TYPES FOR THE CHAIN OF TRUST: NO (LOADER) WRITE LEFT BEHIND

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Rebecca ".bx" Shapiro

DARTMOUTH COLLEGE

Hanover, New Hampshire

April, 2018

Examining Committee:

(chair) Sergey Bratus, Ph.D.

(co-chair) Sean W. Smith, Ph.D.

Devin Balkcom, Ph.D.

Greg Morrisett, Ph.D.

F. Jon Kull, Ph.D.
Dean of Graduate Studies

Abstract

The software chain of trust starts with a chain of loaders. Software is just as reliant on the sequence of loaders that ultimately setup its runtime environment as it is on the libraries with which it shares its address space and offloads tasks onto. Loaders, and especially bootloaders, act as the keystone of trust, and yet their formal security properties – which should be a part of any solid bootloader design – are both underappreciated and not well understood. This is especially problematic given the increasing adoption of loader-based code signing and execution enforcement mechanisms. My thesis digs deeply into how loaders have failed to earn our trustworthiness and how they may continue to harbor vulnerabilities even after memory corruption-based vulnerabilities lose their prevalence. In order to address these issues, I propose a *memory region*-based type system that allows us to better model a loader's intentions and thus mediate its behavior. More specifically, I show how a loader's execution can be broken down into a sequence of *typed* phases, each semantically classified as either a *bookkeeping*, *loading*, or a *patching* substage, while sections of memory are grouped into semantically related *regions* and assigned a type, based on their intended use, by which policy access decisions are made. I demonstrate the feasibility of this technique by applying it to Das U-Boot, a well-known and widely-used bootloader, with minimal changes to the bootloader's implementation. In order to do so, I designed and developed an extensive bootloader instrumentation suite to help analyze a bootloader's behaviors, construct a policy, and completely mediate operations, thereby enforcing behaviors governed by the type system's policy.

There is no moment in life that can't be improved with pizza.

–Daria Morgendorffer, from MTV's Daria

Acknowledgments

I owe a great deal of gratitude to my advisor, Sergey Bratus, for his guidance and patience. Anya Shubina, who played a much grater role in my PhD research than she may even realize. My committee for their insightful advice and direction. Travis Goodspeed and the Scooby Crew along with the generations of trust labbies all who have played a non-trivial role in the generation of this thesis. The Wellesley computer science department faculty and its few (but mighty) computer science students who charmed me into taking my first computer science classes so I could be captivated by its shiny puzzles and enchanting mysteries. My family and friends for their support and patience. Also, all those yaks I shaved in the creation of this thesis (none which were harmed). And you, the reader, even if you are not explicitly listed here, for endeavoring to drink from the fire hose that is this dissertation.

Contents

List of Figures

List of Tables

A chain of trust is a chain of loaders

–Sergey Bratus

1

Introduction

Wherever you encounter a general-purpose computer system, you will find a chain of one or more dynamic loaders, the first of which is known as a bootloader. As a system pulls itself into a state of usefulness and readiness, a tree-shaped chain of loaders are invoked rooted by the **kickoff bootloader**, each loading one or more binary images some of which are themselves loaders, occasionally branching when multiple distinct processors need initialization. Loaders are the centerpiece of the chain of trust, but few have a deep understanding of the bootloading process and even fewer have developed a bootloader or a general-purpose loader from scratch. Loaders are similar to electricity in that they are ubiquitous, in a relatively transparent and seemingly-magical way, but folks will lunge for their pitchforks if they stop working. A dynamic loader's primary objective is to breath life into a static and inanimate binary image so that it can takeover and execute on its own in whatever environment is setup for it by the loader.

Conceptually speaking, bootloaders appear to be fairly simple, as their core purpose is to transform a hardware system from its initial state upon reset into a more

flexible and useful system, eventually loading its target software, be that a full-blown operating system or a simple embedded application. Loaders that do not operate early in a system's boot process have fewer responsibilities, especially with respect to the hardware, and merely act as transducers of system state, bringing a static representation of an executable image to life. However, upon closer inspection, we will find that the mechanisms that allow loaders to achieve their end goals can be multifaceted and complex, and hence prone to bugs and vulnerabilities especially in the absence of a common security model of the loader's properties.

When it comes to bootloaders, the specific actions a bootloader performs (or may perform) are determined by the specifics of the hardware, overall requirements of the system, as well as the developers' goals of the bootloader itself. Many hardware-agnostic goals of the bootloader (such as locating the target image to be loaded) are achieved by hardware-specific mechanisms that do not generalize across architectures. Here lies a large part of the challenge inherent in both studying a specific bootloader and investigating loaders in general: it is hard to separate the generic behaviors of loaders from the hardware and platform-bound ones. This may be the reason why no generic models of loaders emerged: the analysis of general desirable behaviors gets bogged down in hardware or platform-bound details, yet such a generic understanding and separation are necessary for constructing trustworthy loaders.

Bootloaders establish the tone for the system, including for all of the loaders that are (loaded) descendants, yet they are they are the most challenging type of loader to instrument. Furthermore, because bootloaders initially operate in restricted environments with undefined state which requires that at least some of the loader be written in assembly language, it is challenging to make use of standard verification and runtime access control techniques. Despite all of this, bootloaders are still loaders. They perform linking and loading just like any other loader, and sometimes even operate on the same file formats application loaders work with, such as ELF and PE.

Therefore, although I have chosen to focus my dissertation research more specifically on bootloaders; the models, mechanisms, and techniques I use to describe and enforce safe behaviors for a bootloader will not only allow remainder of the loader tree be more trustworthy, but could also be applied to all other kinds of loaders.

The main contributions of my thesis, further expanded in section 1.3, are

1. tools to observe and define loading behaviors during boot,
2. formal apparatus for describing bootloader behaviors, and
3. enforcement mechanisms for these bootloader behavior specifications.

1.1 Our trust in bootloaders

Bootloaders, being the first code invoked by a system as it powers on, are implicitly trusted by all layers of the software they load and patch. There are many ways in which software *implicitly* trusts the originating bootloader, but one of the most noticeable ways arises due to how a processor's security mechanisms are configured and employed. Any processor with configurable security mechanisms must start execution in the most privileged state possible so that its protection mechanisms can be configured. Even if the system's *kickoff bootloader* does not directly configure the hardware's protection mechanisms, all subsequent software implicitly trusts the state in which the kickoff bootloader left the system. Moreover, there has been an increase in *explicit trust* of bootloaders over the past few decades as "secure" and "measured" boot mechanisms became incorporated into an increasing number of bootloading chains, which treat the bootloader as a *static root of trust*.

Code signing mechanisms have also been incorporated in other types of loaders, in both kernel module/driver loaders (such as 64-bit Windows drivers from Windows 8) and application loaders, which employ similar mechanisms to establish the trustworthiness of the code it is about to load and consequently also suffer from the same kinds of vulnerabilities.

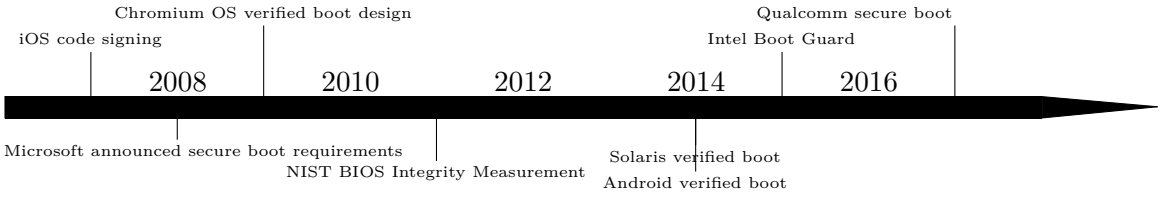


Figure 1.1: Instances of secure boot adoption since 2010

It is difficult to avoid the need to trust a system's bootloader, although mechanisms that act as *secure enclaves*, such as Intel's Secure Guard Extensions (SGX), have been developed to allow for limited computation environments in which (in theory) the executing software only needs to trust the processor, establishing a *dynamic root of trust*. Regardless of the existence of such secure enclaves, most software must trust the (boot)loader(s) that initialized the system on which they execute. And even so, as software that relies on SGX continues to mature, they will likely come to include their own loaders.

Few research teams are equipped with the knowledge and capabilities to discover (boot)loader-related vulnerabilities, especially with respect to loaders that run early in the system initialization sequence, yet there is no lack of discovered loader vulnerabilities. For example, in a talk name "Attacking and defending BIOS in 2015." presented at REcon 2015 [20], 38 BIOS bootloader-related vulnerabilities discovered between 2013 and 2015 were listed, which does not include any non-BIOS bootloader-related vulnerabilities (e.g. [217, 218, 220–223]). And yet, during this time period there are a number of introductions of "secure"-boot-like support mechanisms or implementations by Intel, Solaris, and Android. These and other bootloader-related vulnerabilities will be further discussed in section 2.2, where I will show how they follow certain structural patterns. I will further show how the region-based typed system described

by this thesis mitigates such failure patterns.

1.2 A different perspective on bootloader design

Over the past decade or so, industry has been moving towards more explicitly trusting bootloaders and relying on mechanisms that check bootloader integrity and origin of a bootloader image (e.g. measured boot), and more explicitly defining communication mechanisms between bootloader images of varying origins (e.g. UEFI). Yet little (if any) work has been done specifically to boost our overall confidence in a given bootloader's implementation correctness and safety.

1.2.1 Loader safety properties

Before we began to explicitly trust bootloaders and use them as a root of trust, we implicitly trusted (and continue to trust) that bootloaders (and other types of loaders) are capable of correctly:

- allocate non-overlapping address regions,[1]
- manage address alignment,
- prepare address space mapping,
- patch/link loaded images, and
- extract requirements and restrictions from the loaded resources themselves.

Regions of addresses that loaders (and especially bootloaders) manage can (and often do) change throughout the course of loading as the loader itself discovers new resources that require dedicated space in the system's address map. A resource that a loader discovers can be a device or subsystem, or it can be a software image that the loader is ultimately expected to map to one or more address regions and possibly execute.

[1] I mean **address** in a general sense: sequences of cells that are retrieved via a numeric identifier that indicates its index into the sequence. This can apply to both system memory and system I/O addresses, which on x86 are addressable via `in` and `out` instructions, or even the PCI-bus address space which can be accessed via memory/mapped registers *or* `in`/`out` instructions. Several such address spaces, disparate or interconnected, are encountered in a typical boot process.

In general, the purpose of a loader is to *discover*, *initialize*, and *allocate resources*. During this process of resource discovery, the loader ideally identifies and discovers a special resource – one that contains the binary image and metadata describing the loader's target executable. (For a bootloader, this target is the subsequent stage in the bootloading process.) Throughout a loader's course of resource discovery and initialization, it may encounter a resource that substantially changes the system's current address map(s) once configured. For example, this may happen if it discovers additional volatile memory or a section of code or data that it must map to or relocate in memory. Such behavior can be easily described – when additional memory is initialized, a region of physical addresses which initially does not have a backing store is suddenly usable later in the boot process – however it can be challenging to represent the separation between these two epochs in the loader's source code.

Throughout a loader's iterative process of resource discovery, initialization, and allocation, a loader is either:

- preparing and/or patching/linking a region of memory that will be used by a later phase in the loading process, or
- performing internal bookkeeping (a catchall for any other type of action)

With this simple insight in mind, we can begin to classify a loader's actions by *intent*, so we can establish a typing system for the loader. The objects upon which a loader operates are blocks of addressable bytes that can be assigned a type label, each of which is either bookkeeping data, patching/linking metadata, or within a region that the loader is preparing.

This treatment of memory regions is inspired by the typed regions of Cyclone, a safe dialect of C [109]. Cyclone assigns labels to regions of memory that can be simultaneously deallocated. Pointers declared in Cyclone source code can be augmented with these region labels, allowing the compiler to statically check for unsafe pointer dereferences (policy violations). Similarly, over the course of a loader's

execution, regions of memory change in "availability."[2] Finally, we observe that the "write" operation is a loader's fundamental action. Therefore, in this thesis I introduce address **r**egion-**b**ased **w**rite **a**ccess **c**ontrol – RBWAC[3]. The basic idea, which will be covered in greater depth in chapter 5, is that RBWAC consists of:

- *Substages:* which define loader execution as a sequence of behavior-based substages
- *Region-based access control rules:* for each substage, we formulate an address region-based write access policy

1.2.2 Designing a safe bootloader

In this dissertation I endeavor to narrow the gap between the trustworthiness we *demand* from our bootloaders and the trustworthiness we can *expect* from them. I will describe and focus on bootloader behavioral properties that are:

1. prevalent across bootloaders (and more generally, loaders),
2. whose correctness are important for a bootloader's safety, and
3. can be statically checked and/or dynamically enforced.

In order to accomplish this, I will first *demystify* the inner workings of bootloaders, then I will *decompose* their behaviors into *phases* of execution that can be formally *described* and ultimately *enforced*.

To allow us to analyze and decompose bootloader behavior in a hardware-agnostic manner into phases, I performed a qualitative descriptive study on a range of hardware and open source bootloaders. I also investigated bootloader and other types of loader vulnerabilities as well as other issues that leads to safety violations, in order to identify behavioral properties that are relevant to overall bootloader security.

Finally, to show that my ideas are both feasible and practical, I performed a *case study*, deriving, describing, and enforcing a behavioral security policy on an instance

[2] *Availability* as in whether the loader *may* access a region, *not* whether it *can* address a region since the later is a question handled by hardware.

[3] Pronounced: arrrrrrrrb-whack!

of the open source Das U-Boot bootloader compiled to execute on the BeagleBoard-xM with minimal changes to the original U-Boot source code. During the process of deriving an enforceable policy, I developed a bootloader instrumentation tool suite capable of dynamically tracing, analyzing, and enforcing certain types of bootloader behaviors. Also, to explore the feasibility of applying source code-based static analysis to a bootloader, I worked with Frama-C source code analysis framework to analyze a complete U-Boot stage.

My U-Boot case study demonstrates feasibility of applying a behavior-based policy on a bootloader executing on simple ARM system. However, to demonstrate that such safety mechanisms may work on a range of bootloaders and hardware of varying requirements and complexities, I will also discuss how these mechanisms may be used in other classes of bootloaders and hardware.

1.3 Contributions

In this thesis, I present a suite of tools for observing and characterizing bootloader behaviors. My tools work at a fine granularity – mediating every write a loader makes during execution – while also allowing us to coarsely characterize and aggregate these writes by purpose resulting in a concise description of bootloader behaviors. The major contributions of this thesis are:

(a) A survey of security-related loader failure patters,

(b) a generalization of these failures into a number of formally-defined failure classes,

(c) a formalism for describing the purpose and intent of loader memory accesses at an instruction-level granularity but with a convenient aggregation to memory regions,[4]

(d) a mechanism for enforcing behaviors as specified by my formalism,

[4]The region descriptions in this formalism naturally align with symbols and other ABI metadata constructs. To the best of my knowledge, this is the first practical formalism of its kind applied to legacy bootloaders.

(e) methodologies for refactoring existing bootloader code to take advantage of this formalism as well as for writing fresh bootloader code,

(f) methodologies for *incrementally* strengthening an existing loader's implementation, and

(g) a design and implementation of a policy language with which policy actors and objects can be defined based on both source code annotations and references to objects within a loader's own compiled image,

(h) a case study showing feasibility of this approach, applying this methodology to the well-known Das U-Boot bootloader

My goal is to provide industry bootloader programmers with practical tools and methodologies for writing bootloader code.

Loaders are the stepchildren of most systems. Users do not like them because they are simply another unnecessary delay on the way to The Answer. Compiler writers do not like them because they are parts of the system (what do you mean they're not?). System programmers do not like them because they are just another utility. Perhaps only the business manager likes the loader because so much time (and money) is spent executing it.

–Charles Wetherell, 1978
in "Etudes for programmers" [226]

2

Loaders and their vulnerabilities

Many a philosophical discussions have emerged from simply trying to answer the question of whether a particular behavior is a *feature* or a *bug*. When it comes to loaders, this very question can be difficult to address in a general way. Many of the loaders that are in use today were originally designed in the era when the **robustness principle** dominated – *robust* in the same vein as stated in the RFC for the Transmission Control Protocol, to "...be conservative in what you do, be liberal in what you accept from others" [103]. Some of these *robust* loaders have grown to become less lenient, often incorporating code signing mechanisms. However, only so much rigidity can be imposed on a loader's target executable whose file format is *explicitly* designed to be flexible. For example, there is clear evidence that ELF, the binary file format used in Unix-like systems, was designed to be a flexible format in a future-compatible way. This is evident both in the file format itself, how its use has evolved, and how one of the earliest references on ELF describes the format, e.g.,

> Some object file control structures can grow, because the ELF header

> contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore 'extra' information. The treatment of 'missing' information depends on context and will be specified when and if extensions are defined.
>
> – Page 13-4 of [12]

Because ELF was designed with such flexibility in mind, technical references for ELF rarely describe what ELF-processing software *should not* do – only *expected* behaviors are described. This intentional flexibility allows for a variety of interpretations that pave the way for these philosophical debates as to whether a particular loader's behavior is a vulnerability or a feature.

When there are many way in which a given object can be used and interpreted, we cannot always foresee its consequences. Bratus and Bangert identified discovered and presented a number of such unintended consequence of this hyperflexability by demonstrating methods of crating ELF files whose memory-mapped image loaded by the dynamic linker is seemingly unrelated to its memory-mapped image when directly executed [34]. This hyperflexability has other problematic consequences, one of which I discuss in section 3.1.1 (page 36), where I describe my prior research on ELF metadata-driven virtual machines, the results of which have heavily motivated this thesis.

The remainder of this chapter will skirt this philosophical debate by narrowing our focus onto a less-broadly defined loader vulnerability family of concern, in particular, *behaviors that result in unintended access to intentionally protected assets. Assets* can include data or code, *accesses* can entail reading data, modifying an asset, or code execution.

2.1 A loader's onus

Dynamic loaders are what breathes life into otherwise static binary images. They fill in the gaps between the programmer's view of symbols/shared resources, the compiler, the system's hardware instruction set (and its addressing constraints), and the system's runtime environment/resources. These requirements are often described in an *ABI* (application binary interface) specification. ABIs describe hardware-related details such as data sizes, layout, and alignment requirements; calling conventions; library and kernel interfacing conventions; and how binary images are formatted, including symbol resolution and relocation information. Each unique combination of hardware family and operating system family requires its own ABI specification, although, it is not hard to find commonalities between different ABIs that ultimately allow for binary-level interoperability between systems that adhere to different ABIs.

It is a loader's role to make sure separately compiled units of code work together as smoothly and transparently as possible, ideally without introducing more constraints onto the other actors it works with (with the exception of code signing requirements). In order to fulfill these expectations, loaders perform tasks such as: locating binary images in both volatile and nonvolatile memory, parsing and understanding the binary images, copying binary images, patching up unresolved symbols (code or data references) in a binary image, and managing resources imported or exported from a binary image.

During the past couple of decades, with the introduction of code signing, we have seen a loader's role transform into one most appreciated by business managers (see epigraph) – an arbitrator of executables. Code signing intends to control *which* software a system executes, sometimes for business-related reasons. Companies including Apple, Microsoft, Qualcomm, Motorola, Google, and Oracle (Sun Microsystems) all have developed and built code signing mechanisms into their loaders. Although these code

signing mechanisms are deeply intertwined with a loader's automaton, they are still divorced from the loader as computational objects, treated as an oracle by the loader.

Although, from a software design perspective it makes sense to have the loading and code signing mechanisms act as separate automata, this design can easily lead to a situation where the automata have diverging views of the system's state. A loader that has acquired a security role as an adjunct to code signing *should be designed and implemented with the same amount of care as the code signing mechanisms themselves.* This is the only path to making the links of the chain-of-trust and their code signing mechanisms eventually verifiable. My thesis recognizes this as a problem, and addresses it by introducing a formalism that semantically models loading behaviors, thus allowing for *complete mediation* of security-relevant behaviors.

2.1.1 The evolution of loaders

The duties of loaders have evolved throughout the short history of digital computing, which has resulted in today's de-facto understanding of what is expected from a runtime (dynamic) loader. In the earlier days of computing, runtime loaders helped automate the process of reading *absolute binaries*[1] (image formats that did not require further runtime adjustments) into memory. The DEC PDP-11 (circa 1970) made use of a fourteen-word-long bootstrap loader that could be feasibly be input by hand via a series of switches on the front panel as seen in figure 2.1 [61, pg. 11-9]. This hand-inputted bootloader instructs the machine to load an additional small number of instructions into memory which run diagnostics, determine from which device to boot, and read a binary image from that device into memory for execution. Some early loaders managed and executed binary images that were too large to hold in system memory using a technique called **overlaying**. Overlaying was commonly used in the 1960s to mid 70s, and then reemerged in the 80s until virtual memory techniques

[1]The term *absolute binaries* is rarely used today, however it is heavily used in Baron's 1978 *Assemblers and Loaders* book [18].

generally superseded the need for overlays [139] [2].

Before virtual memory came into use, *all* loaders needed to be sensitive to the system's hardware and required knowledge of the system's physical memory map. Loaders that operate within the context of virtual memory can be oblivious to the hardware's physical address layout, only having to worry about any restrictions or expectations posed by the operating system (or whatever software the manages the virtual memory). Nevertheless, there are still loaders that execute before virtual memory is enabled (i.e., boot loaders), which *must* still be cognizant of their hardware.

The adoption of virtual memory led the way for non-bootloading loaders to be developed in a more hardware-agnostic manner with only a small percentage of their source code comprised of hardware-specific instructions and actions. Most hardware-specific responsibilities – such as instruction and address formatting, byte order, alignment, and relative addressing – remain a task for the compiler/assembler to handle. It is the compiler that emits the most hardware-specific objects/information into the binary for the runtime loader to later interpret. Modern extensible binary image formats such as Mach-O (used in OS X), PE (used in Windows), and ELF (used in Linux)[3] encapsulate such information in a hardware-agnostic manner for the loader to later interpret[4].

This decoupling of loader implementation details from hardware along with the use of richly descriptive object file formats allowed the responsibilities of loaders to evolve and grow. As hardware requirements evolved, so did programming languages and software engineering practices. Thus, dynamic loaders evolved in-step with their hardware and software counterparts. As software began requiring increasingly complex symbol resolution rules, as well as dynamic object initialization, lookup,

[2]Although, it is possible to find use of overlays in modern hardware, such as by Silicon Lab's Si4010 chip.

[3]Both ELF and PE are descendants of the COEFF format.

[4]Older binary image formats, such as `MS-DOS .COM` files, `a.out` files, as well as some still-used bootloader image formats, carry minimal data beside the necessary machine code and statically-defined data.

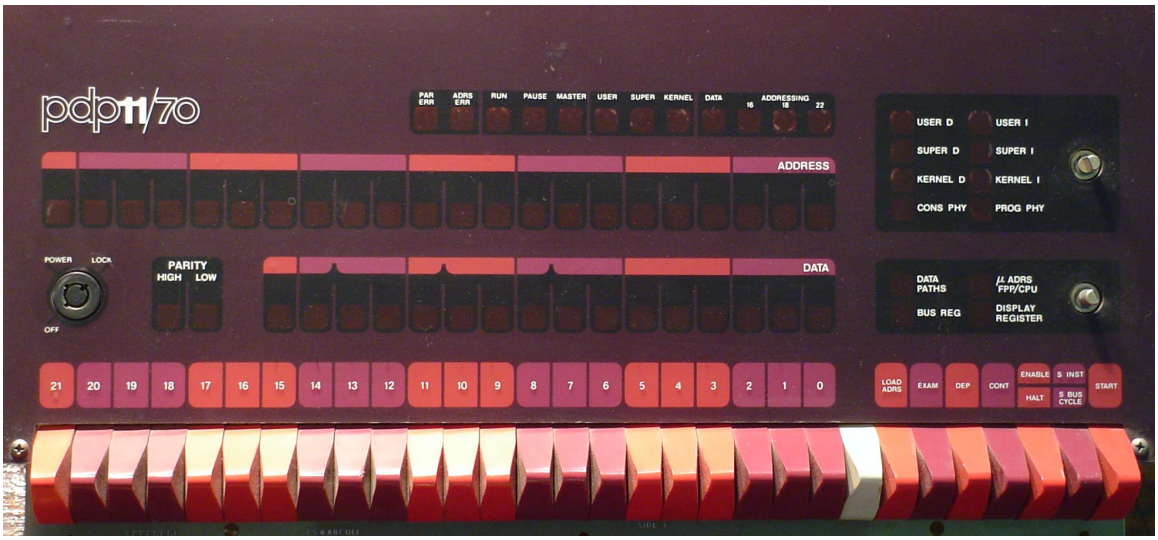


Figure 2.1: Front panel of a PDP-11/70 machine, switches on right-hand side manipulate machine state, switches on left-hand side are for manual input of data and address values. (CC BY-SA 3.0 Retro-Computing Society of Rhode Island) [179]

and destruction, loaders introduced symbol versioning, weak symbols, and C++ class/static constructor support. The GNU binary toolchain eventually incorporated support for embedding *arbitrary* symbol resolution rules into a binary in a hardware-agnostic manner by allowing the compiler to embed an arbitrary function, called an IFUNC (indirect function), that is executed to perform symbol resolution [166]. Mach-O took a different approach, using a dedicated bytecode to encode the same type of information in a highly-flexible manner.

