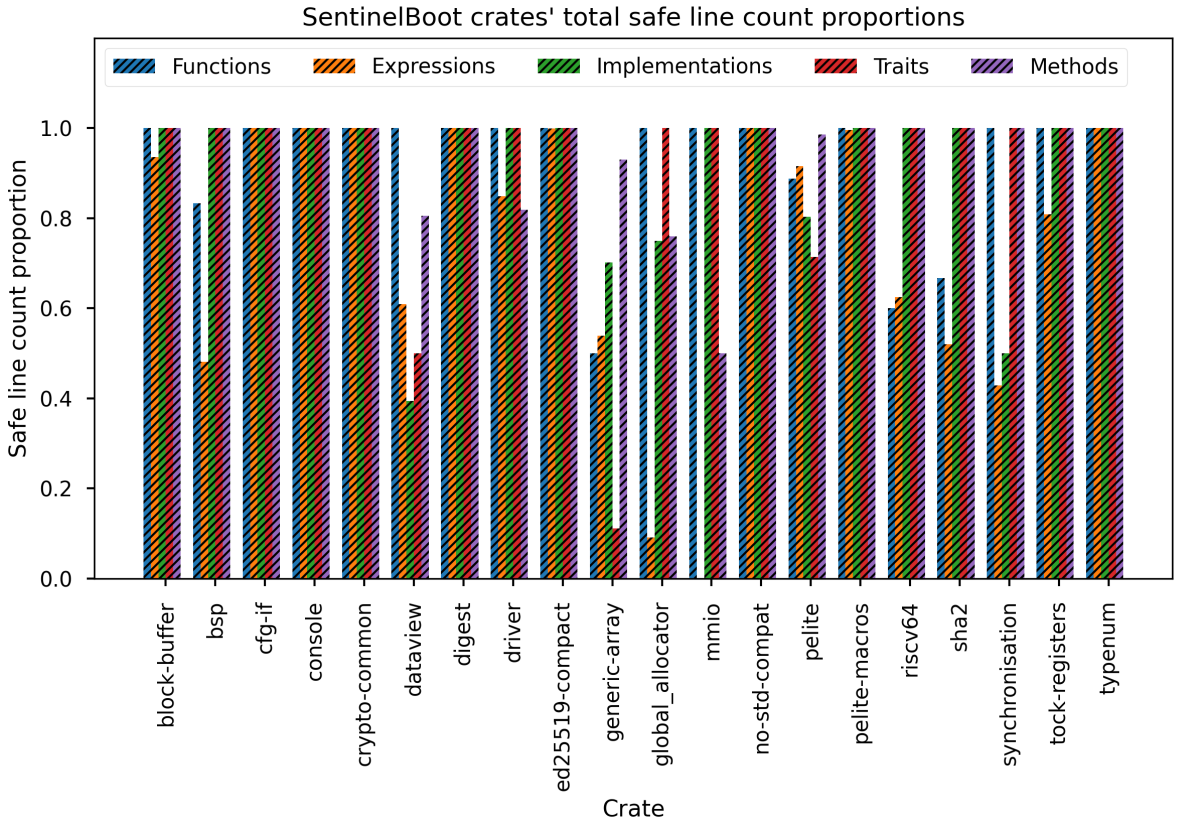

Figure 4.14: SentinelBoot crates' total line safety proportions

Package	Functions	Expressions	Implementations	Traits	Methods
block-buffer	1.0	0.935	1.0	1.0	1.0
bsp	0.833	0.481	1.0	1.0	1.0
cfg-if	1.0	1.0	1.0	1.0	1.0
console	1.0	1.0	1.0	1.0	1.0
crypto-common	1.0	1.0	1.0	1.0	1.0
dataview	1.0	0.609	0.395	0.5	0.806
digest	1.0	1.0	1.0	1.0	1.0
driver	1.0	0.849	1.0	1.0	0.818
ed25519-compact	1.0	0.999	1.0	1.0	1.0
generic-array	0.5	0.539	0.701	0.111	0.93
global_allocator	1.0	0.091	0.75	1.0	0.76
mmio	1.0	1.0	1.0	1.0	0.5
no-std-compat	1.0	1.0	1.0	1.0	1.0
pelite	1.0	0.916	0.803	0.714	0.986
pelite-macros	0.887	0.995	1.0	1.0	1.0
riscv64	1.0	0.625	1.0	1.0	1.0
sha2	0.6	0.52	1.0	1.0	1.0
synchronisation	1.0	0.429	0.5	1.0	1.0
tock-registers	1.0	0.808	1.0	1.0	1.0
typenum	1.0	1.0	1.0	1.0	1.0