The most popular binary file formats used by modern operating systems survived the test of time partially because they were able to adopt to the system's changing needs. Although loaders that execute during boot do not necessarily use these extensible formats, there are some notable exceptions, such UEFI which uses the TE file format (a more compact version of PE), and the bootloading code that runs in early stages of the ARM Linux kernel which handles ELF-encoded binaries. Although there are compelling reasons to allow for flexibility in binary file formats, there are various drawbacks which I will address throughout this chapter. Ultimately this thesis aims to find a sweet spot between flexibility and constraint by overlaying a flexible

intent-base enforcement mechanism on top of any existing loader implementation.

This history of loader evolution demonstrates:

1. Loaders evolved to become hardware-agnostic where possible, taking on new roles and abstractions to do so,
2. flexible binary file formats allowed loaders to more easily evolve to address the changing requirements of its hardware and software environment, and
3. loaders will continue to take on security roles in systems.

All of this motivates the need for another step in loader evolution. This thesis takes on the challenge of distilling abstractions needed to describe and model loading behaviors in a practical manner. These abstractions must be able to encapsulate a loader's intent because *when implementation does not accurately and completely encode intent, vulnerabilities surface.* Many examples of such vulnerabilities will be discussed later in section 2.2.

2.1.2 Loader safety properties

Regardless of whether or not a loader performs code signature checking, we still place a great deal of *implicit* trust in it. We trust our loader to properly initialize and allocate system resources, including the system's address space(s) and hardware. Not only must a loader be able to correctly allocate system resources (typically in a non-overlapping manner), but it must also be able to extract and handle resource requirements and restrictions, such as address alignment, from the resources themselves. On top of managing resources, a loader must also be able to locate, patch, and link any images it loads.

2.1.3 Properties of an address space: a loader's opus

Every operation a loader performs is reflected in the address space(s) it prepares for its target. It is the loader's duty to prepare the address space for its target and it must maintain a memory map (model) of the full address space it is composing,

or at the very least keep track of which regions it *can* and *cannot* reallocate. For example, a bootloader that is working with physical addresses must know which regions are not backed by RAM (e.g., memory-mapped hardware registers) thus can *not* be repurposed. Loaders typically have a more nuanced understanding of their address space and maintain a memory map that divides the addresses it controls into regions based on intended use such as: code, stack, heap, and space in which the target image can be loaded.

Memory maps, just like geographical maps, can be rendered with a variety of granularities, feature layers, and labels. These memory-related properties a loader maintains vary, not only between address regions, but also temporally as the loader executes. Loaders maintain these memory maps both explicitly (in data structures) and implicitly (hard-wired into its source code and the mechanics of its compilation toolchain), which makes it difficult to extract an accurate and complete representation of the loader's memory map in order to reason about the correctness and robustness of a loader's implementation.

Address spaces naturally lend themselves to a hierarchical semantic labeling scheme even though they are often modeled and treated as a flat sequence of consecutive intervals, e.g., with labels that differentiate between the stack, heap, static data, code, etc. An address that resolves to a memory mapped register could also be labeled with information that, for example, indicates what subsystem the register controls and whether the register is a status or control register. Although it is important for a loader to know whether a particular address is located within the stack region, it is not necessarily enough information to determine the intended use of the object at that address for some overarching access-control policy. Sometimes it *is* enough to just determine whether a particular address contains static data, but other times the loader *must* know that, for example, not only an address is static data but it also defines the location of the heap. Such information about an address is often

embedded into the source code itself by how it interacts with the address. However, in order to construct *useful security policies that govern loader behaviors* we must both recognize this semantic hierarchy and provide a means of representing and interacting with it within a security policy.

2.2 Loader vulnerabilities

Due to the extensible nature of loaders and their binary file formats, it can sometimes be difficult to classify a particular behavior as a bug – it is easy to be satisfied as long as the expected final state was achieved. What happens between the loader's invocation and the invocation of the target software it loads may not matter much – barring any failures, unexpected resource hogging, or accesses to privileged code or data that violate the system's policy – with exception to loaders that make use of code signing or loaders that restrict its target's privileges in some manner (e.g., via sandboxing or memory access permissions). With this in mind, I will focus on curbing behaviors that exhibit as *unintended accesses within an address space* using a *typed memory region*-based policy.

Public disclosure and knowledge of loader vulnerabilities is limited and occasionally inaccurate, therefore in this chapter, I will mainly focus on the few publicly disclosed and detailed loader vulnerabilities. Most of these disclosed vulnerabilities are triggered either by a crafted binary file or by some protocol the loader makes use of in order to acquire the binary file (e.g., USB), and result in either misleading the code signing mechanism or a privilege violation. I will later briefly discuss publicly-disclosed vulnerabilities where few details are publicly available. This all will help motivate my proposed type system which guards against such vulnerabilities.

2.2.1 Code signing

Loaders that employ code signing mechanisms typically *intend* to provide some protection against loading binary blobs from unknown origins. Code signing mechanisms

are also used to defend against modification of executable code between the time it is signed by the developer (or other trusted party) and the time it is executed. The mechanisms required to implement code signing add layers of complexity to loading, and with this complexity comes a larger and more prominent attack surface. In order to reap the benefits of code signing, we must be able to ensure that the loaded signed code is in an environment where it can not be overwritten (its integrity must be somehow assured), and any changes made to the image of the code in memory should be of a narrowly-controlled kind[5].

Code signing circumvention There is no shortage of weaknesses that can be leveraged in order to circumvent code signing and get untrusted code to execute. One can directly attack the code signing's cryptography by stealing private keys, such as what happened to an Adobe code signing certificate in 2012 [5]; one can try to disable the mechanism that enforces code signing (a technique that has been used to attack iOS code signing in [137, 225]); one can target mechanisms that execute unsigned code or affect control flow such as in other iOS code signing attacks [154, 155, 225]; or one can target the data included with and in the code signature itself such as in the Android master-key bug [79, 80] and the CVE-2012-0151 vulnerability for signed Windows PE files [87]. Husain, et.al., demonstrated how to bypass Chromium OS's verified boot by modifying the file system [101]. In my 2013 Shmoocon talk, I discussed former implementations of Linux kernel module signing that may be vulnerable to similar metadata-based attacks [190], as well as similar issues with various userland ELF executable code signing implementations in a earlier talk [35].

2.2.2 Out-of-bounds memory accesses

Bootloaders that make trust decisions or actively try to protect the system in some other manner have discernible delineations of privilege and access control rules which

[5]JIT (just-in-time compilation) performed during code execution are out of scope.

makes it relatively straightforward to label certain behaviors as in violation of its intended security policy. It is not as straightforward to identify problematic behaviors for loaders that were not built with such intentions in mind. Nevertheless, there are still certain behaviors any loader can exhibit that we can confidently label as a bug or potential vulnerability, namely: crashes due to memory exhaustion, an out-of-bounds read (such as a NULL pointer dereference), or an out-of-bounds write (such as a buffer overflow).

Out-of-bounds operations are a form of *type overlap*. These out-of-bounds memory operations are a form of, what I am calling, **type overlap**, which is when a single region (object) is assumed to be of a different (and conflicting) type by different actors[6]. Consider the classic stack overflow attack, a technique introduced by Aleph One [6], in which an attacker-controlled stack-based buffer overflow is leveraged to hijack execution flow by overflowing into and overwriting the return address stored on the stack. The way a processes' stack is laid out in memory is really a sketch of how the *memory regions* within the stack are *intended* to be used. Yet, when the return address is clobbered by an overflowing buffer we can see that the function who overwrote the return address had its own set of semantics it applied to that region of memory, intending to use it as storage for the object it is building, whereas the instructions that perform the return apply their own semantics to the same addresses. This clash is an instance of *type overlap*. *Type overlap* may also surface when two different actors, such as the loader and the loader's target software, interact with a single address space. For example, so-called **copy relocation violations** – when the linker/loader relocates a region that is expected to be read-only into writable memory – also an instance of *type overlap*, are not uncommon, and in fact, Ge, et. al. have found numerous examples of this in Ubuntu binaries [82]. *Every* out-of-bounds

[6]Sometimes this type of weakness is referred to as *type confusion*, but I will be overloading and broadening the definition of this term when I more formally define *type confusion* and *type overlap* in 5.2.6 on page 92.

memory access falls in the category of *type overlap* – by the very definition of an action being *out-of-bounds* we clearly know that actions performed are unsound with respect to the software's expectations and behaviors.

2.3 i$DEVICES/iOS bootloader vulnerabilities

iOS-based devices (which I refer to as i$DEVICES) make use of code signing at every link in their trusted boot chain. For many reasons, this particular chain of trust has been heavily targeted by security researchers. Consequently, it has a rich history of publicly documented vulnerabilities, and thus is an interesting staring point in my survey of loader vulnerabilities. In this section I will first provide a general overview of the iOS boot process, then I will summarize various loader-related vulnerabilities that have lead to circumvention of iOS's code signing requirements.

The goal of most iOS exploits is to be able to load and execute arbitrary code on an i$DEVICE, preferably retaining presence on the device between reboots. Most iOS loader-related vulnerabilities targeted by these exploits can be sorted into one of two major categories: **type overlaps** in the form of memory corruption (not just buffer overflows, but memory writes that *should not* have been allowed to occur), and **enforcement/validation failures** due to "features" unintentionally included in production devices.

2.3.1 iOS boot sequence overview

Throughout the past decade, a game of cat-and-mouse has played out between security researchers and Apple, causing the i$DEVICE chain of trust to strengthen and evolve, often growing in complexity. In the remainder of this section I will shed some light on the various ways this game played out. It is important to note that not every i$DEVICE loader vulnerability has been publicly acknowledged and even fewer technical details were released, so what I describe here is a limited picture of the larger situation. The fact that this game is still being played (it has been ten years since iOS was first

introduced in 2007) shows that current tools and techniques are insufficient, further motivating the series of tools I developed that help distill and describe a loader's intent (see chapter 4) and this thesis.

iOS's chain of trust can be roughly decomposed into the following loading stages:

- Stage 1: **BootROM**, the kickoff stage also known as **SecureBoot**
- Stage 2: **LLB (Low Level Bootloader)**
- Stage 3: **iBoot**
- Stage 4: **iOS Kernel**, with separate kernel extension and userland application loaders
    - **KEXT**, the kernel extension loader
    - **execv**, the userland loader
        - i: **launchd**, the first executed userland application
        - ii: **dylib**, userland dynamic linker and loader

2.3.2 Bootloader image format and storage

Not all iOS bootloader stage images are encapsulated within the same file format nor are they all physically stored on the same device. The iPhone has three kinds of nonvolatile storage, each of which are increasingly difficult to overwrite: (1) system storage (a.k.a NAND flash), (2) firmware storage (a.k.a. NOR flash), and (3) ROM. Researchers believe that the ROM is not writable, firmware storage is writable until the LLB is invoked, and system storage can only be written to from ring 0 (system mode). The kernel itself is stored in system storage and is mounted read-only so that it cannot be modified by untrusted software (without making use of a kernel vulnerability). Figure 2.2 illustrates the order in which each boot stage is executed and where each stage is stored. A check mark indicates whether a stage's image is signature checked before the stage is entered.

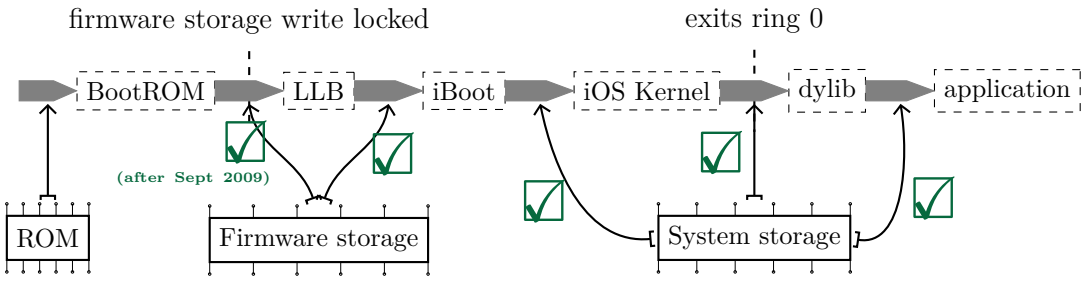


Figure 2.2: iOS boot stages (in dashed boxes) in their execution order (first boot stage is on the left) and corresponding image storage locations. A thin arrow from the boot storage location to a thick stage transition arrow indicates where the subsequent stage's image is loaded from. A checkbox next to such an arrow indicates that a signature check is performed on the image as it is loaded. The dotted lines indicate when firmware storage is write-locked (before the LLB stage is entered) and when ring 0 (system) privileges are dropped thus disallowing modification of system storage (when the dylib stage is entered).

2.3.3 The baseband (modem) boot process

iOS-based devices that have modems (such as iPhones) take additional steps during their boot process to initialize their baseband processor. Even less is known about baseband bootloading, although it is clear that the baseband's boot process has significantly evolved since the introduction of the first iPhone. The baseband in older models of i$DEVICES has its own boot chain that runs in parallel to the boot chain that initializes the application processor (described in the previous sections). This boot chain starts off with the baseband's own ROM-based kickoff bootloading stage. This ROM-based stage then loads the subsequent stage from its own NOR flash (separate from the application processor's NOR flash), which then loads the kernel (Nucleus OS), also from the NOR. Starting with the iPhone4S, the baseband processor stopped making use of separate flash to store its firmware and bootloader, instead booting from (presumably) on-chip ROM into an "emergency" service mode that waits for the application processor to send it a signed image which it then verifies, loads, and executes [1].

2.3.4 Type overlap loading weaknesses

The iPhone Dev Team's certificate-parsing exploit, presented in their 2008 CCC talk, was one of the first highly-detailed publicly-released iPhone exploits [175]. Back in 2008, the iPhone's BootROM did not verify the signature of the LLB (its second stage). Although it is *possible* to overwrite the flash storage containing the LLB, iOS's bootloaders restrict *how* and *when* this storage can be modified (see figure 2.2). Only the BootROM, LLB, and iBoot stages can modify firmware storage, and only during a special *recovery* boot mode. The code that *does* write to firmware storage *first* verifies the signature of the image it writes, which is important because the BootROM (in these earlier versions of iOS) did not check signatures. The iPhone Dev Team discovered a stack-based buffer overflow vulnerability within recovery mode which allowed them to craft a certificate that corrupts memory in such a way that causes signature checking to always succeed, thus resulting in them being able to control what is written to firmware storage during recovery mode. Ultimately, this allowed them to manipulate the bootloader running in *recovery* mode into writing an altered LLB image that does not enforce signature checking to storage, thus compromising the intent of iOS's trusted boot chain. Apple has since implemented signature enforcement in their BootROM of the i$DEVICE models targeted by the rest of the attacks I discuss in the remainder of this chapter.

The 0x24000 segment overflow attack, (also known as *24Kpwn*) was publicly released by the *iPhone Dev Team* in 2009. It exploited a BootROM vulnerability which ultimately allowed for an LLB stage to be loaded in a manner that bypasses signature verification performed by the ROM [55]. The LLB image is stored in an IMG3 file format which is chiefly composed of *tag-length-value*-style blobs of data (e.g. certificate, processor information, code, etc.), proceeded by a general header that provides information on the image's size and file offsets that are signed. In practice,

this design results in there being various length fields scattered throughout the image's metadata, some which are redundant, only some of which are protected by the file's signature. Researchers found that the BootROM relied on a non-signature-checked length value from which the number of bytes it loads into memory is calculated. They also found that the BootROM did not properly perform boundary checking to ensure that the loaded LLB image fits within the region of memory set aside for it. Therefore they could craft a LLB image that not only passed the BootROM's signature check but also corrupted the bootloader's own internal data when being loaded, thus giving them control over the chain of trust.

iBoot Environment Variable Overflow. In 2009, *geohot* found a heap overflow vulnerability in iBoot which could be triggered by setting a specially crafted and particularly long environmental variable in iBoot's console [84]. When crafted this way, it caused iBoot to overwrite an entry in its command table (which contains information on entrypoints to each of its console commands), thus allowing the attacker to redirect execution to injected shellcode when they later execute the targeted command. Examples of how this exploit is used are described in [10, 84].

The evasi0n userland code signing bypass, presented by the *evad3rs* team in 2013, made use of multiple vulnerabilities including one exhibited by iOS's application loader – the first in a series of jailbreaks that function by misleading the loader into overlaying two distinct segments (of code and/or data) from the loaded target executable on top of each other in memory [225]. In the version of iOS *evad3rs* targeted, the kernel depended on a userland daemon (called `amfid`) as well as the dynamic loader (`dyld`) to check the signatures of all loaded code (formatted as Mach-O files). `dyld` additionally required that the Mach-O header of all loaded Mach-O files be marked as executable to prevent itself from loading binaries whose metadata have been altered. This particular implementation of code signing happened to allow data to be appended to the end of a Mach-O file without invalidating its signature However,

if the Mach-O header was (naively) tampered with in order to get the loader to copy these appended bytes to memory, the file's signature would be deemed invalid. *evad0rs* developed a clever way of injecting extra metadata at the end of the file that tricks the loader into loading the unsigned (not marked as executable) metadata appended to the end of the file *on top* of the already loaded and validated Mach-O metadata. Using this technique, they could make the loader overwrite the region of memory containing the Mach-O's signed metadata with their own crafted data. Ultimately, *evad0rs* was able to overwrite data used to locate the signature checking function so that all future calls to perform signature checking would *always* signal success.

Pangu is a jailbreak that, like *evasi0n*, accomplished some of its goals by crafting a binary file so that multiple distinct segments defined in the binary got mapped to overlapping regions of memory by the loader – a technique I will now refer to as *segment overlap* [138]. At the time *Pangu* was released, Apple had patched the loader so that it was no longer vulnerable to *evasi0n*, adding checks to prevent segment overlap. *Pangu* achieved segment overlap via a different mechanism than *evasi0n* – an integer overflow vulnerability present in the segment overlap-checking code which caused the loader to overwrite a previously loaded and validated region in memory with a payload of their choice.

Pangu8 was introduced after Apple patched the integer overflow vulnerability targeted by *Pangu*. The *Pangu8* jailbreak that made use of a different integer overflow vulnerability to achieve the segment overlap, yet again coercing the loader into overlapping data of the attacker's onto a memory region that was already validated and trusted [63]. This vulnerability was partially fixed in iOS 8.1.2, but Pangu was able to again tweak their method in response to the patch until such vulnerabilities were more aggressively patched in iOS 8.1.2 [169].

TaiG was then introduced after *Pangu8*'s overlapping segment attack was rendered moot by iOS 8.1.2, continuing with the tradition of overlapping segment-based attacks. The *TaiG* exploit, a riff on *Pangu8*, targeted a different integer overflow vulnerability which also allowed for segment overlaps. This vulnerability targeted by TaiG was later patched in iOS 8.1.3 [138].

The alloc8 exploit is a more-recently discovered iPhone 3GS BootROM exploit [13]. *alloc8* makes use of a bug in the BootROM's heap implementation. The BootROM's heap returns 0x8 instead of the standard `NULL` (0) when it is unable to fulfill a memory allocation request. Some services that use this heap incorrectly assume the allocater returns NULL on error, and thus treat any returned value other than NULL as a valid address. Consequently, a service may write data to address 0x8 – which happens to be an address with the table of exception handlers – instead of catching the failure. In order to exploit this vulnerability, an attacker must (1) fill up the heap, (2) trigger the vulnerable code to overwrite the exception table so that the function pointer is clobbered with a value of the attacker's choice, ideally an address that the attacker knows to contain shell code, and (3) cause the overwritten exception's handler to be triggered. This can all be achieved by coercing the bootloader, via its recovery mode console, to load a large number of IMG3 files until the heap is filled, and then passing it a specially crafted IMG3 file to be loaded and parsed (while also clobbering the exception table), ultimately redirecting execution to a location of the attacker's choice.

The Pegasus spyware, which achieved persistent unsigned code execution in iOS, was reverse engineered by researchers at Citizen Lab and presented at CCC in 2016 [19]. It leveraged a collection of userland and kernel exploits to achieve persistence across reboots. *Pegasus* made use of a legitimate (and signed) developer tool called *jsc*, which harbored a memory corruption vulnerability that allowed for arbitrary code execution. The attack worked by replacing the executable of a non-essential daemon with *jsc* so it could be persistently leveraged to eventually gain arbitrary kernel execution

privileges (by making use of several other vulnerabilities), which ultimately allowed for an attacker to disable code signing.

A series of USB/serial console memory corruption vulnerabilities in the iOS bootloader's recovery mode console were also discovered. These exploitable vulnerabilities could be triggered via specially crafted packets that cause the loader to overwrite some function pointer with an attacker-controlled value, ultimately allowing the attacker to hijack the loader's control flow. Researchers leveraged a number of these vulnerabilities to achieve code execution in i$DEVICE bootloading chains over the years [85, 106, 174]. Although there are few public details on these vulnerabilities, we do know that these vulnerabilities are mostly related to heap management [96].

2.3.5 Loader enforcement/verification failures

Some of iOS's system variables are intended to be read-only. However, in a talk Levin gave at the RSA conference in 2015, he described various sensitive system configuration variables that were unintentionally writable, which, if changed, allow for unsigned code execution [138]. For example, the two variables `security.mac.proc_enforce` and `security.mac.vnode_enforce` could be modified, and if either `security.mac.proc_enforce` or `security.mac.vnode_enforce` was set to `false` then various aspects of code signing would be subsequently disabled. Apple has since disabled write access to such variables.

Diags, short for diagnostics, is an iBoot console command which executed code at a parameter-specified address, bypassing any code signing [105]. It appears to have been intended for developer use only but was accessible in certain versions of iBoot.

ARM7 Go was independently discovered by Chronic and the iPhone Dev Team. This vulnerability was present in the second generation iPod touch's firmware and allowed them to tell the device's coprocessor (a secondary ARM chip that has access to hardware-based cryptographic functions, memory, and can write to firmware

storage) to execute arbitrary code by sending it a command called `arm7_go` with an environmental variable set to the address at where to begin execution [54]. This "feature" is thought to be a debugging tool that was accidentally not stripped from firmware present in production devices.

The iPhone Dev Team's 2008 baseband exploit, also presented in their 2008 CCC talk [175], targeted the baseband's own nonvolatile firmware storage, which not only contains firmware images but also special region of data called the *seczone* which contains sensitive (intentionally protected) system state. This firmware can only be overwritten when the baseband is in a special *service* mode. Data to be written to the firmware must be wrapped in *secpac*s, which include signed metadata that describe *how* and *where* the firmware is to be overwritten its payload. The service-mode software that is capable of overwriting firmware enforces an additional set of rules that restrict what regions withing the firmware that a secpac can overwrite which include seczone protection. Researchers found that they could craft a secpac using address manipulation tricks in such a way that these restrictions are not properly enforced, resulting in them being able to modify the seczone. Most baseband exploits developed this attack was published also use various instances of memory corruption to achieve their goals [156].

2.4 Loader vulnerabilities beyond iOS

Although some number of iOS loader-related vulnerabilities have been publicly disclosed and documented, iOS is certainly *not* the only software stack which has been found to harbor loader vulnerabilities. In this section I will discuss publicly-documented vulnerabilities discovered in other software stacks, and show how many of these vulnerabilities are similar to those discussed in the previous sections in that, they too, suffer from **type overlap** and **enforcement/verification failures**.

2.4.1 Android phone loader vulnerabilities

Security researchers have also been interested in studying Android's chain of trust. Each Android phone manufacturer (OEM) can build custom bootloaders to enforce code signing, and so code signing-related vulnerabilities are often OEM-specific or phone model-specific. In 2014, Harrison and Li gave a talk at Shmoocon that documented a series vulnerabilities the in Samsung's chain of trusted loaders, a chain of trust that was designed to prevent the system from loading custom kernels [91]. The series of bugs they presented boiled down to variations of the bootloader trusting data that the adversary is capable of controlling and resulted in verification/enforcement failures. For example, in one version of the Samsung Galaxy Note II's boot chain, the bootloader only checked cryptographic signatures of partitions with specific names. This bootloader *expected* all partitions it made use of, such as those containing the kernel, to be named in a particular manner, but in reality, partition names had no technical significance beyond verification checks. Not only were partition names inconsequential, but also their names and contents could easily be modified to trick the bootloader into loading and executing an unsigned kernel.

2.4.2 UEFI

UEFI (the unified extensible firmware interface) has recently received a lot of attention as it is now **The Standard** followed by most general-purpose computer manufacturers. Although folks commonly use the term UEFI to refer to a system's firmware (bootloader), it actually refers to a *specification*, and thus it is more correct to refer to firmware as *UEFI-compliant*.

A handful of research groups who have spent a significant amount of time studying the UEFI specification and UEFI-compliant bootloaders have discovered a myriad of vulnerabilities, some of which are listed in appendix B's table of vulnerabilities. Many of the vulnerabilities discovered by these researchers fall into the category of

improperly protected UEFI variables (non-volatile UEFI configuration data) such as [20, 40, 43, 113, 121, 126, 149]. There are also many discovered vulnerabilities that allow underprivileged code to write to privileged memory/directly execute privileged code (e.g. forms of *type overlap*) such as [20, 39, 42, 72, 75, 99, 100, 126, 149, 200, 228], as well as other types of memory corruption vulnerabilities [120, 121].

2.4.3 Game Consoles

Game console bootloaders many of which employ their own cryptographic code-validation mechanisms to protect their intellectual property and prevent arbitrary software from being run on it, have also been of interest to security researchers. Researchers have mostly targeted these systems' cryptographic mechanisms, as their end-goals are slightly different from those who targeted phone-based bootloaders. Therefore, there are not as many publicly-disclosed memory corruption-based exploits in this realm.

The XBox (released in 2001), which made use of symmetric cryptography to encrypt their bootloader images and it stored its kickoff bootloader, which performed bootloader decryption and verification, in the system's chipset. In 2002, researcher *bunnie* reverse engineered this initial boot stage and extracted its encryption key [97]. In 2008, Busing and Marcan demonstrated that the Wii game console's code signing mechanisms were relatively weak and signatures that pass its signature check could be quickly/cheaply brute-forced [41].

2.4.4 BIOS

BIOS-type firmware (non UEFI-compliant) have also been of interest to security researchers. Consequently, a number of BIOS vulnerabilities have been discovered in which underprivileged code writes to a privileged region of memory, an instance of *type overlap*, such as [11, 20, 24, 38, 51, 73–75, 88, 93, 118, 127, 229–231].

2.4.5 Everything else

There is little public information available about vulnerabilities present in other loaders, however a best-effort summary of loader and bootloader vulnerabilities and their categories can be found in appendix B. The little information I found about each of these vulnerabilities does paint a telling picture. The table that summarizes these findings (table B.1 on 159), shows an overwhelming number of vulnerabilities in the form of *type overlap*, and for those that could not be clearly classified as type overlaps are at the very least likely to be forms of *enforcement* and/or *verification* failures.

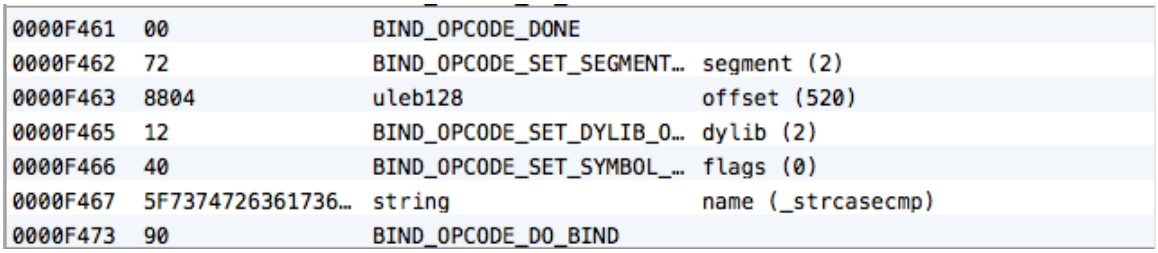
Although I mostly focused on *loader-based* vulnerabilities, is important to highlight recent studies on firmware security. An embedded system's firmware often includes some sort of a bootloader, not just to initialize the system, but to also bootstrap firmware upgrades. In 2014 a group of researchers from Eurocom performed a large-scale vulnerability study of firmware, testing consumer electronics firmware images for unprotected credentials and vulnerable configurations. They found a significant number of cryptography-related vulnerabilities, such as hard-coded credentials, as well as software with known vulnerabilities embedded in firmware images.

2.5 The language of loading

Loaders are **virtual machines**[7]. Their operations are shaped by the environment in which they run, and their bytecode is contained in the image they load. Some loader implementations, such as Mach-O, have gone as far as *explicitly* implementing a higher-level language for their loader. These (sometimes) ad-hoc languages define commands which embody the higher-level procedures the loader performs and provide insight into how the loader's codebase (and its static data) can be *semantically* partitioned.

The Mach-O file format, chiefly used by the iOS and OSX operating systems, explicitly encode various forms of metadata into a custom bytecode. Mach-O files

[7]Here I use the term *virtual machine* in the more general sense – meaning *interpreter*.



0000F461	00	BIND_OPCODE_DONE	
0000F462	72	BIND_OPCODE_SET_SEGMENT…	segment (2)
0000F463	8804	uleb128	offset (520)
0000F465	12	BIND_OPCODE_SET_DYLIB_O…	dylib (2)
0000F466	40	BIND_OPCODE_SET_SYMBOL_…	flags (0)
0000F467	5F7374726361736…	string	name (_strcasecmp)
0000F473	90	BIND_OPCODE_DO_BIND	

Figure 2.3: Screenshot from MachOView's interpretation of a Mach-O file's loader commands (*bytecode*). The left-most column indicates the instruction or instruction parameter's location in the Mach-O file, the second column contains a hex representation of the instruction or data, and the final columns contain more human-readable information on the bytecode such as instruction name and parameter type.

begin with a special header containing a magic number, CPU information, file type information, flags, the number of (variable length) loading commands, and the length of this block of loading commands (the loader's *bytecode*, a.k.a. *binding bytecode*). Mach-O's binding bytecode operates in a transactional style: it sets up a description of how an imported symbol's state should be altered (via a sequence of bytecode instructions), and then commits the transaction via a `BIND_OPCODE_DO_BIND` instruction. Figure 2.3 shows an example of such bytecode, in particular it contains a sequence of bytecode that instructs the loader to (1) reset the loader's bind (dynamic linker) state (`BIND_OPCODE_DONE`), (2) set the loader's state so it knows where to patch, [`BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULIB`], (3) set the loader's state requesting it locate the `_strcasecmp` function [`BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM`], and (4) perform linking based on this current state [`BIND_OPCODE_DO_BIND`].

Some bootloaders feature interpreters that support a high-level loader scripting language, such as the widely-used GRUB and U-Boot bootloaders. The UEFI forum has, in fact, published a formal language specification detailing each command's interface and semantics [210], and so this language is supported by most UEFI-compliant bootloaders including Tianocore, the UEFI-compliant bootloader reference implementation [205]. All of GRUB, U-Boot, and UEFI have their own variations of commands that embody high-level loading operations, such as copying

a target image to memory, which can be accomplished by the `kernel` command in GRUB, U-Boot's `loadp` command, and UEFI's `mm` command. Each of these scripting languages are tangible displays of how each bootloader is *intended* to function and behave. Error conditions defined by these scripting languages also document how each bootloader is *intended* to *fail*. What lies between these *intended* behaviors and failures are *non-explicitly* intended behaviors – and this is where vulnerabilities often lurk.

A well-installed microcode bug will be almost impossible to detect.

–Ken Thompson, 1984
in "Reflections on trusting trust" [204]

3

Obstacles to loader analysis

In this chapter I describe the many obstacles there are to loader analysis, bootloaders in particular.

3.1 Loader metadata and questionable behaviors

Loaders perform powerful transformations as they map binary images into memory and prepare them for execution. Modern executable file formats, such as Mach-O (used in OS X), PE (used in Windows), and ELF (used in Linux), include metadata that instruct the loader as to what transformations to perform such as what locations in its image need to be patched with the absolute address of some object in memory – absolute addresses are often unknown until runtime. These type of metadata allow a single executable file to be adapted to its environment, regardless of exactly *where* and *how* it is laid out in memory.

Existing loader designs do not yield to meaningful analysis of their memory accesses. This is something we *must* address, especially now that loaders are becoming the cornerstone of trust. Loaders are *machines driven by the data they process*, and when

we do not significantly constrain them, it becomes hard, or even impossible, to make any security guarantees about the loader or the system's state when the loader finishes executing. *We must think of these and **all** data as a program that drives a (hopefully!) constrained virtual machine.* A loader's machine performs essential memory operations that *can* and *should* be *explicitly described* and *enforced* (which I do with types and will discuss in chapter 5). Unfortunately, this is where current loader descriptions and designs come up short.

Application and library loaders that exist in general-purpose operating systems are notoriously complex. It is neither hard nor surprising to find examples of loader vulnerabilities, or, in the case in which its classification as a vulnerability is up to debate, "surprising behaviors" that arise due to this complexity which I explore in this chapter. These so-called "**weird machines**" [36], this emergent behavior appearing in the form of an *unintended* virtual machine, can manifest from the manner in which a loader processes the metadata in its target. Although some of these weird machines may not involve precise vulnerabilities (besides the ability to obfuscate the binary's execution), we will see that these weird machines can be powerful enough to make one question not just the *trustworthiness* of a particular loader implementation, but also the *correctness* and *comprehensiveness* of the loader's design and behavioral requirements.

3.1.1 ELF metadata: accidentally Turing-complete

The influence a binary image's metadata has over its surrounding system has historically been underestimated and underappreciated. Such assumptions can undermine a system's security, as I have demonstrated by determining how to compile the Turing-complete language Brainfuck into ELF metadata. I have since learned that these findings have directly affected designs of real-word code signing mechanisms. This research, and the research I conducted on the Mach-O file format, which I describe

later in this chapter, have greatly motivated my thesis's overall direction.

The AMD64 System V Application Binary Interface (ABI) specification on which the AMD64-specific ELF objects are based defines two types of metadata structures that drive the loader's transformation process: (1) symbol metadata and (2) relocation metadata. These symbol and relocation metadata instruct the loader in calculating which addresses to patch and the values to inscribe. When loaders trust relocation metadata, the metadata can mislead it to unintentionally manipulate objects the loader assumes to be invariant during relocation, such as objects the loader itself uses to keep track of its progress. When we do not explicitly model these implicit expectations and can or do not verify or enforce them, vulnerability prevention becomes intractable.

This weird virtual machine, driven by the GNU glibc suite's loader, processes each relocation entry metadata (virtual machine bytecode) in an ELF sequentially and maintains the equivalent to a virtual instruction pointer (IP), stored *in memory*, which contains the address of the next relocation entry (bytecode) to be processed. There is nothing special about the location or value of the IP from the prospective of the loader, it is merely a pointer stored "in-band" with all the other values the loader manipulates. If a relocation entry instructs the loader to patch the IP, the loader will happily oblige. In my 2013 WOOT publication, I discuss how ELF metadata can be crafted as a Turing-complete bytecode when processed by the Ubuntu 11.10's AMD64 gcc toolchain (`eglibc-2.13`), by demonstrating how to build a Brainfuck-to-ELF-metadata compiler [191]. I have since expanded this compiler to include standard input and output functionality, as documented in [188, 189]. This Laissez-faire attitude towards metadata that allows the loader to drive a Turing-complete virtual machine puts us in a position where it is difficult, even impossible, to analyze properties of a piece of software after it is loaded into memory.

Although I have focused on an emergent Turing-complete behavior thus far, it is rarely the case where an attacker needs Turing-complete power over a target in

order to compromise it. A successful attack tends to involve overwriting sensitive data and/or leaking sensitive data . Rarely is a full Turing-machine required to accomplish these tasks. Attackers are merely looking to elevate their privileges. Nevertheless, as Felix 'FX' Lindner says, "You can't argue with a root shell". If we have Turing-complete execution environments lurking in our software, we will have a hard time asserting *any* security properties in our software.

3.1.2 A root shell backdoor embedded in ELF metadata

To demonstrate how sub-Turing-complete weird machines are useful from the perspective of an attacker, I constructed a proof-of-concept attack by carefully constructed ELF metadata that do not make use of any Turing-complete properties of the loader's execution environment, but allow the attacker to insert an obfuscated root shell (a console with administrative privileges) backdoor in `ping` by merely manipulating `ping`'s metadata.

`ping`, a widely-used utility that allows users to quickly test if they can send and receive packets to a remote system, is not a particularly unique piece of software besides the fact that many versions of it run as `setuid` root. `setuid` root means that regardless of *who* executes the program, the binary will be provided root privileges in order to accomplish its task, typically dropping these elevated privileges as soon as it is done with them. It is possible to inject crafted *metadata* into the `ping`[1] executable that causes it to expose a shell running with root privileges to an ordinary user.

There are two features of the `ping` implementation I take advantage of to implement this backdoor: (1) `ping` runs as root, but drops privileges using a call to `setuid`, and (2) `ping` has an optional `--type` command-line argument that takes a single argument that customizes the type of packets sent. If provided, `ping` tests the argument (which I call `<string>`) in the following manner –

[1]More specifically, the version of `ping` in Ubuntu's `inetutils v1.8` package.

```
if(strcasecmp (<string>, "echo") == 0)
```

In this code segment, `<string>` is the command-line argument to the `--type` option.

The location of library functions such as `strcasecmp` and `setuid` are not known at compile time, and one role relocation metadata plays is to instruct the runtime loader (and linker) which offsets into the image, such as those that should contain the address of a library function, need to be updated with such information that is only known at runtime. When library function is called for the first time while `ping` is executing, the dynamic linker uses these relocation metadata to patch the function pointer's value. Given a compiled `ping` binary, we can generate a backdoored `ping` binary with altered, but well-formed, relocation metadata that instruct the loader to overwrite the `strcasecmp` and `setuid` function pointers to redirect these function calls to locations of our choosing.

More specifically, we can build a backdoor that gives a regular user the ability to execute arbitrary programs as root by injecting crafting metadata that when interpreted by the loader does the following:

1. Overrides the call to `setuid()` with a different function so privileges are not dropped (preferably with a function that does not produce any noticeable side effects such as `getuid()`), and
2. overrides the call to `strcasecmp()` with a call to `execl()`.

This keeps `setuid()` from being called thus preventing `ping` from dropping root privileges. It then coerces `ping` into calling `execl(string, "echo")` instead of calling `strcasecmp(string, "echo")`, causing `ping` to treat the value of `string` as a path to a file to execute (via its call to `execl()`). The string pointed to by the variable `string` is directly obtained from the value of the `--type` command-line option, and so the user can supply the path to the executable (such as a shell) that they wish to be executed as root as an argument to `ping`.

With this backdoor in place, `ping` operates normally unless the `--type` argument

```
Symbol table '.sym.p' contains 90 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 000000000060dff0     8 FUNC    LOCAL  DEFAULT  UND

Relocation section '.rela.p' at offset 0xf3a8 contains 14 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend

00000060dfe0  002d00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
00000060e9e0  004e00000005 R_X86_64_COPY     000000000060e9e0 __progname + 0
00000060e9f0  004b00000005 R_X86_64_COPY     000000000060e9f0 stdout + 0
00000060e9f8  005100000005 R_X86_64_COPY     000000000060e9f8 __progname_full + 0
00000060ea00  005600000005 R_X86_64_COPY     000000000060ea00 stderr + 0
00000060eb40  000000000005 R_X86_64_COPY                 0000000000000000
00000060eb40  000000000001 R_X86_64_64                   0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                 0000000000000000
00000060eb40  000000000001 R_X86_64_64                   0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                 0000000000000000
00000060eb40  000000000005 R_X86_64_COPY                 0000000000000000
00000060eb40  000000000001 R_X86_64_64                   00000000000be6e0
00000060e028  000000000001 R_X86_64_64                   0000000000000000
00000060e218  000000000008 R_X86_64_RELATIVE                 0000000000401dc2
```

Figure 3.1: Snippet of `readelf` utility output showing a root shell-backdoor in `ping` (ELF). Added/changed metadata highlighted in green.

is used, in which case it retains its root privileges and executes the argument to `--type` as root. The details of how this is implemented is documented in in [191], however the basic idea is that we augment the executable's metadata with a set of relocation entries that, (1) locate the base address of `libc` based on statically-known addresses of ELF metadata in memory, and (2) add the known offset of the `libc` function we want to be called to the base address of `libc` (e.g. `execl()`), so that the pointer to `strcasecmp` is overwritten with this value. Figure 3.1, which is a snippet of human-readable description of the augmented ELF executable generated with the `readelf` utility, gives a sense of how much metadata were added in order to implement this backdoor in ELF.

3.1.3 Mach-O: wee bytes with potent consequences

Mach-O is a slightly more evolved executable file format than ELF – much of its metadata are encoded in a specialized bytecode, discussed in more detail in section 2.5 (page 32). This linking/loading bytecode lends itself more easily to the same types

of analysis techniques we already apply to other languages and bytecode. This is a notable step in more precisely and explicitly defining a loader's behavior, but unfortunately it stops short of formally describing the access properties of the loading computation (a deficiency addressed in this thesis). ELF metadata, in comparison, do not easily lend themselves to be understood in such a manner because ELF's binding-related metadata are scattered throughout the file and whose interpretation are system-specific and context-sensitive.

Both the Mach-O and ELF formats encode similar information and are processed by loaders that perform similar operations, so is an interesting exercise to insert the same backdoor in a Mach-O-formatted version of `ping` as in the ELF version. I have found that Mach-O's condensed formatting of linking/loading metadata allows for more concise expression of the same backdoor. In fact, the same backdoor can be implemented by merely modifying eleven bytes of the Mach-O-formatted `ping` (in comparison to the 100+ bytes inserted and modified to achieve the same affect in the ELF-formatted `ping`). These eleven modified bytes belong to the string "_strcasecmp" embedded in a loader command specifying that the function of this name should be imported. We modify the bytes of the "_strcasecmp" string that is embedded in the binary's linking bytecode to be "_execlp

x0

x12

x12

x12

x12" which, when interpreted, cause the loader to locate the `_execlp` function (instead of `_strcasecmp`). It then interprets the byte following the null-terminated "_execlp" string, 0x12, as well as the subsequent 0x12 bytes as one-byte long instructions that essentially act as `noop`s. This modification is shown in figure 3.2. Similarly, we can overwrite the bytecode for importing the `_setuid` function by changing its embedded

Original `ping` | `ping` with backdoor

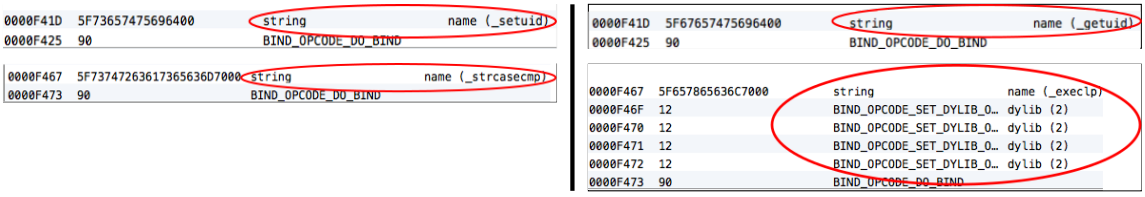


Figure 3.2: Visualization of the binding bytecode implementing the Mach-O `ping` backdoor, overriding setuid. Original bytecode on left, patched on right. The left-most column indicates the bytecode's location in the Mach-O file, the second column contains a hex representation of the instruction or data, and the final columns contain human-readable information on the bytecode such as instruction name, parameter type, or parameter value. (Screenshots taken of MachOView.)

```
< 0016720: 7400 5f73 6574 7569 6400 5f73 6967 6e61 t._setuid._signa
---
> 0016720: 7400 5f67 6574 7569 6400 5f73 6967 6e61 t._getuid._signa

< 0016740: 005f 7373 6361 6e66 005f 7374 7263 6173 ._sscanf._strcas
< 0016750: 6563 6d70 005f 7374 7263 6872 005f 7374 ecmp._strchr._st
---
> 0016740: 005f 7373 6361 6e66 005f 6578 6563 6c70 ._sscanf._execlp
> 0016750: 0012 1212 125f 7374 7263 6872 005f 7374 ....._strchr._st
```

Figure 3.3: Bytewise diff between original and backdoored version of Mach-O `ping` containing position in file (left column of digits), a hexadecimal representation of the modified bytes (middle column) and an ASCII representation of the bytes (right column). Colored values highlight differences between the original (blue) and modified (red) binaries.

string to "_getuid". A byte-wise diff, highlighting the eleven modified bytes, is depicted in figure 3.3.

3.1.4 A PE metadata-driven packer

PE metadata can just as easily be misused in powerful ways. One telling example of loader metadata misuse was demonstrated by *skape* in 2007 [195]. In this paper, *skape* demonstrates how we can create a well-formed PE executable with unintended loading-induced side-effects driven by its metadata. He shows how to craft PE relocation metadata that alter binary's image in memory – effectively acting as a binary unpacker.

Binary unpackers allow portions of a binary to be stored as an obfuscated blob on disk, and perform the runtime translation needed to ultimately execute the binary. Unpackers, and crafted metadata in general, can make static analysis more difficult, especially when they take advantage of other design weaknesses potentially present in a loader.

3.1.5 Loader metadata-based parser differentials

Parser differentials, a term coined by Len Sassaman, Meredith Patterson, and the *language-theoretic security* (*langsec*) movement, arise when two different parser implementations interpret the same data differently. Parser differentials, such as those found in the X.509 certificate authority infrastructure [122], can have profound security consequences. They are also sometimes the root-cause of vulnerabilities in trusted loaders, such as in the case of the Android "master-key" exploit [79, 80].

Linker metadata formats often have redundant representations of the same data, and ELF has many examples of such redundancy. Julian Bangert and Sergey Bratus, in their talk named "ELF Eccentricities", demonstrated multiple ways to take advantage of these redundancies so that the Linux kernel and the popular IDA reverse engineering toolkit interpret the same file differently, disagreeing on the entrypoint's address [15, 34]. When a binary analysis tool does not agree with the runtime as to what code is run, all analyses it performs are rendered moot.

3.2 Reining in a loader

In some ways, a loader is no different from any other piece of software; many of the techniques we use to develop safer and more robust software can also be applied to loaders. Yet, as I've discussed throughout previous chapters, loaders are also an interesting class of software in and of themselves worthy of special consideration. In this section I will discuss techniques used to ensure software safety and correctness with a special focus on loaders and systems software.

A software's design, implementation (including static analysis and testing), and runtime environment all greatly influence the software's safety and correctness in practice. The requirements of a loader are determined by the system's hardware, software environment, and compilation toolchain – and thus many of its design requirements are inflexible. In order to build a safer loader we must either focus on the loader's implementation/runtime environment or we must redesign and rearchitect parts of the rest of the system. My thesis has chosen to focus on the former, but there are a number of examples of research that worked on the later.

3.2.1 Implementation considerations

Loaders are typically implemented in C (with a sprinkling of assembly) for both historic and practical reasons. C allows for a great amount of flexibility which loaders require, but consequently offers no memory safety guarantees. C provides a certain degree of type and scope checking, but loaders often treat regions of memory as sequences of bytes, not as typed objects, during loading and patching operations. This results in C type checking not being applicable to significant portions of critical operations a loader performs.

Loaders are typically memory-mapped to the very address space they are manipulating and must have a certain degree of self-awareness so that they do not corrupt their own code or data. This is often achieved by performing its loading operations relative to labels that describe absolute addresses. However, these labels only artificially exist within the confines of the typing system – they are typically typed as generic pointers (`void *`), and any warnings the C compiler may raise due to a loader's treatment of such pointers are inconsequential. The addresses of these labels and their relative positioning cannot always be resolved until *after* compilation since some addresses it needs are at the boundaries of the loader's own code or data. Loaders work with *compiled units of code and data as objects*, the very objects a compiler

generates. Any safety guarantees offered by the compiler are valid within the confines of these objects – software that operates outside these bounds is more-or-less left to its own devices.

It is useful to consider alternatives to standard C. However, with their safety guarantees comes restrictions that make it hard to implement a realistic loader. Nevertheless, loaders can potentially benefit from using safer string and memory allocation routines, similar to those proposed and implemented by Safe C [22], or additional libraries that enable run-time type checking such as librcunch [123]. Tools that analyze C source code to detect potential problems such as CLINT [111], Splint [76], MemSafe [193, 194], or Eau Claire [50], may also improve safety to a certain degree. It may be feasible to use an alternate C compiler such as Fail-Safe ANSI-C [167], MSCC [234], CETS [158], samurai [170], RTC [152], CCURED [9, 162]. Or we may use any of the techniques demonstrated in [68], which generate binaries with certain memory guarantees – ensured via static analysis and augmenting the binary with runtime monitoring. However, we may find such compiler-based tools enforce safety properties in a manner that is fundamentally incompatible with the needs of a loader. We can also consider using compiler extensions that introduce *control-flow integrity* [4], which guards against control-flow altering attacks by verifying the validity of any indirect control-flow transfers, a type of hardening that has been achieved via compiler extensions such as in [3, 25, 47, 206]. However, (1) loaders, by their very nature, transfer control to addresses/code that cannot be determined by runtime, (2) a number of attacks against loaders do not violate control-flow integrity, and (3) in loaders that enforce code signing, it may not always be possible to distinguish between control transfers into properly signature-checked images and all other images.