Table 4.16: SentinelBoot crates' total line safety proportions

Figure 4.14 and Table 4.16 illustrate the total safe line count data provided by `cargo geiger` for SentinelBoot's crates, including external. The distribution across crates on safe line proportions is evident, with crates such as `mmio` and `global_allocator` having low proportions due to the necessary need to

access raw memory, and other crates external to SentinelBoot such as `generic_array` also lower the safe proportions. As previously mentioned, `generic_array` should not be necessary for SHA256 operation and as such unnecessarily lowers SentinelBoot's safe line proportions. SentinelBoot would benefit from future work to remove external crates with its own implementations, both to minimise bloat and to control unsafe line usage.

4.4.2 Security

The security of SentinelBoot stems from two cryptographic techniques: public key cryptography and hashing. The use of SHA256 as a strong cryptographic hashing function provides great assurance to the integrity of the kernel binary. The use of a digital signature with a trusted certificate authority additionally provides great assurance to the authenticity of the kernel. The result of implementing both cryptographic techniques provides a remote root of trust to a trusted server, however, direct modification of the server's IP address via a social engineering attack or evil maid is a significant vulnerability. Future work to minimise the vulnerability could include hardcoding the server's public key in a ROM upon manufacturing, or other techniques to remove IP changing capabilities from the end device.

4.4.3 Summary

Overall it can be concluded SentinelBoot makes effort to minimise memory unsafety and, aside from the vector cryptography target, achieves a very high proportion of safe line counts. Additionally, SentinelBoot presents a strong security model as well as measures to strengthen the assurances.

4.5 Summary

Throughout this evaluation, it has been demonstrated SentinelBoot achieves all the success criteria in all aspects.

Chapter 5

Summary and Conclusions

This section concludes each of the success criteria, summarising to what extent the success criteria was met and future work that could further exceed it. Finally, a personal reflection is given on the implementation of SentinelBoot as a year-long thesis.

5.1 Boot Time

SentinelBoot is able to cryptographically verify and boot a standard Linux kernel with a 20.1% performance overhead compared to an example U-Boot binary which does not verify the kernel. SentinelBoot achieves this success criterion by being able to execute in less than double the execution time of a standard U-Boot binary. Therefore, it can be concluded SentinelBoot achieves excellent execution time when compared with the success criterion.

5.2 Binary Size

SentinelBoot is able to achieve all functionality in a resulting binary approximately one-tenth the size of an example U-Boot binary. This demonstrates Rust and SentinelBoot's ability to achieve functionality without bloat and, as such, be able to operate under the same or stricter conditions than an example U-Boot binary. It can be concluded that SentinelBoot achieves the binary size success criterion ten-fold.

5.3 Compile Time

All of SentinelBoot's targets are able to compile in less than a quarter of the time of an example U-Boot binary, where the U-Boot binary has been compiled with eight parallel jobs, whereas SentinelBoot was serially compiled. As such the success criterion has been achieved very well, especially given that U-Boot serial compilation may have been 32 times slower. This result demonstrates Rust's ability to be a viable alternative to C where the overhead of its static analysis checks at compile time do not waste developer time to a large extent, and as such does not further raise software development costs.

5.4 Memory Safety

SentinelBoot's QEMU target achieves full functionality with greater than 80% proportion of used lines being safe, with traits achieving approximately 90% safe line proportions, and methods approximately 98%. These values are slightly lower when all targets are considered, at approximately 68% safe function proportion, 87% safe expression proportion, and greater than 90% for implementations, traits, and methods. This is due to the vector cryptography acceleration requiring unsafe assembly instructions.