There are a variety of languages based on C, but with syntactical extensions and stronger type system such as those in Cyclone [109] and Deputy [57]. We can also consider using a safer systems language that is not directly derived from C, such a

Rust [180], Vault [66]. Filet-o-fish [64], Ivy [37], an extension of Haskell [69], PIT [171], mbeddr [216], or even IDRIS [33]. However, just as with safer C compilers, we will likely find a fundamental clash between the safer language's goals and guarantees and the loader implementation's own needs.

Compilers can also be augmented to provide other forms of targeted protections. For example, compilers have been used to inject code that checks for signs of stack corruption before jumping to a return address stored on the stack [62]. Researchers have also produced more general bounds-checking compiler plugins and static analysis tools, such as in [48, 56, 92, 159, 239].

Type systems, which are intuitive and lightweight techniques that help reduce programming bugs by capturing, modeling, and verifying program semantics, come in many forms and complexities, some of which verification of is undecidable. C has its own, fairly minimal, type system that does not provide much in terms of safety guarantees, but instead acts more like a warning system that helps developers identify certain classes of errors in their source code. *Dependent typing* schemes provide an expressive way in which to label source code with intent by allowing for types themselves to depend on their object's value. Dependently-typed languages include Agda, Idris, and Scala. These functional programming languages do not easily lend themselves to systems programming, although, researchers have demonstrated techniques to apply dependent types to low-level software [33, 57, 90]. Other forms of type checking have also been found useful in systems programming, such as the augmented C typing system Johnson and Wagner proposed that statically identify certain forms of kernel vulnerabilities [110], or type qualifiers which have been used to detect format string vulnerabilities in C programs [187].

Region-based memory management was first introduced as a theoretical foundation for dynamic memory management in Standard ML [207, 208]. In the context

of region-based memory management, a *region* is a collection of dynamically-allocated objects that can all be deallocated at the same time. Cyclone used such a memory allocation scheme, requiring each region to be annotated with a type, so that programmers annotate region-located object pointers with the region's type [89]. These typed regions allowed the compiler to differentiate between pointers in different locations (regions) of memory, such as the stack or heap, and make safety decisions based on these typed regions. Cyclone made use of typed regions in order to prevent dangling-pointer dereferences and memory leaks. Mozilla's Rust, a systems programming language that guarantees memory safety, also makes use of such techniques [151].

Domain-specific languages (DSLs) allow developers to quickly and succinctly express programs using semantics that make sense within a particular domain. For example, the MATLAB language and development environment is designed specifically for matrix and mathematical programming. There are also DSLs that make it easier to build correct and verifiable lexers and parsers which can be used to help a loader parse file-system metadata and binary files. Examples include PacketTypes (for generating packet parsers) [150], DataScript (for more general binary parsing) [14], Hammer [211], and Nail [16]. DSLs can be embedded in existing programming languages to take advantage of the language's syntax and features, or work as a stand-alone language. Some loaders implement and make use of their own DSLs, such as those supported by UEFI and U-Boot's consoles and Mach-O's loading bytecode (as discussed in section 2.5), and may be useful starting points in developing a DSL for loaders.

There is also an array of static analysis tools that exist outside the confines of a compiler. For example, KLEE [124], built on top of the LLVM compiler, is a powerful symbolic execution framework which has been used to automatically generate tests that achieve high code coverage [44], but also has been used for other purposes. Bazhaniuk et. al. used KLEE to test a UEFI-compliant firmware implementation for problematic memory accesses, namely for instances when a trusted SMM interrupt

reads memory outside the bounds of the SMRAM region [21]. Venkitaraman and Gupta used abstract interpretation-based analysis methods to determine if a given firmware image's implementation followed its coding standards [214]. More recently, a group of researches at UC Santa Barbara developed a static analysis technique to search for untrusted sources of non-volatile data that can compromise the devices bootloader in a manner that violates Google's Verified Boot guidelines [178, 215]. Static pointer analysis techniques, such as shape analysis, have been used to verify pointer safety in Windows and Linux device drivers [143, 237]. Yamaguchi, et. al., were able to identify previously unknown Linux kernel vulnerabilities using what they call a Code Property Graph which models a software's code and data dependencies [236]. Heelan, and Rooles demonstrated how to use SMT (satisfiability modulo theories) solvers to determine whether a piece of code adheres to various security properties. The HAVOC verifier (based on the Boogie theorem prover) can be used to check for various software properties, and has been used by the Microsoft security team to find variants of existing vulnerabilities [212]. Although these techniques can be used to extract interesting properties of a loader, any technique that does not include a low-level model of the loader's hardware, memory map, and instruction set, will likely fail to capture the loader's *raison d'être*. For example, standard static control-flow graphs cannot model a loader's self-modifications because they assume they implicitly assume that the software's instructions are immutable [8].

Loaders, which often contain self-modifying code, present their own special challenges to verification. In their *Certified Self-Modifying Code* paper, Cai, Shao, and Vaynberg present a Hoare-logic-style framework that supports verification of self-modifying code – modeling the program's own code as a mutable data structure [46]. They then demonstrate how their framework could be used to verify properties of toy examples of self-modifying software, including a simple BIOS-style bootloader (implemented in 11 lines of assembly code). This x86 bootloader, (1) prepares the x86

segment registers (memory addressing initialization), (2) prepares its registers to make a BIOS call to read the hard disk by loading static constants into these registers, and (3) jumps to the loaded target image in memory. They verify that this bootloader correctly sets its registers, while assuming that (1) their models are correct, (2) the BIOS interrupt is correctly implemented and copies the target to a location, (3) a specific disk geometry, and (4) the blocks of bootloader code are correctly loaded to a specific location in memory. Their verification framework includes (1) a model of x86 operating in 16-bit real mode as well as a limited set of x86 instructions, (2) a simplified model of a hard disk, as well as (3) a model of how a BIOS interrupt modifies the system. Their work shows the feasibility of formally verifying self-modifying code, but even verifying a simple bootloader requires a large amount of preliminary work to develop accurate and useful models. Most research in detecting, analyzing, modeling, and formalizing self-modifying code focuses on a more general form of self-modification – self modifications that result in semantic changes (e.g., for obfuscation) [8, 31, 46, 58, 142]. Loader self-modification is typically a relatively simple form of self-modification – *self-relocation*.

Formal analysis techniques have been successfully applied to systems software. Formal verification is typically used to assert the correctness of an implementation given a formal definition of correctness, and have been used against low-level software including compilers, networking software, and kernels. The CompCert compiler, which supports a subset of C, was formally verified to show that its generated machine code *behaves* like the source code from which it was compiled [136]. Guha, Reitblatt, and Foster implemented a machine-verified software-defined networking controller using the Coq proof assistant, proving that it adheres to a formal specification and operational model of software-defined networking. Researchers at Saarland University in Germany built a formally verified paging mechanism, by proving functional correctness of their page fault handler (written in a high-level language with inline

assembly) on a low-level model of the machine (disk and processor) [7]. The VeriFast verifier was used to prove the correctness of the implementation of module loading and unloading mechanisms for a toy kernel, verifying that they verified were that only loaded modules can be executed and only loaded modules can be unloaded [107].

The seL4 microkernel is a formally verified microkernel whose verification includes (1) a formal model the hardware which captures relevant kernel state and state transitions, (2) an executable specification of how the kernel works, including data structure and representation details, and (3) an abstract specification which is the main model of the kernel, and specifies interrupt and fault behavior, system call behavior, and the system call interfaces [125]. They proved that their kernel specification enforces integrity and confidentiality, assuming that (1) the embedded assembly code is correct, (2) the hardware behavior matches their model, and (3) their accompanying 1,200 lines of code that make up the kernel's bootloader are correct – more specifically, that the kernel has been correctly loaded into memory at set to a specific initial state [203].

ORIENTAIS is a formally verified real-time operating system used for automotive applications, which was formally proven to follow the OSEK/VDX standards set by the automobile industry [192]. This involved modeling the requirements set by these standards as pre and post-conditions using Hoare Logic. ORIENTAIS was written in C with annotations to aid verification and the VCC automatic verifier tool as used to perform a functional analysis of this code. In order to assure the kernel was free of deadlocks and termination errors, they created and verified a high-level behavioral model of the kernel's parallel tasks' interactions.

Temporal reasoning can also be useful to formally model and analyze loaders. Cook et. al. have shown how temporal reasoning could be used to prove temporal properties of databases, web servers, and kernels [59]. Temporal and linear logic have been used model and verify heap properties [53, 213, 235], and could be useful not

only for verifying a bootloader's own heap implementation, but for also modeling and verifying a loader's own temporal dependencies.

Both static and dynamic analysis techniques may be helpful in checking the correctness and safety of a loader. Valgrind is a popular instrumentation tool that can detect memory-related bugs during execution [163]. Flayer, which is built on top of valgrind, extends on the types of flaws that can be discovered, and also allows for fault injection [71]. Other dynamic memory error detecting tools include AddressSanitizer [185] and Memcheck [186]. The bug finding tool EXE uses a combination of dynamic and static analysis techniques to search for inputs that crash software [45]. IntFinder uses a combination of type and taint analysis to search for integer handling-related bugs, first using type analysis to select potentially problematic instructions and then applying taint analysis as it executes the software to verify whether it is a bug [49]. The Avatar framework was designed to instrument and test firmware for vulnerabilities via symbolic and dynamic analysis [238].

We can also apply less formal methods, such as fuzzing, to test loaders for memory corruption bugs. Fuzz testing, a technique introduced by Barton Miller [153], is a form of software testing in which invalid program inputs are chosen in attempt to crash the software. If a program crashes while processing such input, it signifies that the program is buggy, and likely vulnerable. In general, fuzz-testing research has greatly focused on software instrumentation and test case generation. Binary-image specific fuzz test case generation tools, such as Melkorr [95, 104], may be useful in testing a loader's durability and safety.

3.2.2 Runtime environment

A software's runtime environment can help protect against certain types of bugs from being exploited. Because a loader cannot necessarily rely on its environment and environment-based protection mechanisms to be in place, as it often the loader's job

to establish any environmental security mechanisms, we would need to understand the mechanism's bootstrapping process to determine whether it can be usefully applied to loaders. Several hardware primitives have been proposed that enable more efficient bounds-checking mechanisms, including hardbound [67], and watchdog [157]. SafeProc proposes a new series of ISA instructions that detect violations in memory references [86].

3.3 Final remarks

Loaders are susceptible to not only classic low-level memory corruption vulnerabilities, but also to high-level "weird-machine"-like issues. There are already many architectural solutions in place that can help proactively protect against low-level vulnerabilities, such as address space layout randomization (ASLR) and hardware-enforced memory read/write/execute protections. However, these protections are mainly *software-initialized* and thus we cannot depend on them to be in place for a loader. Compilers can perform static analyses and insert instrumentation to both proactively and retroactively protect again general classes of vulnerabilities, but loaders often function outside a compiler's sphere of influence. Other static analysis and formal techniques can be helpful in identifying potential vulnerabilities, as well as verifying certain properties of a loader. However, before we can use formal analyses to verify behavioral correctness of a loader's implementation, we must first model the behaviors and properties we want to test. I discuss a potential model in chapter 5. Only after we decide on the set of properties and behaviors to enforce can we consider applying a more complete and formal verification technique.

Security won't get better until tools for practical exploration of the attack surface are made available.

–Joshua Wright, 2011
"Wright's Principle"

4

A detour: bootloader instrumentation and analysis

At this point I will narrow my focus from speaking about loaders in general to speaking more specifically about bootloaders. In order to compose a model that describes loading behaviors central to security properties for verification, it is useful to study real bootloaders. Bootloaders operating in their natural habitat can provide us with insights into the messy ecosystem in which bootloaders exist. Real bootloaders provide insight into engineering/manufacturing constraints and practices, inconsistencies or implementation bugs, and implementation complexities that we would otherwise not discover. For this reason, I decided to study a popular bootloader implementation, U-Boot (also known as Das U-Boot) by instrumenting it, gathering data on its runtime behavior, designing a method of modeling bootloader behaviors for verification

Table 4.1: BBxM hardware information

Primary processor	TI am37x (ARM)
# documented registers	~113425
Size of on-chip ROM	112KB
Size of public on-chip ROM	64KB
Size of "secure" on-chip ROM	32KB
Total size of on-chip RAM	64KB
Total size of public on-chip RAM	2KB
Total size of "secure" on-chip RAM	62KB
Size of external RAM	512MB

(presented in chapter 5), building an instance of a model for my case study's target system along with an access control policy, and demonstrating it feasibility by building policy enforcement mechanisms. U-Boot is a well-established (first released in 1999 and still under active development) and highly-flexible open-source bootloader that works on a large variety of hardware and architectures, making it a good candidate for this case study.

More specifically, I studied U-Boot executing on the BeagleBoard xM (BBxM), an ARM-based development board with a TI am37x system on chip (SoC) containing an ARM processor. The TI am37x processor is effectively a member of TI's family of OMAP3 processors which have been used in a range of products from general development boards like the BBxM, to smartphones, to smart watches like the Motorola 360, to smartglasses like the Lumus SLEEK. Table 4.1 contains some basic information about the BBxM.

We must capture *any* potentially relevant loader behavior before distilling the desired properties and then enforcing them. Therefore, in this chapter I will introduce techniques currently used to instrument and debug bootloaders. I will then introduce the suite of instrumentation tools I specifically designed to study bootloader behavior. I use these tools to collect and analyze U-Boot's address write patterns in order to define substages, address region divisions, and their type corresponding labels from which I build a memory access control policy.

There is a general lack of usable, affordable, and accessible tools and that can be used to debug bootloaders. Bootloaders are especially challenging to debug, especially before the it configures an output (e.g., serial console) device. Furthermore, there can be multiple (sometimes distinct) address spaces a debugger needs to keep track of while the instrumented loader is executing: the loader's own address space and the loader's loadee's address space. These address spaces may potentially change over time, sometimes overlapping (e.g. when a loader is actively relocating an image), and symbols within each address space may appear, disappear, or be relocated over time, all of which standard debuggers do not automatically track.

4.1 Bootloader instrumentation techniques

There are two classes of bootloader debugging methods: bare metal and emulation-based. Bare metal debugging methods require special hardware support, and emulation-based debugging methods require a software model of the emulated hardware.

In this section I will discuss both hardware and emulation-based classes of bootloader debugging techniques as well as their strengths and weaknesses with regards to studying bootloader behavior.

4.1.1 Bare metal debugging

JTAG is a popular method of instrumenting software executing on bare metal as it has be broadly adopted among chip and hardware manufacturers. Although the term JTAG is often used to mean a hardware instrumentation technique, JTAG actually refers to the protocol with which the hardware is debugged. More specifically, JTAG is the lower levels of the protocol stack – the "application-layer" details of what can be done via the JTAG interface is determined by the hardware manufacturer.

Whether and how JTAG can be used to debug a bootloader varies – manufacturers can chose to disable JTAG or to not make specific hardware components "visible" from JTAG (e.g., registers). Also, because JTAG is implemented in hardware and thus may

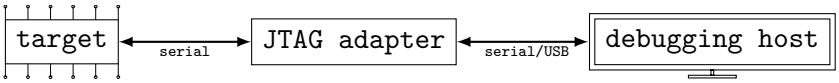


Figure 4.1: Typical JTAG debugging setup.

be affected by hardware initialization (such as software-controlled power settings), a JTAG debugger cannot always completely mediate a bootloader's execution even when the bootloader is not trying to hide parts of itself from JTAG.

The JTAG cat-and-mouse game. Through the eyes of manufacturers, JTAG is a bit of a double-edged sword. As much as it is a cheap method of allowing them to develop and debug their own hardware, it can also be used by anybody else in possession of the hardware to reverse engineer and/or modify the system. Over the years, chip manufacturers and OEMs have developed methods of limiting access to these hardware debugging interfaces with varying levels of success. Techniques ranging from "security by obscurity"/requiring expensive hardware and NDAs, to eFuse-based protection mechanisms, to tamper protection, have been implemented. ROM-based bootloaders are just one type of resource that manufacturers seek to both access and protect throughout this ongoing game of *J-TAG*[1].

U-Boot on the BeagleBoard xM case study notes: The *BBxM*'s bootloader can be debugged using a *Flyswatter 2* JTAG adapter driven over a USB/serial connection by *OpenOCD* (*Open On-Chip Debugger*) executing on the host. When working with debugging the BBxM's bootloader via JTAG, I encountered the following challenges:

- Single-stepping through `smi` instructions (or any instructions the belong to a secure monitor interrupt handler) causes further JTAG accesses to fail.
- Single-stepping through instructions that modify certain memory-mapped registers, such as those that control hardware clocks, causes JTAG to fail.
- OpenOCD caches the processor's register values and does not always provide

[1]Pun intended, my apologies.

up-to-date register values to the debugger.

- OpenOCD does not process debugging metadata and thus cannot always correctly determine whether a given instruction is ARM or Thumb-encoded.

4.1.2 Emulation-based debugging

It can be much easier to debug a bootloader running on software-emulated hardware because the emulator itself runs on a fully-booted system. Not all hardware have corresponding software emulator implementations, and even when one is available, it may not be complete or even accurate[2]. Even emulations that are based on the hardware's HDL (hardware description language) source code[3] from which the hardware itself is generated, tested, and verified, are not perfect representations of their hardware due to timing and other types of anomalies caused by the emulation's host machine.

Despite these downside of emulation-based debugging, this technique ultimately gives us more control over the software we are instrumenting. It allows us single step through any instruction (including `smi` instructions) without issue and with little JTAG-imposed overhead.

U-Boot on the BeagleBoard xM case study notes: QEMU has a software-based implementation of the *BBxM* which not only includes an emulation of its main processor (the am37x), but also the BBxM's peripherals which include its power management chip, external RAM, and the processor's GPIO (general purpose I/O) pins. Although this emulation is functional, it is also incomplete and at times inaccurate. For example, the very U-Boot bootloader that happily executes on bare-metal refuses to boot in QEMU's emulation because of how the emulated ROM stores

[2]This behavioral gap between bare-metal hardware and emulated/virtualized hardware is often called a "red pill" by researchers who are interested in determining whether code is running on a virtualized machine or on bare-metal.

[3]HDL source is typically not publicly available, although you can find sample HDL-base implementations of hardware on OpenCores at `http://opencores.org`.

the parameters it passes to the bootloader (this and my work-around is discussed in appendix A.2 on page 151). In general, I found emulation-based debugging to be both more reliable and more efficient, and thus it served as my primary instrumentation technique.

4.2 Dynamic bootloader analysis in practice

The true work a bootloader performs is expressed through its transformations of ts address spaces. Or, in other words, its executed sequences of write operations. Bootloader exploits often involve write operations performed at the wrong address and at the wrong time. My hypothesis is that it is both possible and useful to model a bootloader as a set of *temporal-based rules that determine whether a write operation at a given address is allowed* (see chapter 5 for more details). Therefore, the goals of my bootloader analysis tools are twofold:

1. Track location and relative order of every write operation, and
2. derive control flow-based substages to divide the bootloading process into phases where each has as consistent intended use of each region of memory it modifies.

These tools were essential in developing a useful and enforceable bootloader policy for a real-world bootloader, as I will later account in section 5.4.1 (page 102).

4.3 Dynamically tracking memory writes

An intuitive starting point in analyzing a loader is to study its memory write behaviors with the goal of determining what addresses in memory are modified (and when) during execution. There are two methods we can employ to trace memory writes of a bootloader running on emulated hardware: watchpoints and breakpoints.

4.3.1 Watchpoints

Watchpoints are a form of instrumentation that halt execution whenever the value at a particular address changes. Watchpoints most directly embody what I wish to track

– they halt execution when an address is written so I can collect information on the memory write. We can imagine scattering watchpoints across the entire address space so that we collect data on every memory write. Hardware-supported watchpoints are efficient but are limited in number, and software-based watchpoints can impose a large amount of overhead (linear to the number of watchpoints). Setting a watchpoint for every physical address imposes an incredibly high performance hit. Nevertheless, I modified QEMU so I can insert watchpoints at a page-like granularity, triggering data collection whenever an address within a particular block (a page-like unit of memory in QEMU's model of the emulated hardware's address space) is written.

U-Boot on the BeagleBoard xM case study notes: The BBxM has a 32-bit physical address space, supporting addresses up to 0xFFFFFFFF. However, not all of its physical addresses *are* writable. Therefore, it may be sufficient to only insert watchpoints for these writable address regions for information gathering purposes. For these purposes, I initially inserted a watchpoint for each 0x400-byte block in addresses 0x40200000-0x40210000 (on-chip RAM – 64 0x400-byte blocks) and 0x80000000-0xC0000000 (the address range mapped to external RAM, which is not initially available – 1,048,576 0x400-byte blocks). This regions do not include any memory-mapped registers. An additional 1,410,624 watchpoints would be required to instrument every writable block of physical memory including its memory-mapped registers which imposes a significant memory and runtime overhead. It takes on the order of one hour to execute the BBxM's U-Boot SPL stage with only RAM-backed watchpoints in place using this implementation.

4.3.2 Breakpoints

We can also make use of breakpoints to collect data on memory writes by inserting a breakpoint at *every* instruction that modifies memory, querying the machine's state when the breakpoint is hit in order to determine which address was modified. Although

setting large numbers of breakpoints can impose a significant initial overhead, there is not much of a performance hit during execution beyond that due to periodically in halting execution in response to the breakpoints.

To further reduce runtime overhead, we can identify places in the code where large consecutive blocks of memory are modified within a loop – what one may expect to occur during a `memcpy` operation – in such a way that we can pre-calculate the entire range of addresses written to by this loop *when the loop is first entered* and not need halt execution for subsequent iterations of the write instruction. I refer to these types of writes that occur in loops that contain a single write instruction and one conditional branch **longwrites**

All the information we need to determine which addresses need these breakpoints, as well as which registers need to be read to calculate the breakpoint instruction's write destination, can be determine by statically analyzing the bootloader's binary image. This information can be extracted, preprocessed, and saved into a database *before* the bootloader is executed (and reused for all future executions) in order to reduce the runtime overhead imposed by the instrumentation.

I have implemented this preprocessing using the `capstone` disassembler toolkit which I use to scan and disassemble each instruction in the image's binary to check if it is a write instruction. The only ARM/thumb instructions that modify memory are: `push`, `stm`, `str`, and `stl`.

For every write instruction contained in the binary, my tools insert the following information to a database:

- **Virtual address**: Address of instruction in memory (barring any relocation) as calculated from the executable's metadata
- **Register operands**: names of registers referenced in instruction's operands (including stack pointer if `push` instruction, or CPSR if instruction includes a condition code)

- **Write size**: number of bytes written by this instruction

U-Boot on the BeagleBoard xM case study notes: In order to trace *every* memory write performed by the BBxM's bootloader, I set a breakpoint at every `push`, `stm`, `str`, and `stl` instruction. I also precalculate the write destinations for four different *longwrites* performed by the bootloader, including one in the `memset()` function, one in `memcpy()`, and a basic block that zeroes out the BSS region of memory. The bootloader's BSS is 196,382 bytes in size but by treating the write instruction that zeroes out the BSS region as a *longwrite*, I am able collect information on each individual write within the loop via a single breakpoint, instead of the over 49,000 breakpoints that would have been triggered within the loop had I not employed this technique. My tool collects data on around 400,000 write operations generated from 1,596 breakpoints in just over 2 minutes (a significant improvement over the hour it takes when employing the watchpoint method) per successful boot in which its target is located on a FAT-formatted SD card. More statistics are shown in table 4.3.

4.3.3 Memory-mapped registers

A secondary bootloader-analysis tool I have developed parses all the tables in the BBxM processor's technical reference (the TI am37x processor [201]) that contain information on the processor's memory-mapped registers, in order to generate a complete database on these registers. For each register listed in the technical reference, my tool collects the following information:

- **Unique identifier**
- **Physical address**
- **Register width** (in bits)
- **Register permissions** (read-only, read/write)

With these data extracted from the documentation, I can test the bootloader to see if it tries to access any undefined registers or write to read-only registers. Although I

did not find instances of U-Boot modifying read-only registers, I did find what appears to be mistakes in the documentation (more on this in section A.1 on page 149).

4.3.4 Relocation

Bootloaders often shuffle objects around regions of memory which may include the bootloader's own data and/or code. We must be aware of these transformations to best understand the bootloader, especially when regions of memory that contain breakpoints are relocated (watchpoints in-of-themselves are agnostic of relocation as long as all addressable regions of memory are being watched). There is no standard way in which bootloaders perform and document such transformations. Therefore, obtaining a list of a bootloader's relocation operations is a manual process subject to human error.