It is unfair to classify U-Boot, written in C, as 100% memory unsafe, but it is evident that the use of Rust has enhanced the memory safety, and that SentinelBoot makes good effort to utilise safe operations where possible, and as such the success criterion has been achieved.

5.5 Security

SentinelBoot's security model leverages public key cryptography and cryptographic hashing. These two methods are standardised and form the basis of modern cybersecurity, as being able to bypass such functions are viewed as computationally infeasible. The primary attack vectors that affects SentinelBoot are evil maid and social engineering which both involve modifying the IP address where SentinelBoot

bases its root of trust; future work to minimise these vectors are presented later. Overall, SentinelBoot's security model can be viewed as very strong and satisfying the success criterion.

5.6 Summary

The evaluation demonstrates that the success criteria have been met, with the numeric criteria falling within acceptable ranges. SentinelBoot successfully cryptographically verifies a standard Linux kernel while maintaining a minimal number of unsafe lines of code, showcasing Rust's potential to replace C in system software and reduce the likelihood of memory safety vulnerabilities.

5.7 Future Work

From the evaluation, there is the potential for future work to improve upon SentinelBoot's implementation. Possible endeavours include:

- Place a more careful focus on crate usage to minimise pulled in dependencies and unnecessary features.
- Where possible, aim to develop equivalent implementations of used crates whereby only used features are implemented and control over unsafe usage is available.
- Where viable, improve SentinelBoot's security model by the use of onboard RAM or memory protection for the public key, server IP, and boot flag.

5.8 Reflection

The primary goal of the SentinelBoot project was to learn. The environment it operates in are rarely taught, discussed in-person or online, or presented in media. SentinelBoot presented an excellent opportunity to learn and implement new topics while simultaneously attempting to write the implementations as memory safe as possible in the Rust programming language. The journey of writing SentinelBoot taught me the role of linking, the operation of serial drivers and memory allocators, Rust, debugging skills with GDB, JTAG, and OpenOCD, and finally project management skills due to the length and scale of the project.

Secondly, SentinelBoot aimed to demonstrate potential weak points in Rust as a system software programming language compared to C. Rust proved challenging at times due to its additional constraints on source code, such as the doubly linked list problem, lifetimes, error handling, and the change in methodology from fail-fast I've typically written in C. Despite this, Rust was enjoyable to program in, and throughout implementation I noticed points where Rust really helped in guarding against a memory safety mistake I had not noticed, and without the constraints from the compiler I could not have written an equal memory safe project in C.

Finally, SentinelBoot was not always optimal; development would have benefited from learning more before jumping in. Writing a C implementation for an ARM board would have covered the basics in a simpler form and allowed building upon them. Switching earlier from the VisionFive 2 board to the HiFive Unmatched and learning how to use the JTAG would have saved considerable time and suffering during the development process. Additionally, I would have, if able to, streamlined development more and, with the additional time, implemented the TFTP Ethernet driver to provide SentinelBoot full functionality.

In summary, SentinelBoot was very successful: it taught me a huge amount, demonstrated what the project set out to do, and while the improvements and future work outlined earlier could serve to benefit SentinelBoot, these considerations were only made with the hindsight and knowledge I obtained from this project itself.

Bibliography

[1] J.E. Cooling. "Languages for the programming of real-time embedded systems a survey and comparison". In: *Microprocessors and Microsystems* 20.2 (1996), pp. 67–77. ISSN: 0141-9331. DOI: https://doi.org/10.1016/0141-9331(95)01067-X. URL: https://www.sciencedirect.com/science/article/pii/014193319501067X.

[2] U-Boot. *U-Boot/u-boot: "Das U-Boot" source tree*. URL: https://github.com/u-boot/u-boot (visited on 10/04/2024).

[3] White House. *Back To The Building Blocks: A Path Toward Secure and Measurable Software*. 2024. URL: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[4] William Bugden and Ayman Alahmar. *Rust: The Programming Language for Safety and Performance*. 2022. arXiv: 2206.05503 [cs.PL].

[5] Amélie Gonzalez, Djob Mvondo and Yérom-David Bromberg. "Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel". In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. PLOS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 18–25. ISBN: 9798400704048. DOI: 10.1145/3623759.3624547. URL: https://doi.org/10.1145/3623759.3624547.

[6] Stefan Lankes et al. "RustyHermit: A Scalable, Rust-Based Virtual Execution Environment". In: *High Performance Computing*. Ed. by Heike Jagode et al. Cham: Springer International Publishing, 2020, pp. 331–342. ISBN: 978-3-030-59851-8.

[7] Claudio Zeeb, Francois Bry and Thomas Prokosch. *MEMORY MANAGEMENT IN RUST*.

[8] Msrc. *Microsoft*. URL: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/ (visited on 10/04/2024).

[9] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488. ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. URL: https://doi.org/10.1145/2663716.2663755.

[10] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584915001366.

[11] Peter Kafka. "The Automotive Standard ISO 26262, the Innovative Driver for Enhanced Safety Assessment & Technology for Motor Cars". In: *Procedia Engineering* 45 (Dec. 2012), pp. 2–10. DOI: 10.1016/j.proeng.2012.08.112.

[12] John Hatcliff et al. "Certifiably safe software-dependent systems: challenges and directions". In: *Future of Software Engineering Proceedings*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 182–200. ISBN: 9781450328654. DOI: 10.1145/2593882.2593895. URL: https://doi.org/10.1145/2593882.2593895.

[13] Petra Heck, Martijn Klabbers and Marko van Eekelen. "A software product certification model". In: *Software Quality Journal* 18.1 (Mar. 2010), pp. 37–55. ISSN: 1573-1367. DOI: 10.1007/s11219-009-9080-0. URL: https://doi.org/10.1007/s11219-009-9080-0.

[14] Roland Kammerer. "Linux in safety-critical applications". PhD thesis. 2008.

[15] Karim Eldefrawy et al. "Smart: secure and minimal architecture for (establishing dynamic) root of trust." In: *Ndss*. Vol. 12. 2012, pp. 1–15.

[16] *The U-Boot Documentation - Das U-Boot documentation*. URL: https://docs.u-boot.org/ (visited on 10/04/2024).

[17] Akshay Bhat. "Secure boot, chain of trust and data protection". In: *Embedded World Conference*. 2019.

[18] Hui Xu et al. "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs". In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (Sept. 2021). ISSN: 1049-331X. DOI: 10.1145/3466642. URL: https://doi.org/10.1145/3466642.

[19] Oscar Llorente-Vazquez et al. "Detection, exploitation and mitigation of memory errors". In: *Logic Journal of the IGPL* 32.2 (Mar. 2024), pp. 281–292. ISSN: 1367-0751. DOI: 10.1093/jigpal/jzae008. eprint: https://academic.oup.com/jigpal/article-pdf/32/2/281/57088662/jzae008.pdf. URL: https://doi.org/10.1093/jigpal/jzae008.

[20] Jinyu Gu et al. "EPK: Scalable and Efficient Memory Protection Keys". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 609–624. ISBN: 978-1-939133-29-36. URL: https://www.usenix.org/conference/atc22/presentation/gu-jinyu.

[21] Raymond Chen. *An amusing story about a practical use of the null garbage collector*. Mar. 2019. URL: https://devblogs.microsoft.com/oldnewthing/20180228-00/?p=98125 (visited on 23/07/2023).

[22] Speykious. *CVE-Rs: Blazingly fast memory vulnerabilities, written in 100% safe rust.* URL: https://github.com/Speykious/cve-rs (visited on 27/02/2024).

[23] Patently Apple. *A Major Tectonic Shift away from Arm to RISC-V may be in the works for Qualcomm, Samsung, Google, Nvidia and Apple.* URL: https://www.patentlyapple.com/2023/06/a-major-tectonic-shift-away-from-arm-to-risc-v-may-be-in-the-works-for-qualcomm-samsung-google-nvidia-and-apple.html (visited on 03/08/2023).

[24] Bruce Dubbs. *GNU grub.* URL: https://www.gnu.org/software/grub/index.html (visited on 26/05/2023).

[25] Ronald G Minnich, James Hendricks and Dale Webster. "The Linux {BIOS}". In: *4th Annual Linux Showcase & Conference (ALS 2000)*. 2000.