For each relocated region, it is important to know the following:

- value of the program counter when it begins the relocation process,
- value of the program counter when the relocated region is ready to be used,
- physical address and size of the region that is relocated,
- physical addresses to which the region is copied, and
- (if available) offset of relocated region within binary image.

My tools use this information to keep track of *all* breakpoints or symbolic references, especially those that are copied or moved during execution.

Bootloaders typically relocate only a handful of regions, any given chunk of objects at most once, and typically in the style of a *longwrite*. Because of this, it is feasible to manually determine relocation information by searching dynamically collected memory write information for such *longwrite* patterns.

U-Boot on the BeagleBoard xM case study notes: The U-Boot boot sequence on the BBxM is composed of two U-Boot stages, (1) a simple stage called "SPL" (secondary program loader) that is small enough to be loaded by the BBxM's on-chip

Table 4.2: Relocations performed by U-Boot stages

Stage name	Function	# copied bytes	Purpose
SPL	`cpy_clk_code()`	264	Intends to move `go_to_speed` function into faster memory (see appendix A.3.1)
main	`cpy_clk_code()`	108	Intends to move `go_to_speed` function into faster memory (see appendix A.3.1)
main	`relocate_code()`	0x14402c	Moves `main` image towards end of RAM to make space for target image

boot ROM and loads (2) a second U-Boot stage I call "main" (which ultimately loads the Linux kernel). The SPL performs just one relocation of a region of code. The main stage performs two separate relocations: first a small region of code and then later *all* of the stage's own code and static data is relocated. It is possible to statically calculate all of the data needed to allow the instrumentation tools to keep track of all relocations at runtime. U-Boot's relocation phases are summarized in table 4.2.

4.4 Data collection and analysis

Throughout the course of execution, my instrumentation tools collect the following information about each memory write operation:

- **Write index**: Number of write operations performed before current operation
- **Relocation offset**: Offset of current program counter with respect to its expected memory address (as specified by its symbol definition)
- **Program counter**: Address of fetched write instruction
- **Link register**: Value of link register during operation
- **Current program status register**: Value of CPSR register during operation
- **General purpose register values**: The values of all other registers needed to compute destination of write

- **Substage**: Current substage (substages are defined and discussed in section 5.2.2 on page 81)

Using these collected data about each write we can later calculate the following about each write:

- **Destination**: Destination address of write operation
- **Size**: Number of bytes written by operation (negative if a `push`)

Pseudocode that calculates the instruction's write destination given the write instruction is shown in figure 4.2. Pseudocode to calculate the number of bytes a given instruction writes is shown in figure 4.3

4.4.1 Dynamic call graph generation

It is also useful to collect data on the bootloader's execution flow and generate a dynamic call graph. The DTrace tracing framework's `flowindent` plugin does exactly this, unfortunately, DTrace is a kernel-based instrumentation framework and therefore cannot be depended upon to instrument a bootloader. Therefore, I implemented a *gdb* debugger plugin called `calltrace` that can generate a bootloader's (or any software's) dynamic call graph.

The `calltrace` plugin works by inserting a special *function entrypoint* breakpoint at every named address (symbol) that falls within the bootloader's executable segments. When one of these *function entrypoints* is triggered, the plugin increments a global counter, logs the function name and counter value, and sets a special finish breakpoint which is triggered when the current stack frame is popped off the stack as the function returns. When the finish breakpoint is hit, it decrements the global counter and logs the event before disabling the finish breakpoint. This procedure produces a simple text representation of the bootloader's function calls and returns.

`calltrace` formats its output in a way that can be easily navigated and manipulated as a tree-structured outline with the `emacs` text editor, as can be seen in

```
def calculate_store_offset(instruction):
   lshift = 0
   disp = 0
   referenced_registers = register_values(instruction)
   for o in instruction.operands:
     if o.type == ARM_OP_MEM:
        lshift = o.mem.lshift
        disp = o.mem.disp
   if lshift > 0:
      regs[1] = (regs[1] << lshift) % (0xFFFFFFFF)
   return (sum(regs) + disp) % (0xFFFFFFFF)

# assume instruction is a memory store instruction,
# one of: push, stl, stm, or str
def register_values(instruction):
   regs []
   mne = instruction.mnemonic
   if mne.startswith("push"):
      regs = ["sp"]
   elif mne.startswith("stl") or mne.startswith("stm"):
      # similar to push, register is only in first operand
      regs = [instruction.operands[0].reg_name]
   else: # mnemonic is "str"
      for o in instruction.operands:
         if o.type == ARM_OP_MEM:
            if o.mem.base > 0:
               regs.append(instruction.reg_name(o.mem.base))
            if o.mem.index > 0:
               regs.append(instruction.reg_name(o.mem.index))
   return [get_reg_value(r) for r in regs]
```

Figure 4.2: Pseudocode used to calculate write destination at runtime for the currently fetched write instruction

```
def calculated_store_size(instruction):
   mne = instruction.mnemonic
   if mne.startswith("push") or mne.startswith("stl") or
    ↪  mne.startswith("stm"):
      (read_regs, write_regs) = instruction.regs_accesses()
      if "sp" in read_regs:
          read_regs.remove("sp")
      return -1*length(read_regs)*WORD_SIZE
   elif mne.startswith("str"):
     if ins.has_condition_suffix():
        # strip off conditional suffix
        mne = mne[:-2]
     size = mne[-1]
     if mne == "str":
        return WORD_SIZE
     elif size == "b":
        return 1 # byte
     elif size == "h":
        return WORD_SIZE/2  # half
     elif size == "d":
        return WORD_SIZE * 2  # double
  raise Exception("Cannot calculate number of bytes written")
```

Figure 4.3: Pseudocode to calculate number of bytes a written by given instruction

figure 4.4. Its output contains a list of all function calls and returns in the order they occur. Each function call is logged on a new line in the file as: (1) a series of asterisks representing the function's depth within the call stack, (2) a "$>$" symbol indicating the function was called, (3) the function's name, and (4) the location within the executable's source code from which the call was made. When the function returns, a new line is added to the log consisting of: (1) asterisks that represent the functions position in the call stack when it was called, (2) a "$<$" symbol to indicate the function is returning, and (3) the function's name.

My `calltrace` gdb plugin is a helpful tool in understanding U-Boot, or any other software's, runtime behavior and implementation, and I heavily relied on it identify semantically distinct phases in U-Boot's source code (which are discussed in section 5.5.1 on page 117).

4.5 Related firmware instrumentation work

There have been several academic papers published on firmware instrumentation that are worth noting here. The Avatar framework introduced an instrumentation technique that involved running the firmware inside a modified emulator that intercepts all I/O accesses, and forwards them to the physical hardware [238]. This technique allows us to execute firmware in a fully-instrumented environment without having to also emulate all of the device's peripherals in software. The SURROGATES system improved on Avatar's design, reducing the latency associated with I/O accesses so peripheral interactions operate closer to real-time.

4.6 Bootloader static analysis

The debugging methods I described in this chapter are helpful in understanding a bootloader's inner workings but they lack the vigor and completeness needed in order to make formally verifiable statements. Static analysis can be more challenging to

Figure 4.4: Example output generated from my `calltrace` tool rendered with `emacs`'s `outline-mode` syntax highlighting

perform on bootloaders than other types of software for many reasons including that they make use of hardware-specific assembly instructions, may alter registers that greatly affect system state (such as enabling page tables), and are often written in multiple languages (assembly and a "higher-level" language, like C). This means that in order for a static analysis tool to useful, it must work with a representation of the underlying hardware during analysis. Frama-C, a well established static analysis framework, is one such tool that is capable of performing static analysis on C source code while also making use of a simple model of the underlying hardware. It can perform a *value analysis* on C source code (hence the "C" in Frama-C) to determines all possible values a variable may take on, given its knowledge of C and the system's architecture. To give you ta sense of the inherent challenges of applying existing static analysis tools to bootloaders, I worked with Frama-C to produce a value analysis of the U-Boot SPL. Details on how I accomplished this are documented in appendix section C.1 on page 165.

Table 4.3: U-Boot instrumentation statistics at a glance

	SPL stage	**Main stage**
Total LoC[a] in U-Boot	~1,244,000	
LoC that form executable segments	~4,000	~25,066
Format of image	TI-defined	u-boot legacy uImage
Size of image	49,848 bytes	313,908 bytes
Number of symbols defined in ELF	1087	6235
Number of write instructions in ELF	1,592	7,283
Time to perform initial static analysis[b]	~2 mins	~6 mins
Execution time in QEMU	$<$ 1 sec	$<$ 4 secs
Execution time in QEMU instrumented with write-tracking breakpoints	~2 mins	~40 mins
Execution time in QEMU instrumented with watchpoints	~1 hour	4 days 13 hours
Frama-C analysis time	~8 mins	N/A
`calltrace` tool execution time	~1 min	~44 mins
Execution in QEMU instrumented with policy-enforcing breakpoints	~2 mins	N/A

[a] Lines of code

[b] Executing on an Intel Core i7-6600U CPU at 2.6 GHz machine with 16GB of RAM

Each of the linker's input files contains a set of segments of various types. Different kinds of segments are treated in different ways.

–John Levine
in "Linkers and Loaders" [139]

5

Access properties & region typing

With this strengthened understanding of the role loaders play in a system and how they may be implemented in practice (as discussed in chapter 2), we can now build a descriptive model of a loader's behaviors for the purpose of policy enforcement. My primary goal is to develop a policy mechanism that eliminates out-of-type writes to memory, so that, for example, a loader will not be able to accidentally (or intentionally) corrupt its internal bookkeeping state while it is loading a future stage's image, or corrupt an already loaded and measured (verified) image. Additionally, this technique should be able to enforce that certain tasks *occur* and occur in the *expected order* in order to prevent, for example, enforcement/validation failures. My system focuses on memory (address map) write accesses to create a type system governing memory writes that can be statically and/or dynamically checked to determine whether its memory writes obeys the type system's rules. This use of typed regions was inspired by Cyclone's (a type-safe derivative of C) region-based memory management [89]. However, instead on defining regions in terms of groups temporally-related objects (in terms of usage), my use of regions more directly corresponds to *sections* in an ELF file,

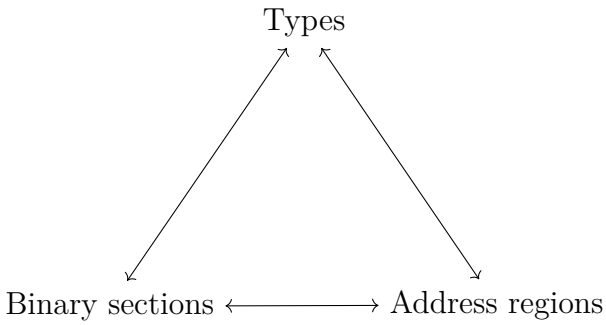


Figure 5.1: Sections in a binary, address regions in memory, and RBWAC types are all interdependent.

and is inspired by ELFbac's approach to constructing behavioral policies at the ABI level – by treating ELF sections (code, data, and metadata) as policy objects [17].

The loader policy proposed here is based on the observation that sections within a binary file are essentially types. All of the most well-known and popular binary file formats – Mach-O, PE, and ELF – have their own methods of labeling blocks of consecutive bytes containing *semantically related* objects that admit the same sets of operations (from the point of view of the binary toolchain).

Binary compilation toolchains prefer to co-locate semantically similar objects so their generated binaries are smaller, faster to parse, and easier to debug. Because of this design, runtime loaders do not have to themselves coalesce objects, and can instead directly copy sections into memory, which consequently creates memory regions containing the same semantically related objects. Therefore, a region of memory can also be thought of as a typed object, *and* the region's type is closely related to the type of the file's section from which it was derived.

If we develop a deeper understanding of how a loader's behaviors are driven by its target image – the image which can be thought of as containing the loader's *bytecode* – and how these operations correspond to the loader's higher-level semantics – which are sometimes explicitly defined (as discussed in section 2.5 on page 32), but are otherwise implicitly defined by the binary toolchain – we can more

meaningfully identify section and region types from which a type-based policy can be constructed.

In this chapter, I first define a simple system including its hardware and bootloader which I will use as a motivating example as I then present my typing system that captures the intentions of the bootloader's write operations. Throughout the first part of this chapter I will focus on a relatively simple policy for this system. I will then demonstrate the flexibility of this typing system by extending this simple policy into a more restrictive policy with more precisely defined semantics. Finally, I will document how this typing system can be applied to the real-world U-Boot bootloader of my case study. The purpose of working through these three examples is so that (1) we can initially get used to notation and simple properties, (2) we can work through non-trivial properties, which will finally allow us to (3) apply this typing system to a real product: the U-Boot SPL bootloading stage executing on a BeagleBoard-xM.

5.1 Mew-Boot on a ManulBoard: a toy system

For the purpose of building a simple illustrative example system for this type system, let us consider an imaginary piece of hardware based on the BeagleBoard called the ManulBoard. Similar to the BeagleBoard-xM, the ManulBoard contains a single-core 32-bit ARM-based system-on-a-chip (SoC) that is capable of being the basis of a mobile device.

5.1.1 ManulBoard hardware description

The ManulBoard system's hardware includes the following components:

- an ARM-based SoC,
- a serial interface (for console input and output),
- read-only non-volatile memory (ROM) containing a Mew-Boot bootloader image mapped to an region of addresses starting at 0x00000000,

- external non-volatile memory formatted with a filesystem that contains multiple images from which the subsequent stage can be selected (via the serial console),
- volatile memory (RAM) that is *not* initially memory mapped, and
- various registers (including those that must be used to configure the volatile memory) mapped to a region starting at 0x60000000.

5.1.2 Initial ManulBoard memory layout

Figure 5.2a illustrates the layout of the ManulBoard's initial memory map as the first instruction of its kickoff (ROM-based) bootloader (Mew-Boot) is fetched. Its address space eventually evolves to look more like figure 5.2b once it initializes the hardware. After the target image is loaded, the ManulBoard's memory map will like one illustrated in figure 5.2c.

5.1.3 Mew-Boot bootloader description

Mew-Boot ultimately invokes some target executable, but before it does so, it must:

1. initialize the serial console so that it can ask the user to select the target,
2. initialize its volatile memory to create space memory for a stack, bookkeeping data (which may be statically or dynamically defined), and space for the target image to be loaded,
3. copy statically-defined data in the bootloader's ROM-based image into a RAM-backed address region so that the data are writable,
4. initialize non-volatile memory (storage) and search it for potential target images,
5. prompt the user to select which target image to load , and finally,
6. load the target image into an unused RAM-backed region.

When Mew-Boot is ready to jump into its target, the target software expects the memory map to be as shown in figure 5.2c.

It does not matter to us what overall role the target partakes in the system (nor does it matter to the bootloader), as long as the bootloader is able to setup the

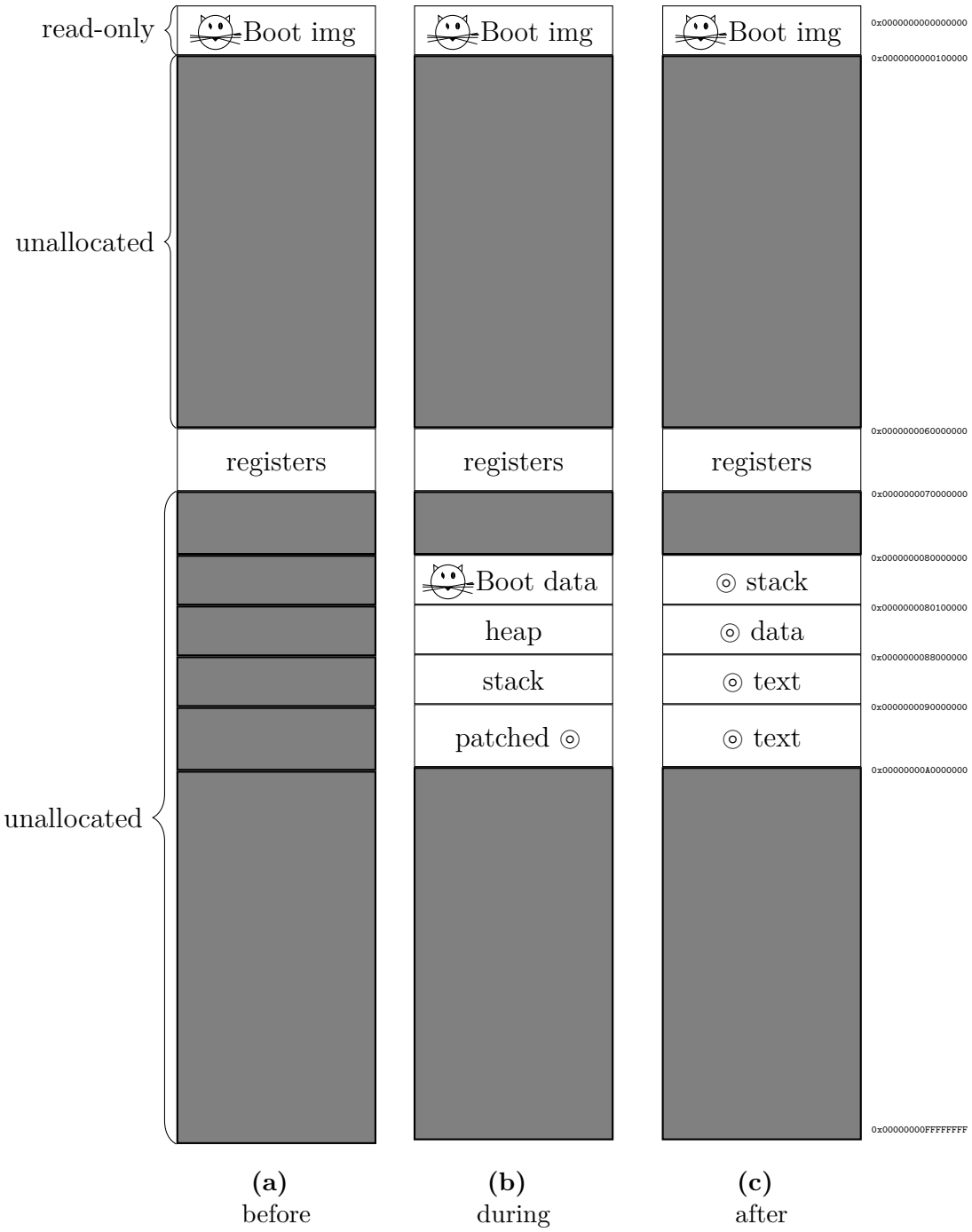


Figure 5.2: ManulBoard memory maps before (a), during (b), and after (c) MewBoot (Boot) execution while loading the target (◎) image

system so its target can be executed and its target can query the environment for any additional information it needs. Mew-Boot's target image could contain another bootloader stage, a kernel, or even a standalone application. Nevertheless, the target must have *some* knowledge of the memory regions and their boundaries within its environment including the read/write semantics of these regions. Neither the target nor the loader can afford to be sloppy – sloppiness leads to vulnerabilities.

Even this terse description of the ManualBoard and Mew-Boot gives us good sense of how the system's memory map evolves as blocks of memory become used, abandoned and re-purposed throughout execution[1]. At any point in time, the ManulBoard's memory map can be decomposed into a series of non-overlapping regions, each of which can be assigned a label that represents the region's intended use.

This description of Mew-Boot also allows us to identify actions that demarcate major transitions in address space usage. From it we can subdivide the bootloading process into a series of substages so that each contain (at some granularity) a semantically-unchanging address space. Each substage can then be provided with a label that identifies its purpose based on how it intends to interact with its address space. By decomposing the bootloading process into such phases, we can then model the bootloader as an automaton of phases, each of which contains a set of semantically-labeled regions and a set of behaviors we expect the phase to exhibit. The bootloader traverses though these substages in some pre-determined manner over the course of a successful boot.

These semantic labels are the basis on which our typed regions are defined and they naturally align themselves with how memory maps are typically modeled. For example, the memory map in figure 5.2a which simply illustrates the device's

[1] Any *pre-repurposed* use of a reused block is likely a vulnerability, very much in the same vein as a *use-after-free* vulnerability.

physical memory layout when reset already has semantically labeled regions – the bootloader's code is in the region mapped to the lowest addresses, and a region starting at 0x60000000 contains its memory-mapped registers. Later in the ManulBoard boot process (depicted in figure 5.2b), new regions are conjured for Mew-Boot's static data, the heap, the stack, and the target image. Even though the labels I just mentioned are fairly general, we can still use them to define a *meaningful* intent-based typed-region-based access control policy, one that ensures that, e.g., the heap is not used until it is available, that the target image is not executed until the image is ready, and that any repurposed regions are not accessed by operations that are only aware of its original purpose.

5.1.4 ManulBoard/Mew-Boot vs. BeagleBoard-xM/U-Boot

This toy system composted of the ManulBoard and Mew-Boot is intended to be a simple, but realistic, system that highlights a subset of policy-relevant behaviors exhibited by U-Boot executing on a BeagleBoard-xM. In particular, Mew-Boot, like U-Boot, must initialize hardware *before* loading and patching a target. Also, both systems have regions of RAM-backed memory that are used but are *not initially available* when the bootloader is first invoked. Although U-Boot on the BBxM does not have data stored on read-only memory that must be relocated to writable memory, it does have to manually manage data located in the `.bss` section – global data whose values are (expected to be) initially zero – and thus it should not access these data until it *explicitly* zeros out the region of memory for them.

5.2 Address region-based write access control

Bootloaders must initialize and manage system resources as they locate, load, and patch their subsequent target stage. They perform these actions in a sequential manner shaped and driven by its hardware and software resource dependencies. Periods of execution which encapsulate different phases of intended address-space use can be

thought of as distinct *substages*, each of which may initialize resources, load (copy) objects into a region, or patch a region for a future substage. Based on this observation, we can assign a type to each such substage in a manner that allows us to *label the intent of each its write operations* and *mediate each write* (either statically or dynamically) that takes place during the substage, ensuring that the write's destination falls within a compatible region based on the region and substage types. I call this method of modeling and restricting a loader's behavior RBWAC[2], address **R**egion-**B**ased **W**rite **A**ccess **C**ontrol. Without loss of generality, I limit the focus of my thesis to only mediating write accesses, however, these ideas can be extended to incorporate any number of memory use-related operations. The remainder of this section describes the various components that make up the most general form of RBWAC: its formal definition, temporal logic (which governs substage typing), region typing, and access control rules. We will then describe RBWAC$^{\mu}$, a class of RBWAC policies for loaders, and finally describe a couple of example RBWAC$^{\mu}$ instances.

5.2.1 RBWAC definition

A complete RBWAC policy instance is simply defined with the following tuple:

$$(S, R, O, b, e, \mathbb{S}, \mathbb{E}, \mathbb{F}, \mathbb{Q}, \mathbb{R}, \texttt{used\_regions}, L, U, \texttt{semantics\_of},$$
$$\texttt{typeof\_region}, \texttt{typeof\_substage}, \mathbb{P})$$

Where S is the set of substage types,

R is the set of region types,

O is the set of mediated operations,

b and e define the minimum and maximum addresses supported by this policy (non-negative integers),

$\mathbb{S} = \{s_0, s_1, \ldots s_{n-1}\}$ is the set of n substages,

$\mathbb{E} \subseteq \mathbb{S}$ is the set of entrypoints,

[2]Pronounced: ARRRR...whack, the B is silent!

$\mathbb{F} \subseteq \mathbb{S}$ is the set of allowed exitpoints, denoting successful execution upon entry,
$\mathbb{Q}$ is a function that defines allowed substage transitions such that $\mathbb{Q} : \mathbb{S} \rightarrow \mathcal{P}(\mathbb{S})$ (range is the power set of $\mathbb{S}$),
$\mathbb{R}$ is the set of substage region definitions, $\mathbb{R} = \{r_0, r_1, \ldots, r_m\}$ where each $r_i \in \mathbb{R}$ defines a region of consecutive bytes, i.e., $r_i = (b_i, e_i)$ where $b \leq b_i < e_i \leq e$,
`used_regions`() is a function that returns which regions define the address space's labeling during a particular substage where: $\texttt{used\_regions} : \mathbb{S} \rightarrow \mathcal{P}(\mathbb{R})$.

Region definitions must adhere to certain restrictions. For each $s_i \in \mathbb{S}$ and A_i where $\texttt{used\_regions}(s_i) = A_i$, A_i must be **complete** and **non-overlapping**.
The definition of **complete** is:
for all valid addresses a (i.e., $b \leq a < e$) and $\forall s^* \in \mathbb{S}$,
$\exists r^* \in \texttt{used\_regions}(s^*)$ such that $r^* = (b^*, e^*) \ \wedge \ b^* \leq a < e^*$.
Non-overlapping is defined as:
for some valid address a, $r_i = (b_i, e_i)$, and $r_j = (b_j, e_j)$, where $r_i, r_j \in \texttt{used\_regions}(s^*)$,
$(b_i \leq a < e_i) \ \wedge \ (b_j \leq a < e_j) \quad \Leftrightarrow \quad i = j$.
Additionally, I say that a region $r \in \mathbb{R}$ is **in-scope** for some substage $s \in \mathbb{S}$ if $r \in \texttt{used\_regions}(s)$. This is useful because $\forall s \in \mathbb{S} : \texttt{used\_regions}(s) \subseteq \mathbb{R}$.

To help formalize the notion of a region's semantics, we also define two sets of labels, L and U, as well as a few functions that operate on these sets:
L is a set of semantic labels which can define a region's intended use (e.g., heap, stack, executable),
U is the set of usage labels, which can define a region's semantic status (e.g., reserved, ready, in-use).
`semantics_of` is a function that returns a region's semantics during a given substage i.e., $\texttt{semantics\_of} : (\mathbb{R} \times \mathbb{S}) \rightarrow \mathcal{P}(U \times L)$. For example, if region r^* solely contains

the current stack during substage s^*, then `semantics_of`$(r^*, s^*) = \{$(stack, in-use)$\}$.
`typeof_region` is a function that returns the type of a region *with respect to* a substage (during substage execution) i.e., `typeof_region` : $(\mathbb{R} \times \mathbb{S}) \rightarrow R$.
`typeof_substage` is a function that returns the type of a substage,
`typeof_substage` : $\mathbb{S} \rightarrow S \cup \{success, failure\}$.

Substages that are typed as either *success* or *failure* are special substages in that once one of either type is entered, policy enforcement is discontinued. As their names suggest, a *success* substage denotes the target was successfully loaded, and a *failure* substage indicates that loading failed.

An RBWAC instance's access rules are formally defined by a policy $\mathbb{P}$. $\mathbb{P}$ is a function that defines the policy – $\mathbb{P} : (\mathbb{S} \times \mathbb{R} \times O) \rightarrow \{\text{deny}, \text{allow}\}$. It *must* be the case that for any $r^* \in \mathbb{R}$, $s^* \in \mathbb{S}$, and $o^* \in O$ where
`semantics_of`$(r^*, s^*) = \emptyset$: $\mathbb{P}(s^*, r^*, o^*) = \text{deny}$.

I have defined additional RBWAC-specific terminology that I will use throughout this chapter. A **mediated event**[3] is a tuple (s^*, r^*, o^*) representing an operation that occurred during execution, where $s^* \in \mathbb{S}$, $r^* \in \mathbb{R}$, and $o^* \in \mathbb{O}$. During a mediated execution, the **current substage** is defined as the substage that was most recently entered. I define an **invocation** of a loader to be an ordered list, $\mathbb{I}$, of all substages that are entered during a mediated execution.

This thesis focuses on policies constructed from a less general form of RBWAC, a specific RBWAC **class** for loaders called RBWAC$^{\mu}$ where

[3]The idea of mediated events is based on Schnieder's concept of "complete mediation" [183].

$|S^\mu| = 3$,

$|R^\mu| = 6$,

$O^\mu = \{\texttt{write}\}$,

$b^\mu = 0$, $e^\mu = 2^{32}$,

$\mathbb{E}^\mu = s_0$,

$\mathbb{F}^\mu = \{s_n\}$ where $n < |\mathbb{S}|$,

$\mathbb{Q}^\mu(s_i) = s_{i+1}$ for $i < n$.

In general, we define a **RBWAC class** to be a partially instantiated RBWAC policy with *at least* the following components defined: $(S, R, O, b, e, \mathbb{P})$. An **RBWAC instance** is a fully-defined RBWAC policy.

5.2.2 Substages

Substages are, in essence, similar to typestate as proposed by Strom and Yemini in 1986 [199], except at a far coarser granularity. With typestate, the operations an object of some type supports are dependent on the object's state – its *typestate*. Substages dictate the system's state and state transitions. Whether a particular operation may be applied to a region is not only dependent on the region's type, but also on the type of the current substage.

A single stage of the (boot) loading process can be subdivided by purpose into a sequence of consecutive *substages*, each of which is a policy *principle* whose type determines which access rules are in place. The substage's type should reflect the substage's *intent*, be that internal loader bookkeeping (which includes hardware initialization), or preparing an address region of a future substage. With my ELF metadata-driven weird machine [191] as inspiration, I focus on defining types that indicate a phase's overall function in the bootloading process: *loading*, *patching*, or *bookkeeping*, as defined in table 5.1. I have found that these choices in substage types

Table 5.1: RBWAC$^{\mu}$ substage type definitions (S^{μ})

name	symbol	semantic description
loading	$\boldsymbol{\delta}$	"loading" (e.g., copying) a future *substage*
patching	$\boldsymbol{\pi}$	patching/linking a loaded future substage[a]
bookkeeping	$\boldsymbol{\beta}$	not performing loading or patching

[a] Although it is possible to unite the *patching* and *loading* types without weakening the RBWAC$^{\mu}$ policy, these two behaviors are semantically different operations.

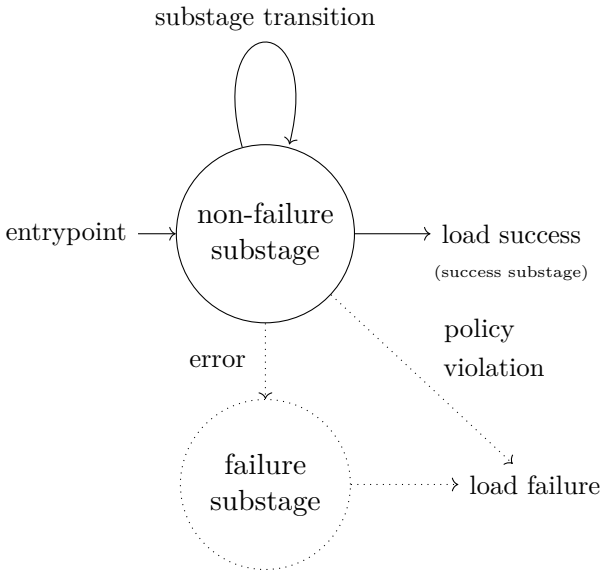


Figure 5.3: Success, failure, and sequential substage transitions of RBWAC$^{\mu}$ policy. Policy is enforced until *success* substage is entered. If policy is violated (either due to an out-of-order substage transition or a type violation), then loading fails.

to be sufficient and pose minimal challenges when I applied RBWAC$^{\mu}$ to an existing codebase.

Given that a significant portion of loading vulnerabilities (as discussed in chapter 2) have a temporal component (e.g., code not being validated before being loaded), it is useful to build a policy mechanism around these substages that enforces temporal dependencies. Such rules are represented by the function $\mathbb{Q}$ (allowed substage transitions) in the RBWAC formalism. As a starting point, I decided to only focus on policies with fixed linear substage ordering.

The idea of requiring a fixed linear ordering of substages may seem fairly restrictive at first but, in fact, allows us to simply model the sequence of tasks a bootloader (or even an application loader) performs to successfully loads its target. It also gives us the added benefit of being able to uniquely identify which substage a loader is in by inspecting its call graph. This is especially useful when working with static analysis tools such as Frama-C. It also forces the loader to be written in a style that makes it easier to statically determine whether a function that makes assumptions about currently defined regions is invoked *only* when these properties hold (assuming we can assert exactly which substages exhibit such properties) – partially motivating this design decision. Other substage transition models are not discussed in this thesis[4].

5.2.3 A short detour: design patterns and substage transitions

This linear substage ordering is fairly straightforward to achieve in practice: each substage in the sequence is equivalent to an entrypoint of a function that behaves like a forward continuation which never returns[5]. Such a continuation-passing style of programming is often implicitly present in loader implementations, used when a loader finishes initializing some feature upon which all later phases of the loading process depend. Rarely do such features later become unavailable.

Substage entrypoints have additional dedicated semantics in comparison to regular function entities because they allow for regions of memory to be retyped upon transition. Defining substage entrypoints as non-returning continuations helps us more simply model such semantics.

Continuation passing. Figure 5.4 shows an example continuation passing design pattern already present in the U-Boot bootloader source code– in this case, after

[4]Future work can explore different models of allowed substage transitions such as lattices.

[5]Although it is not strictly necessary to use non-returning continuations to implement linear substages, e.g., a global variable can be used to keep track of the current substage, I found non-returning continuations helpful when working with static analysis tools.

it finishes relocating a region of code with its call to `cpy_clk_code` (on line 6), it performs a forward call to the `s_init` continuation (on line 10). Code within the phase that begins upon entry of `s_init` assumes that `cpy_clk_code` was executed.

Trampoline-style continuation passing. This continuation-passing programming style is also often mixed with a more imperative/procedural programming style where a `main()`-like function sequentially calls a series of functions, each of which eventually returns to the initial caller/trampoline (such as `main()`). An example of where U-Boot uses this design pattern is shown in figure 5.5. In instances when one of these procedures is a substage entrypoint, converting the call into a non-returning continuation is a relatively trivial (and potentially automatable) task.

An array of function pointers. Another design pattern U-Boot makes use of when performing an ordered sequence of tasks is *iteration over an array* – sequentially calling each function pointer in a list. One example of this is shown in figure 5.6, where `initcall_run_list` iterates over the function pointers in the array passed to it named `init_sequence`. If we decide that one of the function pointers in the array is a substage entrypoint, we can transform it into a non-returning continuation by dividing the list of function pointers into two – one containing all function pointers before the entrypoint and the entrypoint itself, the second containing the remaining function pointers. This design pattern, as well as the `main()`-style pattern, largely occurs naturally in bootloaders; indeed, U-Boot only needs limited adjustment to deliberately introduce non-returning continuations that are invoked using such a design pattern.

Lazy device initialization. Bootloaders also occasionally perform lazy device initialization – only initializing a device when it is first used. For example, U-Boot does not initialize data structures related to its SD (also referred to as MMC) card reader until the first time it accesses the SD storage. Such a design pattern makes separating bootloading phases, such as bookkeeping from loading, not as

```
ENTRY(lowlevel_init)
    ldr         sp, SRAM_STACK
    str         ip, [sp]        # stash ip register
    mov         ip, lr                # save link reg across call
    ldr         r1, =SRAM_CLK_CODE
    bl         cpy_clk_code
    mov         lr, ip                # restore link reg
    ldr         ip, [sp]        # restore save ip
    # tail-call s_init to setup pll, mux, memory
    b         s_init
ENDPROC(lowlevel_init)
```

Code fragment based on U-Boot's definition of `lowlevel_init` in arch/arm/cpu/armv7/omap3/lowlevel_init.S

Figure 5.4: Example of continuation-passing in U-Boot source code

```
void s_init(void) {
  watchdog_init();
  try_unlock_memory();
  omap3_invalidate_l2_cache_secure();
  set_muxconf_regs();
  prcm_init();
  per_clocks_enable();
}
```

Code fragment based on U-Boot's definition of `s_init` in arch/arm/cpu/armv7/omap3/board.c

Figure 5.5: Trampoline design pattern in U-Boot source code

straightforward, however, I did not find U-Boot's use of lazy device initialization posed problems when defining substages.

```c
int initcall_run_list(const init_fnc_t init_sequence[]) {
  const init_fnc_t *init_fnc_ptr;
  for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
    unsigned long reloc_ofs = 0;
    int ret;
    if (gd->flags & GD_FLG_RELOC)
      reloc_ofs = gd->reloc_off;
    ret = (*init_fnc_ptr)();
    if (ret) {
      printf("initcall sequence %p failed at call %p (err=%d)\n",
        init_sequence, (char *)*init_fnc_ptr - reloc_ofs, ret);
        return -1;
    }
  }
  return 0;
}
```

Code fragment based on U-Boot's definition of `initcall_run_list` in common/initcall.c

Figure 5.6: Example use of arrays of function pointers in U-Boot source code

The Mew-Boot bootloader on the ManulBoard can be decomposed into substages in a variety of ways. I will begin by describing a simple RBWAC instance that consists of the following five substages:

- Substage 0: [`_start`: type $\boldsymbol{\beta}$, *bookkeeping* (defined in table 5.1 on page 82)] Initialize serial console, volatile memory, and stack
- Substage 1: [`copy_data`: $\boldsymbol{\beta}$] Copy static data and initialize heap
- Substage 2: [`load_target`: $\boldsymbol{\delta}$ (*loading*)] Copy selected target image to volatile memory, and patch to reflect its current position in memory
- Substage 3: [`jump_to_image`: *success*] Jump to target image's entrypoint
- Substage 4: [`halt`: *failure*] Mew-Boot encountered an error from which it cannot recover

Although simple, this policy instance can be significantly extended, which I describe in section 5.3.2.

5.2.4 Substage transitions

RBWAC$^{\mu}$'s set of substage types naturally align with function boundaries because of the the close relationship between the source code's *unit structure* and *intent*. RBWAC$^{\mu}$'s substages are meant to model the loader's linear sequence phases when it successfully loads and executes its target. A *substage transition* occurs when an entrypoint to the subsequent substage is executed. *Every* write the loader performs is measured against the policy as dictated by the type of the most recently-entered substage, i.e., the **current substage**.

An instance of this RBWAC$^{\mu}$ class must define which functions mark substage entrypoints ($\mathbb{S}$) and allowed substage transitions ($\mathbb{Q}$). When $\mathbb{Q}$ requires a fixed linear sequence (which I will assume to be the case throughout the rest of this chapter), any out-of-order substage transition is considered a *policy violation*. More formally stated: Given the function $\mathbb{Q}^{\mu}$, which defines allowed substage transitions, and the set $\mathbb{A} \subseteq \mathbb{S}$ which contains the $|\mathbb{A}| = m$ substages that are allowed to be entered (are in the range of $\mathbb{Q}^{\mu}$), such that $\mathbb{A} = \{s_0, s_1, \ldots, s_{m-1}\}$, then $\mathbb{Q}^{\mu}(s_i) = (s_{i+1}) \quad \forall i < m$. Any transition into a substage $s^* \notin \mathbb{A}$ is considered to be a **policy violation**. An *invocation* $\mathbb{I}$ is defined as **successful** if the last substage in its sequence is in $\mathbb{F}$.

We can easily model failure paths in loader execution (e.g., halting) by defining a substage $s_f \in \mathbb{S}$, such that $\forall s^* \in \mathbb{S},\ s_f \notin \mathbb{Q}^{\mu}(s^*)$.

The final substage s_{m-1} in the sequence defined by $\mathbb{Q}^{\mu}$ is considered to be a special **success** substage (e.g., $\mathbb{F} = \{s_{m-1}\}$), which upon entry indicates the loader has completed execution and that its policy has been successfully and completely enforced. This final substage should be an entrypoint to a short function that merely jumps to the target's entrypoint.

The first substage in the substage sequence, s_0, should be the loader's own entrypoint.

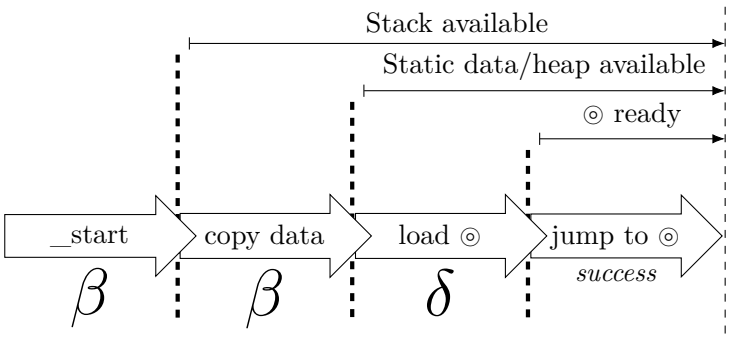


Figure 5.7: Substage sequence defined by the simple ManulBoard Mew-Boot policy. Each arrow contains a substage name and points to the policy's subsequent substage. The label below each substage's arrow denotes the substage's type. Important features common to one or more substage are indicated above the substage arrows. The ◎ symbol is shorthand for *target*.

The ManulBoard example policy has four *non-failure* substages – s_0–s_3 (the union of the domain and range of $\mathbb{Q}^\mu$) – which are included in the policy's substage sequence, as well as one failure substage – s_4 – which is entered upon a call to `halt()`. Figure 5.7 depicts these substages, their types, and also indicate during which substages important resources, such as the stack, are available.

When the ManulBoard's bootloader is invoked and s_0 is entered, there is no writable memory (RAM) available until the bootloader initializes it by writing to a particular memory-mapped register. Likewise, the stack isn't available until the RAM is initialized *and* the stack pointer register is set properly, which happens just before `copy_data` (s_1) is invoked. During the `copy_data` (s_1) bookkeeping substage, the static bootloader data are copied to RAM and the bootloader's stack is initialized so they are ready by the time `load_target` (load ◎, s_2) is invoked. The `load_target` *loading* substage (s_2) should be the *only* substage that writes to the region of reserved for the target image. Finally, the target image *must* be fully prepared for execution by the time `jump_to_image` (s_3) is invoked.

Table 5.2: RBWAC$^{\mu}$ region types (R^{μ})

name	symbol	semantic description
READ-ONLY	**r**	read-only
STACK	**s**	region containing stack
BOOKKEEPING	**b**	loader's internal state
GLOBALS	**g**	non-stack regions writable by all substage types
FUTURE	**f**	region reserved for future substage image
PATCHING	**p**	region containing future substage image

5.2.5 Region typing

RBWAC assigns types to regions of memory containing related objects to explicitly indicate its intended use. RBWAC$^{\mu}$'s six region types are: READ-ONLY (**r**), STACK (**s**), BOOKKEEPING (**b**), GLOBALS (**g**), FUTURE (**f**), PATCHING (**p**), each are defined with more detail in table 5.2. Therefore, the set of region defined in RBWAC$^{\mu}$ is,

$$R^{\mu} = \{\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{b}, \boldsymbol{g}, \boldsymbol{f}, \boldsymbol{p}\}.$$

Regions require us to explicitly specify the *intended use* of portions of the address space accessed during a given substage. Because regions are defined orthogonally to the bootloader itself, it is possible to *statically* check that all *in-scope* ($r \in$`used_regions`(s)) regions during a particular substage are *complete* and *non-overlapping* (as defined in section 5.2.1 on page 78). In the context of this thesis, any address/region that I do not explicitly label with a type is assumed to be READ-ONLY.

By virtue of the *locality of reference* principle, it is fairly convenient to type memory addresses in this manner, as nearby addresses typically have a similar intended use. This allows specialization to more explicitly be part of programmer's mental model and consequently expressed by the policy.

Regions allow for a highly flexible type and policy granularity. A single object (such as a variable or memory-mapped register) can either be assigned its own region and typed differently than its surrounding data, or incorporated into a neighboring region. Therefore, for example, a memory-mapped register that controls

a specific subsystem can be treated differently from other nearby memory-mapped registers. Therefore, an RBWAC policy can seamlessly mediate subsystem/external hardware accesses, allowing us to control access to memory-mapped I/O as well as all other objects in memory in a *centralized* and *consistent* manner. This is a compelling alternative to the variety of access control configuration methods implemented by different chip manufacturers, which include specialized instructions (e.g., `in` and `out` in x86-based architectures) or special registers (accessed via normal load and store memory operations), on top of which layers of address translation may be applied (if an IOMMU is in use, such as Intel's VT-d). Bootloader are *expected* to properly configure these access controls, which are not always straightforward to configure. There are numerous examples of bootloaders not correctly configuring these controls (such as in [178]) resulting in a vulnerable system. RBWAC allows bootloader developers to *formalize* and *validate* such expectations set by hardware manufacturers.

An address's semantic use may evolve over time, due to the very nature of loaders. Thus, our type system we must be able to allow for such region layout transformations in a dynamic but controlled fashion, so that region types reflect the programmer's expectations for the current substage throughout the entire bootloading process. Therefore, for each defined substage, there must also be a set of *complete* and *non-overlapping* regions defined specific to that substage. This plays out as a controlled transformation of regions from the point-of-view of the memory map, – regions definitions may evolve, **but only during substage transitions**.

Engineers typically think of hardware in terms of memory maps, such as the one illustrated in figure 5.2b (page 75), which cleanly the purposes of various memory regions. I use memory map diagrams in place of formal region definitions throughout this chapter because they plainly represent an ordered set of non-overlapping regions.

Region definitions for each of Mew-Boot's substages, is illustrated in figure 5.8. For the purpose of clarity, region subdivisions remain constant for all substages, only a region's type may differ between substages. (I will model region definitions for all example policies I present in this chapter in this manner.) If a region's type differs between consecutive substages, then the type conversion occurs *during* the substage transition *before* the subsequent substage begins execution.

The ManulBoard's initial substage's region definitions, for the *bookkeeping* substage s_0 (_start), is illustrated by the memory map in figure 5.8a. The only region the bootloader *can and should be able to write* during the first substage is the region containing memory-mapped registers, which is labeled as a BOOKKEEPING region named r_2 in figure 5.8a. The BOOKKEEPING type is used here to indicate that these registers should only be modified when configuring the system. This substage's region definitions (which happens to only contain the one BOOKKEEPING region, with the remaining addresses typed as READ-ONLY) represents the bootloader's *intended* memory usage during this substage, as it sets up the RAM and stack.

Figure 5.8b shows the regions defined for the `copy_data` (s_1) substage. As the `copy_data` substage is entered, the **stack** region (r_6), which was initialized by the previous substage, becomes available. Also, a portion of RAM reserved for the target image (region r_7) is typed as a FUTURE region as this substage is entered. During this *bookkeeping* substage, now that the RAM is available, static data are copied from the ROM to the RAM (into the r_4 BOOKKEEPING region), and the heap (BOOKKEEPING region r_5) is initialized.

When the `load_target` (s_2) *loading* substage is entered, static data in r_4 (which have initialized by the previous substage), maintaining their BOOKKEEPING type, and the recently-initialized heap becomes available as a GLOBAL region so that it can be used while the target is loaded. These region definitions are depicted in figure 5.8c. The target's image must be ready for execution by the time the s_3 (`jump_to_target`)

success substage is entered, at which time RBWAC mediation ends and control is transferred to the target.

Bootloading failures that are explicitly defined in source code, which are implemented as non-exiting loops in the U-Boot SPL, should be modeled as *failure* substages. This is achieved by defining a substage at the entrypoint to the failure's function that is not included in either the domain or the range of $\mathbb{Q}^{\mu}$ (the function defining allowed substage transitions). In the case of Mew-Boot (as well as in U-Boot), bootloading failures handled by the source code all result in a call to `halt()`, which contains an infinite loop. We model this explicit failure in Mew-Boot by defining a *failure* substage, s_4, whose entrypoint is the `halt` function, but is neither in the range nor domain of $\mathbb{Q}^{\mu}$. Any transition into s_4 is thus automatically and immediately considered a *policy violation* (violations are discussed in the following section).

5.2.6 Policy violations

An RBWAC policy consists of typed substages, substage transition rules (encoded as a linear sequence in RBWAC$^{\mu}$), typed regions, region definition transition rules, and type-based substage/region access rules. The two conditions that indicate a policy violation are (1) an out-of-type write, and (2) an out-of-order substage execution.

These two violation conditions capture the weaknesses I have found to be common among vulnerable loaders (as discussed in chapter 2 and appendix B): *type confusion*, *verification failure*, and *enforcement failure*. These three kinds of loader weaknesses are often related to and/or are consequences of each other – type confusion may lead to verification and enforcement failures. A verification failure may lead to type confusion or enforcement failure. Likewise, an enforcement failure may lead to a verification failure or type confusion.