[26] Coreboot. *Coreboot/coreboot: Mirror of https://review.coreboot.org/coreboot.git.* URL: https://github.com/coreboot/coreboot (visited on 26/05/2023).

[27] Egormkn. *Egormkn/MBR-Boot-manager: Master Boot record with a boot menu written in Assembly.* URL: https://github.com/egormkn/mbr-boot-manager (visited on 26/05/2023).

[28] Olafhering. *Olafhering/grub: Git://git.savannah.gnu.org/grub.git.* URL: https://github.com/olafhering/grub (visited on 26/05/2023).

[29] Director Himanshu Kathpal. *CVE-2021-3156: Heap-based buffer overflow in Sudo (baron samedit).* Dec. 2022. URL: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit (visited on 26/05/2023).

[30] United States Government. *NSA ANT product catalog.* NSA. (Visited on 26/05/2023).

[31] Ievgeniia Kuzminykh and Maryna Yevdokymenko. "Analysis of Security of Rootkit Detection Methods". In: *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. 2019, pp. 196–199. DOI: 10.1109/ATIT49449.2019.9030428.

[32] Masoudeh Keshavarzi and Hamid Reza Ghaffary. "I2CE3: A dedicated and separated attack chain for ransomware offenses as the most infamous cyber extortion". In: *Computer Science Review* 36 (2020), p. 100233. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2020.100233. URL: https://www.sciencedirect.com/science/article/pii/S1574013719300838.

[33] Jiewen Yao and Vincent Zimmer. "Firmware Secure Coding Practice". In: *Building Secure Firmware: Armoring the Foundation of the Platform*. Berkeley, CA: Apress, 2020, pp. 495–569. ISBN: 978-1-4842-6106-4. DOI: 10.1007/978-1-4842-6106-4_14. URL: https://doi.org/10.1007/978-1-4842-6106-4_14.

[34] Trusted Firmware. *TrustedFirmware-A (TF-A).* URL: https://www.trustedfirmware.org/projects/tf-a/ (visited on 10/04/2024).

[35] Rust-Osdev. *Rust-OSDEV/bootloader: An experimental pure-rust x86 bootloader.* URL: https://github.com/rust-osdev/bootloader (visited on 10/04/2024).

[36] Wenjun Xiong and Robert Lagerström. "Threat modeling - A systematic literature review". In: *Computers & Security* 84 (2019), pp. 53–69. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2019.03.010. URL: https://www.sciencedirect.com/science/article/pii/S0167404818307478.

[37] Alexander Tereshkin. "Evil maid goes after PGP whole disk encryption". In: *Proceedings of the 3rd International Conference on Security of Information and Networks*. SIN '10. Taganrog, Rostov-on-Don, Russian Federation: Association for Computing Machinery, 2010, p. 2. ISBN: 9781450302340. DOI: 10.1145/1854099.1854103. URL: https://doi.org/10.1145/1854099.1854103.

[38] Fatima Salahdine and Naima Kaabouch. "Social Engineering Attacks: A Survey". In: *Future Internet* 11.4 (2019). ISSN: 1999-5903. DOI: 10.3390/fi11040089. URL: https://www.mdpi.com/1999-5903/11/4/89.

[39] Avijit Mallik. "Man-in-the-middle-attack: Understanding in simple words". In: *Cyberspace: Jurnal Pendidikan Teknologi Informasi* 2.2 (Jan. 2019), p. 109. DOI: 10.22373/cj.v2i2.3453.

[40] Axel Simon and Lily Sturmann. *Current trusted execution environment landscape*. Sept. 2021. URL: https://next.redhat.com/2019/12/02/current-trusted-execution-environment-landscape/.

[41] Pascal Nasahl et al. "HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 187–199. ISBN: 9781450382878. DOI: 10.1145/3433210.3453112. URL: https://doi.org/10.1145/3433210.3453112.

[42] Rust-Embedded. *Rust-embedded/rust-raspberrypi-os-tutorials: Learn to write an embedded OS in rust*. URL: https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials/tree/master (visited on 10/04/2024).