The term *type confusion* has historically been used to describe the bugs/vulnerabilities that arise when an actor with an object in a manner inconsistent with the object's

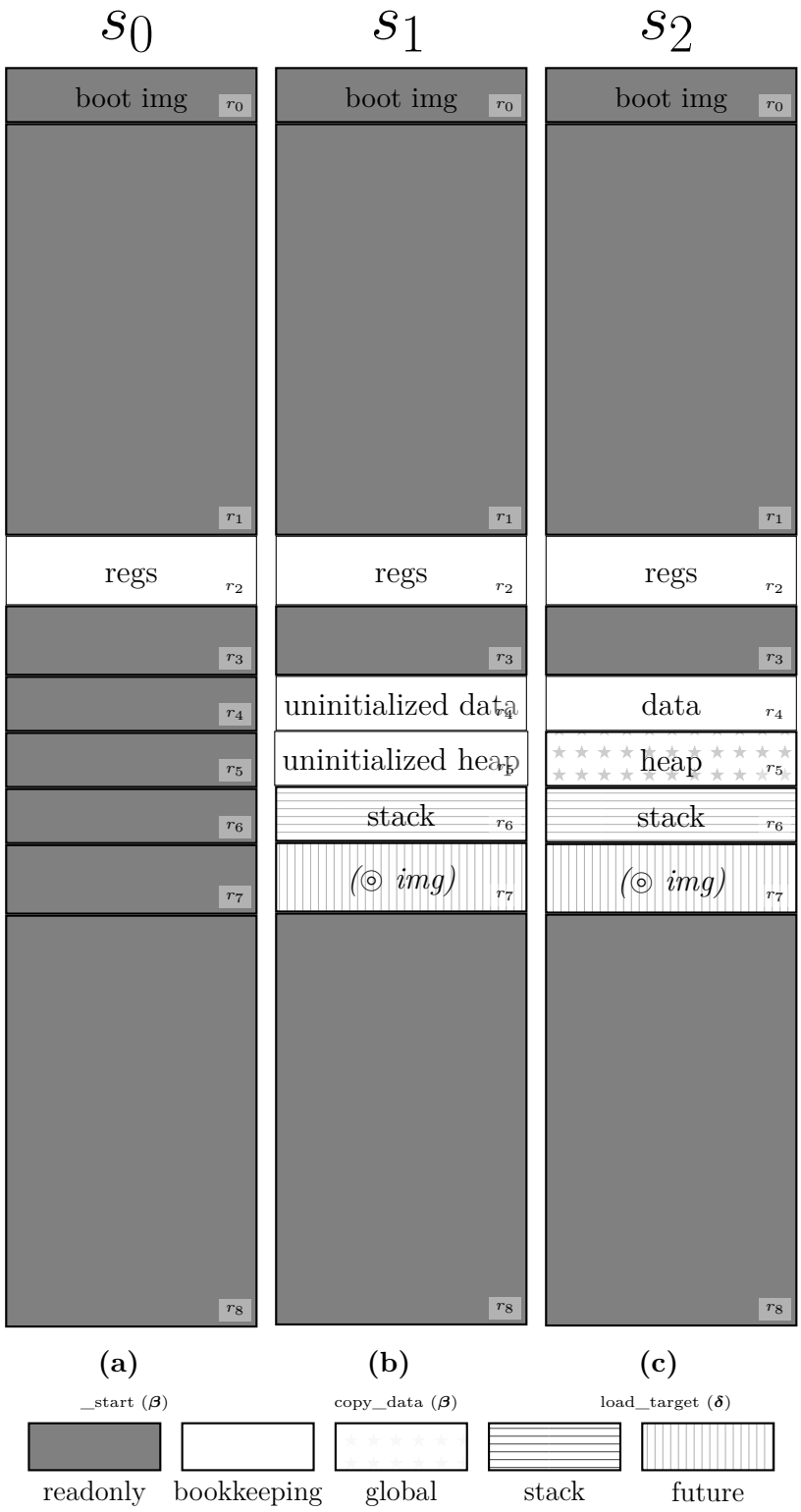


Figure 5.8: Region definitions for substages s_0–s_2 in basic ManulBoard Mew-Boot policy. Each region's name is displayed on lower right-hand of its position in the memory map diagram, its shading indicates the region's type.

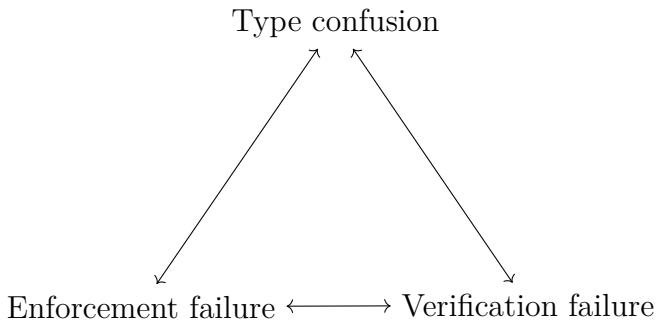


Figure 5.9: Type confusion and verification failures, and enforcement failures are all interdependent and occasionally co-exist

type. However, I will use this term more generally to include *all inconsistencies that involve object/memory region's semantics*. With this in mind, **type confusion** can be broken into two subclasses: *type overlap* and *type superposition*. **Type overlap** is when an actor interacts with an object in a manner inconsistent with the object's type (i.e., what others refer to as *type confusion*). **Type superposition** is when multiple separate actors incorrectly assume that they are interacting with the same object/region.

There are not many known instances of *type superposition*-based loader weaknesses. Such weaknesses are no necessarily rare, rather I believe that they are more complex in nature than *type overlap*, *verification*, and *enforcement* failures in that they require having *multiple* actors that interpret to allow for conflicting interpretations. For example, independent parser implementations that parse/interpret the same object differently. The Android master-key bug [79, 80], is one of the few known instances of loader-related *type superposition*, and is caused by independently-implemented parsers that interpret the same data structure that contains code signing information differently. Therefore, although I believe this class of software weaknesses is important, I will not be formally defining or discussing *type superposition*-based loader weaknesses in this thesis, instead leaving a deeper investigation of such weaknesses as future work.

A more formal definition of *type overlap* is as follows: Let $e^* = (s^*, r^*, o^*)$ (where $s^* \in \mathbb{S}$, $r^* \in \mathbb{R}$, and $o^* \in \mathbb{O}$) represent an operation that occurred during execu-

tion, and $\mathbb{P}^{\mu}$ be the system's policy. **Type overlap** is defined as a mediated event $e^* = (s^*, r^*, o^*)$ where $\mathbb{P}^{\mu}(s^*, r^*, o^*) = \text{deny}$.

Verification and *enforcement* failures are similar with respect to RBWAC's formalism, as both are forms of temporal violations. A formal definition of these failures is as follows:

Let substage $s^*p \in \mathbb{S}$ be a *non-failure* substage (in the domain or range of $\mathbb{Q}^{\mu}$) in which some property p is validated. A **verification failure** is said to have *occurred* in an *invocation* $\mathbb{I}^*$ if $s^p \notin \mathbb{I}^*$. An *enforcement* failure implies that there is some action, a, that *only* and *always* occurs during substage $s^a \in \mathbb{S}$, and is expected to occur *before* substage $s^* \in \mathbb{S}$ where $s^* \neq s^a$, but didn't. In other words, an *invocation*, $\mathbb{I}^*$ is said to exhibit an **enforcement failure** if s^a is not before s^* in $\mathbb{I}^*$.

The occurrence of an enforcement or verification failure indicates that the boot-loader did not execute all intended actions or executed them in an incorrect order. In order to protect against such failures, a RBWAC policy designer should consider introducing a substage that narrowly contains this action so that it is more visible to other developers (as a significant event), *and* can be explicitly targeted by the policy.

5.2.7 Policy rules and logic

Each RBWAC policy instance must define the set of rules that provide a deterministic allow or deny decision based on the *currently-executing* substage's type, the type of the region within which the address being operated on lies, and the type of the operation being performed. An example rule set is discussed in section 5.3. Such rules and desired behavioral properties based on such rules could also be formally modeled as a *linear time logic* [173].

5.3 RBWAC$^{\mu}$ sample policy instances

In this section I present a summary of three different RBWAC$^{\mu}$ policy instances, (1) a simple policy for the ManulBoard, (2) a more complex ManulBoard policy, and (3) my

case study on the BeagleBoard-xM U-Boot SPL. All three policy instances share the following RBWAC class characteristics:

$S^\mu = \{\boldsymbol{\delta}, \boldsymbol{\pi}, \boldsymbol{\beta}\}$ as defined in table 5.1 on page 82, $R^\mu = \{\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{b}, \boldsymbol{g}, \boldsymbol{f}, \boldsymbol{p}\}$ as defined in table 5.2 on page 89, $O^\mu = \{\texttt{write}\}$, $b^\mu = 0$, $e^\mu = 2^{32}$ (0xFFFFFFFF is the highest addressable value in these architectures), $\mathbb{E}^\mu = \{s_0\}$, and $\mathbb{F}^\mu = \{s_n\}$ where $n < |\mathbb{S}|$. Furthermore, $\mathbb{Q}^\mu$ should only allow for a single sequential ordering of substages, and thus be in the form of: $\mathbb{Q}^\mu(s_i) = \{s_{i+1}\}$ if $0 \leq i < n$ (as discussed in section 5.2.4). Finally, $\mathbb{P}^\mu$ is defined based on the set of predicated defined in figure 5.10 so that given $s_t \in \mathbb{S}, r_u \in \mathbb{R}$, $\texttt{typeof\_substage}(s_t) = t$, and $\texttt{typeof\_region}(r_u, s_t) = u$,

$$\texttt{allowed}(s_t, r_u, \texttt{write}) \;\Leftrightarrow\; \mathbb{P}^\mu(s_t, r_u, \texttt{write}) = \text{allow}$$

```
substage(S)                 :- true
region(R)                   :- true
loading_substage(s_δ)       :- true
patching_substage(s_π)      :- true
bookkeeping_substage(s_β)   :- true
stack_region(r_s)           :- true
global_region(r_g)          :- true
bookkeeping_region(r_b)     :- true
future_region(r_f)          :- true
patching_region(r_p)        :- true

allowed(S,R,write) :- stack_region(R)

allowed(S,R,write) :- global_region(R)

allowed(S,R,write) :- bookkeeping_substage(S),
                      bookkeeping_region(R)

allowed(S,R,write) :- loading_substage(S),
                      future_region(R)

allowed(S,R,write) :- patching_substage(S),
                      patching_region(R)
```

Figure 5.10: Predicates that form the RBWAC$^\mu$ class' policy rules ($\mathbb{P}^\mu$)

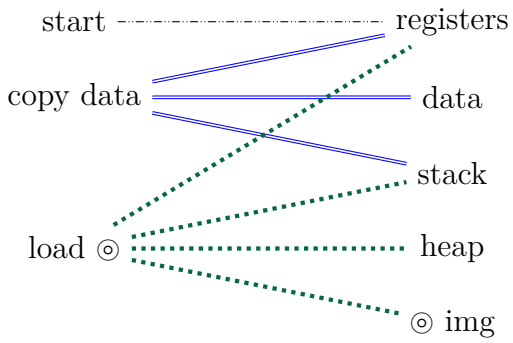


Figure 5.11: Explicitly defined semantic relationships between substages and regions for basic ManulBoard Mew-Boot policy. Nodes on the left represent substages, regions on the right. An edge between a substage and region denotes write access is allowed.

5.3.1 Basic ManulBoard policy

The substages of the basic ManulBoard policy that I have been using as an example throughout the earlier parts of this chapter can be formally defined as an instance of the RBWAC$^{\mu}$ class with,

$\mathbb{S} = \{s_0, s_1, \ldots s_4\}$,

$$\texttt{typeof\_substage}(s_i) = \begin{cases} \boldsymbol{\delta} & \text{if } i = 2 \\ \textit{success} & \text{if } i = 3 \\ \textit{failure} & \text{if } i = 4 \\ \boldsymbol{\beta} & \text{otherwise} \end{cases},$$

and $\mathbb{F}^{\mu} = \{s_3\}$.

Substage s_0 corresponds to `_start()`, `copy_data()` is the entrypoint to substage s_1, `load_target()` is the entrypoint to substage s_2, `jump_to_target()` is the entrypoint to substage s_3 (the *success* substage), and `halt()` is the entrypoint to a *failure* substage (s_4)

This combination of region and substage definitions generates a policy that can be illustrated as a bipartite graph, as shown in figure 5.11, where the left set of nodes represents each substage, the right set of nodes represents each region definition, and an edge between a region and substage indicates `write` operations are allowed.

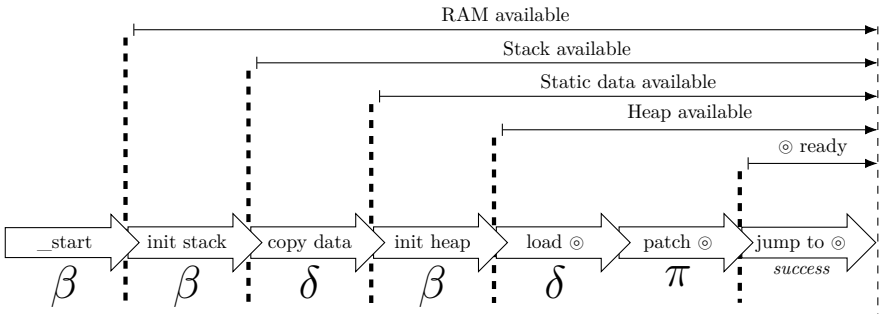


Figure 5.12: Substage sequence defined by more complex ManulBoard Mew-Boot policy. Each arrow contains a substage name and points to the policy's subsequent substage. The label below each substage's arrow denotes the substage's type. Important features common to one or more substage are displayed above the substage arrows. ◎ symbol is a shorthand for *target*.

5.3.2 A more complex ManulBoard policy

RBWAC is naturally flexible, allowing for a multitude of granularities. If we wish to introduce more finely-grained mechanisms that limit access to static data until they are available, as well as separate the target loading phase from its patching phase, we can do so by including more substages and region definitions. This more finely-grained policy introduces three substages that are not present in the previously-described basic policy: (1) the *init_stack* s_1 substage, which occurs after RAM is initialized (during s_0 *_start*), but before data is made available by the (now s_2) `copy_data` substage, (2) `init_heap`, substage s_3, which occurs after static data is made available by `copy_data`, to protect the heap until it is initialized, and (3) `patch_target` (substage s_5) which captures the bootloader's target image patching operations so they can be treated differently from the bootloader's loading operations that occur earlier during the `load_target` (now s_4) substage. A high-level view of this more detailed policy is depicted in figure 5.12.

The regions defined by this more complex policy are similar to the basic policy, but with additional region transitions that supplement the three new substages. The region definitions that complement the three additional substages are depicted in

figure 5.13. Figure 5.14 graphically represents this more complex policy.

This more complex RBWAC$^{\mu}$ policy instance for the ManulBoard is defined as:

$\mathbb{S} = \{s_0, s_1, \ldots s_7\}$,

$$\texttt{typeof\_substage}(s_i) = \begin{cases} \boldsymbol{\delta} & \text{if } i \in \{2, 4\} \\ \boldsymbol{\pi} & \text{if } i = 5 \\ \textit{success} & \text{if } i = 6 \\ \textit{failure} & \text{if } i = 7 \\ \boldsymbol{\beta} & \text{otherwise} \end{cases}$$

and $\mathbb{F}^{\mu} = \{s_6\}$.

5.4 Retrofitting an RBWAC$^{\mu}$ instance

Until this point I have limited our focus on a simple and imaginary bootloader executing on pretend hardware. However, my thesis aims to present *practical* loader hardening techniques. Therefore, instead of focusing on how to build a hardened bootloader from scratch, I focused on developing a *methodology* for *point-rearchitecting* existing bootloaders so that they can be *incrementally* hardened. To this end, I have written a third RBWAC$^{\mu}$ policy instance, specifically for the popular U-Boot bootloader, and more specifically for U-Boot SPL bootloading stage compiled for BeagleBoard-xM (BBxM) development board. Like the previous example policy instances, this policy's overall goal is to prevent the loading and patching behaviors from overwriting bookkeeping data, and vice versa – recall that such accesses have been many attack's primary tool (as exemplified in [120, 175, 225]).

U-Boot, like the imaginary Mew-Boot, goes through a sequence of phases before it successfully loads its target. However, unlike Mew-Boot, which consists of a single stage that ultimately executes the final target, U-Boot is divided into *two* separate stages (each stored in its own binary image) – (1) the SPL (secondary program loader),

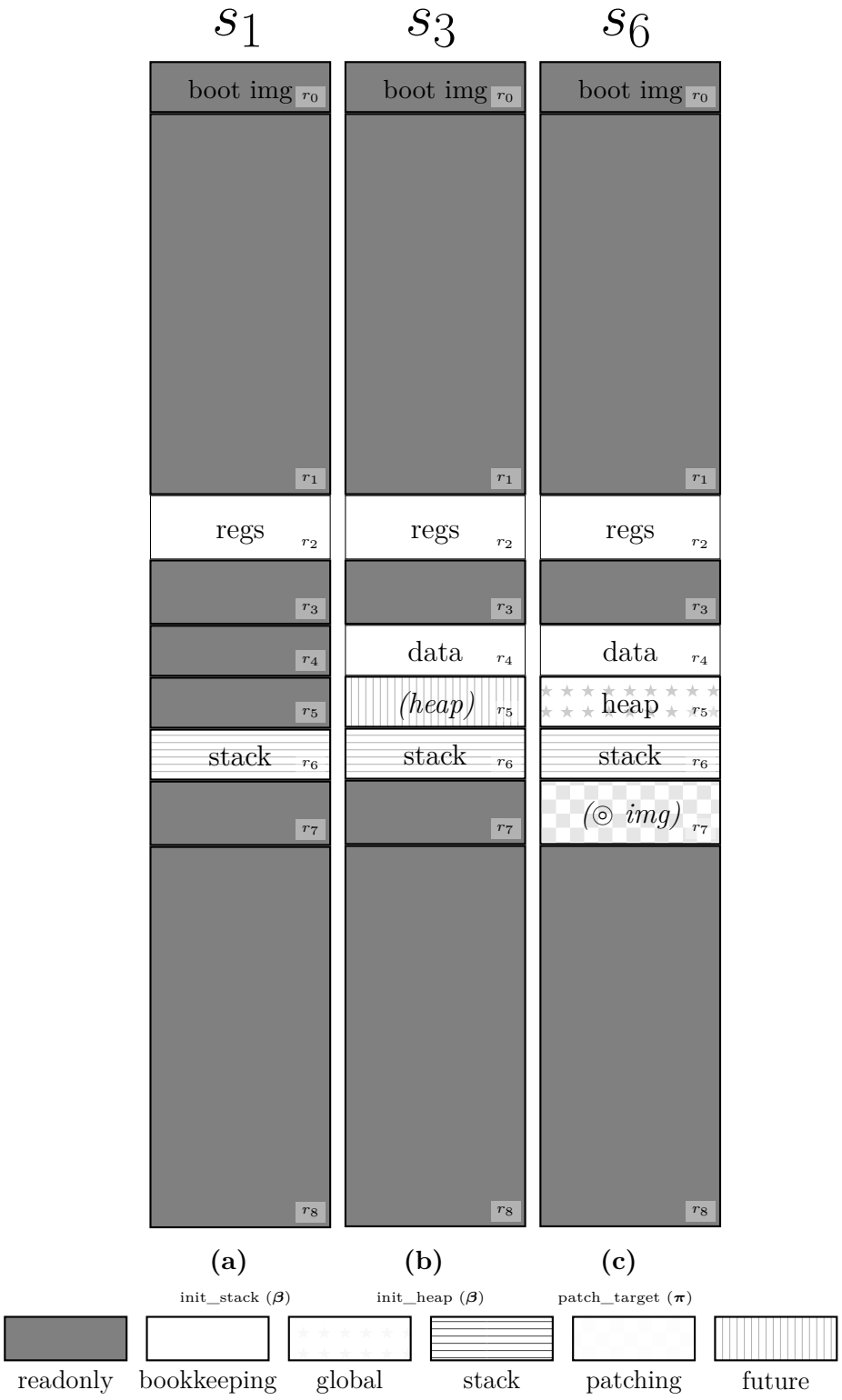


Figure 5.13: Region definitions for more complex ManulBoard Mew-Boot policy. Each region's name is displayed on lower right-hand of its position in the memory map diagram, shading indicates region's type. Region definitions for substage s_0 is in figure 5.8a, s_2 is in figure 5.8b, and s_5 is in figure 5.8c all on page 93.

Table 5.3: RBWAC$^{\mu}$ policy $\mathbb{P}$'s allowed writes

		Substage type		
		bookkeeping ($\boldsymbol{\beta}$)	*loading* ($\boldsymbol{\delta}$)	*patching* ($\boldsymbol{\pi}$)
Region type	STACK (**s**)	✓	✓	✓
	BOOKKEEPING (**b**)	✓		
	GLOBALS (**g**)	✓	✓	✓
	FUTURE (**f**)		✓	
	PATCHING (**p**)			✓
	READ-ONLY (**r**)			

✓denotes write is allowed

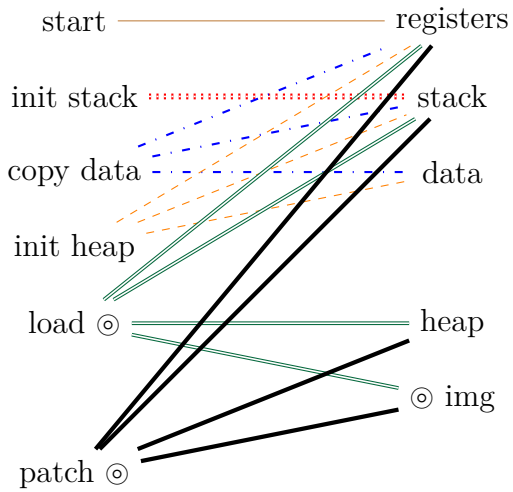


Figure 5.14: Explicitly defined semantic relationships between substages and regions in more complex ManulBoard Mew-Boot policy. Nodes on left represent substages, regions are on the right. An edge between a substage and region denotes write access is allowed.

which is a small binary with few features so that it can fit inside the BBxM application processor's 64KB of on-chip RAM, and (2) a stage which I call the *main* stage – a large, full-featured stage loaded by the SPL onto external RAM after it initializes said RAM. Both stages are built from the same codebase – much of what is present in the SPL is also compiled into the *main* stage. Although I also worked on a policy for the more-complex *main* stage, I will only present a policy for the SPL stage here.

5.4.1 Bootloader reconnaissance and substage extraction

Method. Using the tools described in section 4.4 on page 63, I identified substages, region definitions, and relocation behavior by iteratively searching for and testing candidate policies. In order to develop a candidate policy, I iteratively (1) collect information on the bootloader's executed sequence of write operations – the fetched write instruction's register states, write index, and destination address – via my tracing tools discussed in section 4.2 on page 58, and then (2) query this database for particular write patterns characteristic of loading and patching operations as well as writes destined to known objects, until (3) no unaccounted-for writes remain. In particular, I search for *consecutive sequences of writes* performed to *consecutive addresses* by the *same instruction* – such a pattern suggests indicate an image or data structure is being copied into another region of memory, indicative of potential loading of a future substage. My tools also help identify candidate substage entrypoints by generating dynamic call graphs of the bootstage's control flow and correlating write operations with its corresponding node in the call graph.

Relocation support. Before I can truly test a candidate RBWAC$^{\mu}$ policy instance, I first needed to identify *all* phases of runtime loader self-relocation/modification. This is important so the tools can keep track of and identify *all* symbols/objects by location *throughout* execution, even after phases of self-modification. It is not uncommon for bootloaders to relocate *their own code or data* during execution, but

continue to address non-relocated or pre-relocated objects. For example, figure 5.15 shows one example of U-Boot self-relocation where it relocates its entire memory-based image. Here it uses general-purpose registers (`r0`, `r1`, `r2`, `r3`, `r4`) to keep track of its self-relocation process. After it finishes copying itself, it jumps to `fixloop` (on line 16) to patch absolute addresses in the relocated image so they reflect their current location. These updated addresses are calculated based on a table generated by the compilation toolchain (ELF's `.rel.dyn` table). More specifically, the patching performed by `fixloop` (starting on line 13 of figure 5.15) (1) iterates through the `.rel.dyn` table of compiler-generated offsets to image-based *absolute* pointers (which defined with respect to the beginning on the image), and (2) updates the pointer *at its relocated address* so its value reflects *the pointer's relocated address*. In general, loader verification will likely have to deal with invariants of such self-relocating loops. Any *complete* RBWAC$^{\mu}$ policy must account for such operations. This highlighted relocation procedure, however, is not performed by the SPL (it is only performed by the *main* stage), and therefore we will not encounter it during the U-Boot SPL case study.