[43] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006.

[44] QEMU. *QEMU Documentation*. URL: https://www.qemu.org/documentation/ (visited on 10/04/2024).

[45] Lawrence Hunter, Kiran Ostrolenk, Nazar Kazakov, and Dickon Hood. *Adding RISC-V vector cryptography extension support to QEMU*. June 2023. URL: https://www.codethink.co.uk/articles/2023/vcrypto_qemu/ (visited on 25/07/2023).

[46] Lawrence Hunter and Kiran Ostrolenk. June 2023. URL: https://riscv-europe.org/media/proceedings/posters/2023-06-08-Lawrence-HUNTER-poster.pdf (visited on 25/07/2023).

[47] Raspberry Pi Foundation. *Teach, learn, and make with the Raspberry Pi Foundation*. URL: https://www.raspberrypi.org/ (visited on 10/04/2024).

[48] Docker. *Docker Documentation*. Mar. 2024. URL: https://docs.docker.com/ (visited on 10/04/2024).

[49] GNU. *Using LD, the GNU linker - Command Language*. URL: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html (visited on 10/04/2024).

[50] Rust Embedded. *Introduction - The Embedded Rust Book*. URL: https://docs.rust-embedded.org/book/ (visited on 10/04/2024).

[51] Tao Lu. "A Survey on RISC-V Security: Hardware and Architecture". In: *CoRR* abs/2107.04175 (2021). arXiv: 2107.04175. URL: https://arxiv.org/abs/2107.04175.

[52] RISC-V. *RISC-V Priveleged ISA*. URL: https://riscv.org/technical/specifications/privileged-isa/ (visited on 10/04/2024).

[53] Jacob Gorban and Luke E. McKay. *UART IP Core Specification - UART16550 Core Technical Manual documentation - Revision 0.6.1*. URL: https://uart16550.readthedocs.io/en/latest/uart16550doc.html (visited on 10/04/2024).

[54] University of Washington. *Lecture 11: Dynamic Memory Allocation continued*. URL: https://courses.cs.washington.edu/courses/cse374/20au/lectures/11-dyna-mem-alloc-continued/11-dyna-mem-alloc-continued.pdf.

[55] Sourceware. *GDB: The GNU project debugger*. URL: https://sourceware.org/gdb/ (visited on 10/04/2024).

[56] IEEE. *IEEE Standards Association*. URL: https://standards.ieee.org/ieee/1149.1/4484/ (visited on 10/04/2024).

[57] Fercerpav and Ntfreak. *Open on-chip debugger*. URL: https://openocd.org/ (visited on 10/04/2024).

[58] Torvalds. *Linux/documentation/arch/RISCV/boot.rst at master · torvalds/linux*. URL: https://github.com/torvalds/linux/blob/master/Documentation/arch/riscv/boot.rst (visited on 10/04/2024).

[59] Kernel. *Linux and the Devicetree - The Linux Kernel documentation.* URL: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html (visited on 10/04/2024).

[60] Rajeev Sobti and Ganesan Geetha. "Cryptographic hash functions: a review". In: *International Journal of Computer Science Issues (IJCSI)* 9.2 (2012), p. 461.

[61] National Institute of Standards and Technology. *FIPS 180-4.* URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[62] Philip N. Klein. "Public-Key Cryptosystems and Digital Signatures". In: *A Cryptography Primer: Secrets and Promises.* Cambridge University Press, 2014, pp. 157–170.

[63] Ev Kontsevoy. *Comparing SSH keys - RSA, DSA, ECDSA, or EDDSA?* URL: https://goteleport.com/blog/comparing-ssh-keys/ (visited on 11/04/2024).

[64] RISC-V. *RISCV/RISCV-crypto: RISC-V cryptography extensions standardisation work.* URL: https://github.com/riscv/riscv-crypto (visited on 10/04/2024).

[65] Michael J Flynn and Kevin W Rudd. "Parallel architectures". In: *ACM computing surveys (CSUR)* 28.1 (1996), pp. 67–70.

[66] NSA. *Ghidra.* URL: https://ghidra-sre.org/ (visited on 10/04/2024).