If we take a step back and inspect what happens before and after the call to `relocate_code`, we will find more examples of how U-Boot subtly manages absolute addresses around its self-relocation. The instructions surrounding the call to `relocate_code` are shown in figure 5.16. Before U-Boot self-relocates via its branch to `relocate_code`, it sets up the `lr` register (line 5), which traditionally holds the current function's return address, to contain the address of the relocated label `here` (defined on line 9), so that when `relocate_code` returns (as shown on line 31 of figure 5.15), it fetches instructions from *relocated location* of `here` and `not` the (still-present) original location to which it would otherwise return. At this point, U-Boot calls `c_runtime_cpu_setup` (line 11) at its relocated address, *despite what the comment next to the instruction suggests*. What is not apparent is that

```
1 ENTRY(relocate_code)
2   ldr r1, =__image_copy_start # r1 <- SRC &__image_copy_start
3   subs r4, r0, r1         # r4 <- relocation offset
4   beq relocate_done         # skip relocation
5   ldr r2, =__image_copy_end # r2 <- SRC &__image_copy_end
6
7 copy_loop:
8   ldmia r1!, {r10-r11}        # copy from source address [r1]
9   stmia r0!, {r10-r11}        # copy to   target address [r0]
10  cmp r1, r2                  # until source end address [r2]
11  blo copy_loop
12
13  # fix .rel.dyn relocations
14  ldr r2, =__rel_dyn_start # r2 <- SRC &__rel_dyn_start
15  ldr r3, =__rel_dyn_end # r3 <- SRC &__rel_dyn_end
16 fixloop:
17  ldmia r2!, {r0-r1}          # (r0,r1) <- (SRC location,fixup)
18  and r1, r1, #0xff
19  cmp r1, #23                  # relative fixup?
20  bne fixnext
21
22  # relative fix: increase location by offset
23  add r0, r0, r4
24  ldr r1, [r0]
25  add r1, r1, r4
26  str r1, [r0]
27 fixnext:
28  cmp r2, r3
29  blo fixloop
30 relocate_done:
31  bx lr
32 ENDPROC(relocate_code)
```

Code fragment based on U-Boot's definition of `relocate_code` in arch/arm/lib/relocate.S

Figure 5.15: Example of self-relocation implementation in U-Boot source code

```
1    # Set up intermediate environment (new sp and gd) and call
2    # relocate_code(addr_moni). Trick here is that we'll return
3    # 'here' but relocated.
4
5     adr lr, here
6     ldr r0, [r9, GD_RELOC_OFF]   # r0 = gd->reloc_off
7     add lr, lr, r0
8     b relocate_code
9  here:
10    # Set up final (full) environment
11    bl c_runtime_cpu_setup  # we still call old routine here
12    b clear_bss
```

Code fragment based on U-Boot's definition of `_main` in arch/arm/lib/crt0.S

Figure 5.16: How U-Boot manages absolute pointer values in registers before and after self-relocation

the branch to `c_runtime_cpu_setup` (on line 11) is encoded as a branch *relative* to the instruction's program counter. Given we know `relocate_code` passes execution to the newly-relocated address of `here` and the branch at this location is calculated relative its program counter, the branch's target will also be a relocated address. A proper loop invariant must account for such details in order to formally model and verify self-relocation, even if they are not so easily extracted from the loader's source code.

Memory write patterns. Using my instrumentation tools, I can extract information on *every* memory write performed during execution, including the number of write operations that have occurred, destination of each write, and number of bytes written – these are the data from which I build a RBWAC$^{\mu}$ policy instance for the U-Boot SPL.

At this point it is helpful to introduce additional terminology for analyzing and classifying memory write patterns. Let us define a **write operation** o as a tuple:

$$o = (n, \texttt{pc}, \texttt{va}, \texttt{dest}, \texttt{node}),$$

where: n is the number of write operations that occurred before this write operation

(its *write index*), `pc` is value of the processor's program counter (instruction pointer) that performed the write and `va` is the *virtual address* of the write instruction from the point-of-view of the compilation tool chain – where it expects the instruction to be located in memory *before* any relocation. Virtual addresses are helpful in looking up statically-generated information such as a write instruction's mnemonic, operands, the number of bytes it writes, and location in the source code from which it was generated. `dest` is the address to where the write is performed, and `node` uniquely identifies its corresponding not in the dynamically-generated call graph, i.e, when the write occurred with respect to the call graph.

We can use these collected *write operations* to generate an enhanced call graph – a **write-augmented call graph** – and annotate each node with the data collected on the write operations performed during that function call. This so-called *write-augmented* call graph helps us assign *intent* to individual write operations. These write-augmented call graphs also can highlight memory writing patterns, such as phases when consecutive bytes of memory are written to by a consecutively repeated write instruction (within a tight loop), which one may expect to observe during relocation.

To capture occurrences of repeated consecutive memory writes, we make a projection called `W_to_B` of the set W containing all *write operations* onto a set B containing all *block write operations*. `W_to_B`: $W \rightarrow B$. A **block write operation** $\hat{b}_i$ is defined as:

$$\hat{b}_i = (i, \texttt{pc}, \texttt{va}, \texttt{dest}, \texttt{size}, \texttt{repetitions}).$$

For a given $\hat{b}_i$, i represents the block write's index in the projection's ordering; `pc`, `va`, `dest`, and `node` are the same as in a *write operation*; `size` is the total number of bytes written starting at address `dest`; and `repetitions` is the number of *write operations* that were combined in order to form this single *block write operation*. Figure 5.17 shows pseudocode that calculates this projection.

```
def W_to_B:(W):
    B = set()
    last = None
    last_size = None
    last_pc = None
    last_va = None
    last_dest = None
    last_node = None
    last_reps = 0
    index = 0

    for (n, pc, va, dest, node) in sorted(W):  # Sort W by index
        size = lookup_size(va)
        if (last_pc == pc) and (last_va == va) and \
                    (last_dest + last_size == dest):
            last_size += size
            last_reps += 1
        else:
            if last_n is not None:
                B.insert((index, pc, va, last_dest,
                          last_size, last_node, last_reps))
                index += 1
            last_n = n
            last_pc = pc
            last_va = va
            last_dest = dest
            last_size = size
            last_node = node
            last_reps = 1

    # insert final block write
    if last_n is not None:
        B.insert((index, last_pc, last_va, last_dest,
                  last_size, last_node, last_reps))
    return B
```

Figure 5.17: Pseudocode that implements projection of *write operations* onto *block write operations*

```
err = spl_load_image_fat(&mmc->block_dev,
                         CONFIG_SYS_MMCSD_FS_BOOT_PARTITION,
                         CONFIG_SPL_FS_LOAD_PAYLOAD_NAME);
```

Code fragment based on U-Boot's definition of `spl_mmc_do_fs_boot` in common/spl_mmc.c

Figure 5.18: U-Boot code that locates SPL's target in a FAT-formatted SD card

I define a **repeated block write** as any block write $\hat{b}_i$ where

$$\hat{b}_i = (i, \texttt{write\_index}_i, \texttt{pc}_i, \texttt{va}_i, \texttt{dest}_i, \texttt{size}_i, \texttt{repetitions}_i), \text{ and } \texttt{repetitions}_i > 1.$$

A *repeated block write* has two indices (1) i, which is its position relative to all other *repeated block writes*, and (2) $\texttt{write\_index}_i$ which is its index of its earliest (non-repeated) *block write operation*.

BBxM U-Boot SPL Internals

For the purpose of this case study, I focus on a common BBxM U-Boot SPL configuration where the target image is stored in a file named "u-boot.img" located in the root directory of an inserted FAT-formatted SD card which is loaded by the processor's on-chip-ROM-based kickoff bootloader using its own, limited, FAT filesystem driver[6]. U-Boot's SPL stage is designed to operate on systems with limited volatile memory and thus the SPL has limited flexibility. This is evident in the few kinds of targets it supports and by the fact that the partition number and file name of the target image are hard-coded into its source code, as we can see in figure 5.18, which shows how it loads its target image from the SD, and figure 5.19, which shows the hard-coded values it references to to load the target from a FAT-formatted SD card.

The overall sequence of events that happen during the BBxM's U-Boot SPL stage as it boots from a SD card are:

1. Saves pointer to boot parameters passed to it by the BBxM's boot ROM via the `r0` register

[6]The am37x's SD card-based boot procedure is described in detail in [202]. If the driver finds a MBR-type partition table, it requires that the target image be stored in a file named "MLO" in the *root directory* of a FAT12/16/23 file system contained within an *active primary* partition.

```
/* FAT sd card locations. */
#define CONFIG_SYS_MMCSD_FS_BOOT_PARTITION 1
#define CONFIG_SPL_FS_LOAD_PAYLOAD_NAME "u-boot.img"
```

Code fragment based copied from U-Boot file include/configs/ti_armv7_common.h

Figure 5.19: Hard-coded values in that allow U-Boot SPL to locate the target image on a FAT-formatted SD card.

2. Initializes CPU and hardware-specific features such as caches
3. Repositions stack to slightly larger region (from $\texttt{stack}_0$ to $\texttt{stack}_1$)
4. Relocates `go_to_speed` function
5. Performs more low-level initialization including hardware clock configuration and processor pin multiplexing
6. Moves stack again as it begins to reshuffle on-chip memory (to $\texttt{stack}_2$)
7. Zeros out and initializes a special *global data* structure that holds important bootloader bookkeeping information (structure definition in figure 5.20)
8. Moves stack yet again ($\texttt{stack}_3$)
9. Initializes external volatile memory (also knows as the *SRAM*)/bx/Music
10. Moves stack to final position ($\texttt{stack}_4$)
11. Zeros out BSS region
12. Initializes heap
13. Parses and saves boot parameters passed to SPL (from the previous/kickoff boot stage) into *global data* (pointer to parameter structure was saved in step 1)
14. Initializes console
15. Initializes SD reader drivers and hardware
16. Checks how the SD card is formatted
17. Loads beginning (header) of file containing target image into memory
18. Parses the header to determine target's specified load address and entrypoint
19. Loads full target image into memory at specified load address
20. Jump's to target's entrypoint

Table 5.4: A successful SPL execution's write operations

# memory writes	~400,000
# block writes	~10,000
# repeated block writes	~700

There are two major phases of the U-Boot SPL delineated by when the external memory (also referred to as the SRAM) becomes initialized (step 9 of the preceding list). Before external SRAM is initialized, the U-Boot SPL has little memory available to it, and so it carefully dances around the in-use regions of the available RAM (its on-chip RAM) as it performs its initialization tasks. The location of the stack is changed *four* times as a byproduct of this dance during this pre-SRAM phase.

Figure 5.21 shows how the sequence of steps outlined as the SPL's boot procedure overlap with (a portion of) the call graph generated from an execution of the SPL as it loads the U-Boot main stage image (also) located on an inserted SD card. The operations highlighted in this figure form the basis its RBWAC$^{\mu}$ policy instance. From this call graph we can get a sense of where non-returning continuations – from which substages are defined – are present in the bootloader.

BBxM U-Boot SPL substage and region derivation

A great deal of useful information can be derived from a single successful run of the SPL stage. Given no difference in hardware or contents of non-volatile storage, the SPL's memory write behaviors are fairly deterministic and do not vary much between different executions, besides the number of times particular timing-dependent actions are repeated. For example, when U-Boot interacts with hardware subsystems that operate asynchronously with respect to its CPU, such as the SD/MMC card reader, U-Boot uses a *busy loop*-type design pattern that waits until the subsystem is ready. A couple versions of this busy loop modify memory, e.g., by calling a function that reads the hardware clock at each iteration (which executes a `push` instruction), as can be seen in figure 5.22.

```
typedef struct global_data {
 bd_t *bd;
 unsigned long flags;
 unsigned int baudrate;
 unsigned long cpu_clk; /* CPU clock in Hz!                */
 unsigned long bus_clk, pci_clk, mem_clk;
 unsigned long have_console; /* serial_init() was called */
 unsigned long env_addr; /* Address  of Environment struct */
 unsigned long env_valid; /* Checksum of Environment valid? */
 unsigned long ram_top; /* Top address of RAM used by U-Boot */
 unsigned long relocaddr; /* Start address of U-Boot in RAM */
 phys_size_t ram_size; /* RAM size */
 unsigned long mon_len; /* monitor len */
 unsigned long irq_sp; /* irq stack pointer */
 unsigned long start_addr_sp; /* start_addr_stackpointer */
 unsigned long reloc_off;
 struct global_data *new_gd; /* relocated global data */
 struct udevice *dm_root; /* Root instance for Driver Model */
 struct udevice *dm_root_f; /* Pre-relocation root instance */
 struct list_head uclass_root; /* Head of core tree */
 const void *fdt_blob; /* Our device tree, NULL if none */
 void *new_fdt; /* Relocated FDT */
 unsigned long fdt_size; /* Space reserved for relocated FDT */
 struct jt_funcs *jt; /* jump table */
 char env_buf[32]; /* buffer for getenv() before reloc. */
 int cur_i2c_bus; /* current used i2c bus */
 unsigned long timebase_h, timebase_l;
 unsigned long malloc_base; /* base address of early malloc() */
 unsigned long malloc_limit; /* limit address */
 unsigned long malloc_ptr; /* current address */
 struct udevice *cur_serial_dev; /* current serial device */
 struct arch_global_data arch; /* architecture-specific data */
} gd_t;
struct arch_global_data { /* needed by most of timer.c on ARM */
 unsigned long timer_rate_hz;
 unsigned long tbu;
 unsigned long tbl;
 unsigned long lastinc;
 unsigned long long timer_reset_value;
 unsigned long tlb_addr, tlb_size;
 u32 omap_boot_device;
 u32 omap_boot_mode;
 u8 omap_ch_flags;
};
```

Code from arch/arm/include/asm/global_data.h and include/asm-generic/global_data.h

Figure 5.20: U-Boot's definition *global data* structure

Figure 5.21: Ordering of subset of function calls (prepended by >) and returns (<) during successful execution and loading of *main* stage from SD card, produced by my *calltrace* tool (introduced on p. 64).

```
while (size) {
        ulong start = get_timer(0);
        do {
                mmc_stat = readl(&mmc_base->stat);
                if (get_timer(0) - start > MAX_RETRY_MS) {
                        printf("%s: timedout waiting for status!\n",
                                        __func__);
                        return TIMEOUT;
                }
        } while (mmc_stat == 0);
```

Code fragment from U-Boot's definition of `mmc_read_data` in drivers/mmc/omap_hsmmc.c

Figure 5.22: Example of busy loop in U-Boot which contains a function call. This results in memory writes due to values being pushed to the stack as part of the function call.

As summarized in table 5.4, a successfully executed SPL performs on the order of 400,000 writes to memory. These 400,000 individual writes can be projected onto about 10,000 *block writes*. Of these 10,000 *block writes*, close to 700 are classified as *repeated block writes* – in other words, 700 of the 10,0000 *block writes* happened in a tight loop where no other memory write occurs and are likely part of a memory copying/relocation operation.

By searching for memory write patterns such as repeated block writes (as one would expect from operations like `memcpy()`), I was able to identify when and where major loading and patching operations occur. From these data, I found one instance of self-relocation – of a function that incidentally is never executed. I also when U-Boot zeroes a region to hold *global data* bookkeeping structure, zeroes the BSS region, copies filesystem metadata, reads in the target's image header, and reads in the rest of the target image (in 512KB chunks). Figure 5.23 summarizes these findings by showing the first 4,800 *repeated block writes* that occur during a successful execution of the SPL, overlaid with the corresponding call graph (more-or-less) vertically separated by time (function calls that happen earlier appear closer to the top of the plot). The last 5,200 *repeated block writes* continue the pattern that starts at around the $4{,}400^{th}$ *block write*, all of which comprise the writes produced from loading the target image off

the SD card, one 512K sector at a time. All of these data were collected and analyzed in under five minutes (including an initial binary analysis to determine breakpoint locations that takes about two minutes, but only needs to be performed once).

Patching phases can be identified by executing the bootloader under a policy that separates loading phases from all other behaviors – all writes that violate this policy are likely patching the target image. These invalid writes emerge when FUTURE regions are modified outside any *loading* substages. Using this technique, I identified a phase when U-Boot tested its recently-initialized external memory by writing and reading special values to small number of external memory-backed addresses. Likewise, I also found a handful of statically-allocated and heap-allocated variables that are modified during loading and patching phases. It also became apparent that the stack was relocated multiple times during the boot process. Each iteration of this discovery process took fewer than three minutes to execute with the help of my tools. These tools made all the difference in developing a RBWAC policy.

I ultimately settled on a sequence of thirteen substages that separated the U-Boot SPL's bookkeeping, loading, and patching phases. Approximately 50 lines of code were modified or added to achieve continuation-style substage transitions. However, nearly half of these changes resulted from copy and pasting portions of the code to allow for conditional compilation of the RBWAC-friendly control flow[7].

5.5 BBxM U-Boot SPL RBWAC$^{\mu}$ policy

My goal for this case study was to design an RBWAC policy that ensures the following security properties, $\boldsymbol{p_0} - \boldsymbol{p_4}$:

$\boldsymbol{p_0}$: Regions of memory containing in-use code (text) cannot be overwritten (type overlap, enforcement, or verification failure)

[7]Although I made minimal changes to the U-Boot codebase in order for RBWAC, I did alter the compilation process so that unused functions were not built into the binary image. This change assured that a debugger-friendly image could also fit in the BBxM's on-chip RAM. Additionally, I inserted markup in the form of empty `#define` statements to aid my static analysis tools.

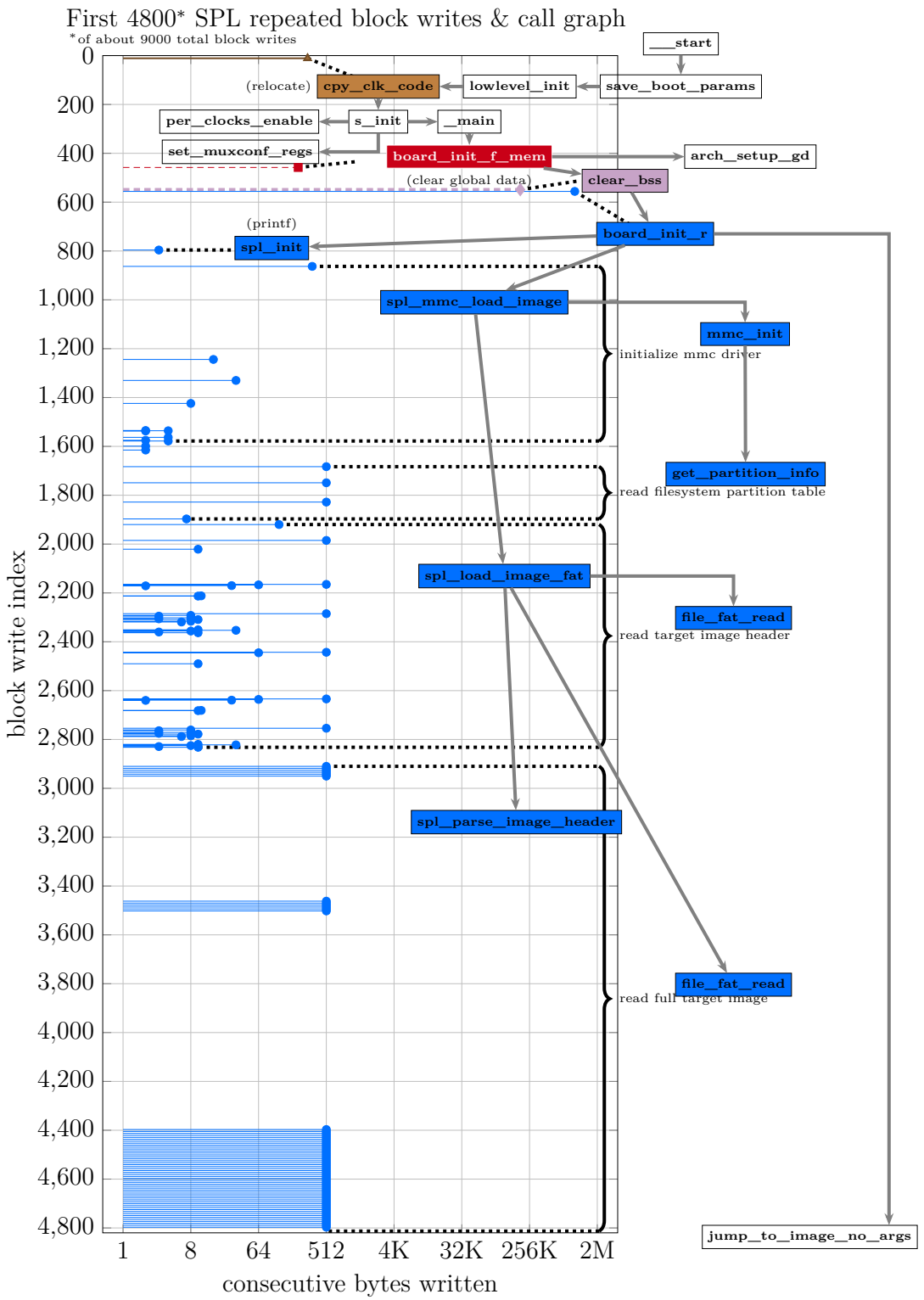


Figure 5.23: The first 4,800 *repeated block writes* that occur during a successful execution of the SPL, overlaid with the corresponding call graph (more-or-less) vertically separated by time (function calls that happen earlier appear closer to the top of the plot). The last 4,800 *repeated block writes* continue the pattern that starts at around the 4,400th *block write*, all of which comprise the writes produced from loading the target image from the SD card, one 512K sector at a time.

Table 5.5: BBxM U-Boot SPL policy statistics at a glance

Total # substages	14
# *failure* substages	1
# *bookkeeping* substages	9
# *loading* substages	2
# *patching* substages	3
# named regions	27
# temporarily retyped symbols	20
# times region definitions change	9
Median # writes per substage	9
Min # writes per substage	0
Max # writes per substage	351999

- $\boldsymbol{p_1}$: Regions reserved for future substage images cannot be written to during bookkeeping phases (type overlap)
- $\boldsymbol{p_2}$: Loading and patching phases cannot corrupt data intended for bookkeeping (type overlap, enforcement, or verification failure)
- $\boldsymbol{p_3}$: Regions reserved for external memory, the BSS, the heap, and the SRAM's *global data* cannot be written to by *bookkeeping* substages until initialized (type overlap)
- $\boldsymbol{p_4}$: An overall reduction in the amount of writable memory at any given point during execution (least privilege [182])

More formally, $\boldsymbol{p_0}$ requires that during any substage $s_i \in \mathbb{S}$, and
$\forall\, r_j = (b_j, e_j) \in \texttt{used\_regions}(s_i)$ which contain executable anywhere between b_j and e_j (i.e., (in-use, executable) $\in(\texttt{semantics\_of}(s_i, r_j))$ $\longrightarrow \mathbb{P}^{\mu}(s_i, r_j, \texttt{write}) =$ deny.
Any violation of this property indicates a clear type overlap, possibly due to a verification or enforcement failure with respect to the target image's metadata.

$\boldsymbol{p_1}$ requires that $\forall\, r_j \in \mathbb{R}$ in which any part of the region is reserved for a future substage, (i.e., there is some $s_k \in \mathbb{S}$ where $k > i$ and $\texttt{typeof\_region}(r_j, s_k) =$ FUTURE), if $\texttt{typeof\_substage}(s_i) = \boldsymbol{\beta}$ $\longrightarrow \mathbb{P}^{\mu}(s_i, r_j, \texttt{write}) =$ deny.
Any violation of this property also indicates a clear type overlap. However, because

bookkeeping behaviors are generally not directly influenced by the target image, it is unlikely that there are also any accompanying enforcement and/or verification failures.

$\boldsymbol{p_2}$ requires that $\forall\ s_i \in \mathbb{S}$ where

$\texttt{typeof\_substage}(s_i) = \boldsymbol{\delta} \vee \boldsymbol{\pi}$,

if $\texttt{typeof\_substage}(r^*, s_i) = \boldsymbol{b}\ \longrightarrow \mathbb{P}^{\mu}(s_i, r^*, \texttt{write}) = \text{deny}$

Similarly to $\boldsymbol{p_0}$, any violation of this property indicates type overlap, probably due to either a verification or enforcement failure.

$\boldsymbol{p_3}$: suppose that r_s contains the SRAM, r_s contains the BSS, r_h is the heap, and the *global data* is r_g. Let s_p be the earliest substage in the sequence of valid substage transitions (in $\mathbb{Q}^{\mu}$) where $(\texttt{initialized}, \texttt{SRAM}) \in \texttt{semantics\_of}(r_s, s_p)$,

$\forall\ s^* \in \{s \mid \texttt{typeof\_substage}(s) = \boldsymbol{\beta}\ \wedge\ s \text{ is } \textit{before } s_p\} \longrightarrow \mathbb{P}^{\mu}(s^*, r_s, \texttt{write}) = \text{deny}$.

Similar assertions should also hold true for r_h, r_g, and r_i. $\boldsymbol{p_3}$ is similar to $\boldsymbol{p_1}$, in that any violation of it would surface as a type overlap.

$\boldsymbol{p_4}$ requires that, for the range of all *writables* which we define as the function

$W : \mathbb{S} \rightarrow \mathbb{R}$ where $W(s) = \{r \mid \mathbb{P}^{\mu}(s, r, \texttt{write}) = \text{allow}\}\ \longrightarrow$

$$\sum_{\forall\ s \in \mathbb{S}\ \wedge\ \forall\ r=(b^*,e^*) \in W(s)} (e^* - b^*) \quad < \ e - b.$$

5.5.1 BBxM substage and region definitions

My U-Boot SPL policy divides execution into fourteen substages, including the final success and failure substages. A simplified version of the U-Boot SPL's function call graph generated by the IDA Pro disassembler that also highlights the entrypoint, success, and failure substages are depicted in figure 5.24.

A description of the behavior and type of each of the SPL's substages is in table 5.6.

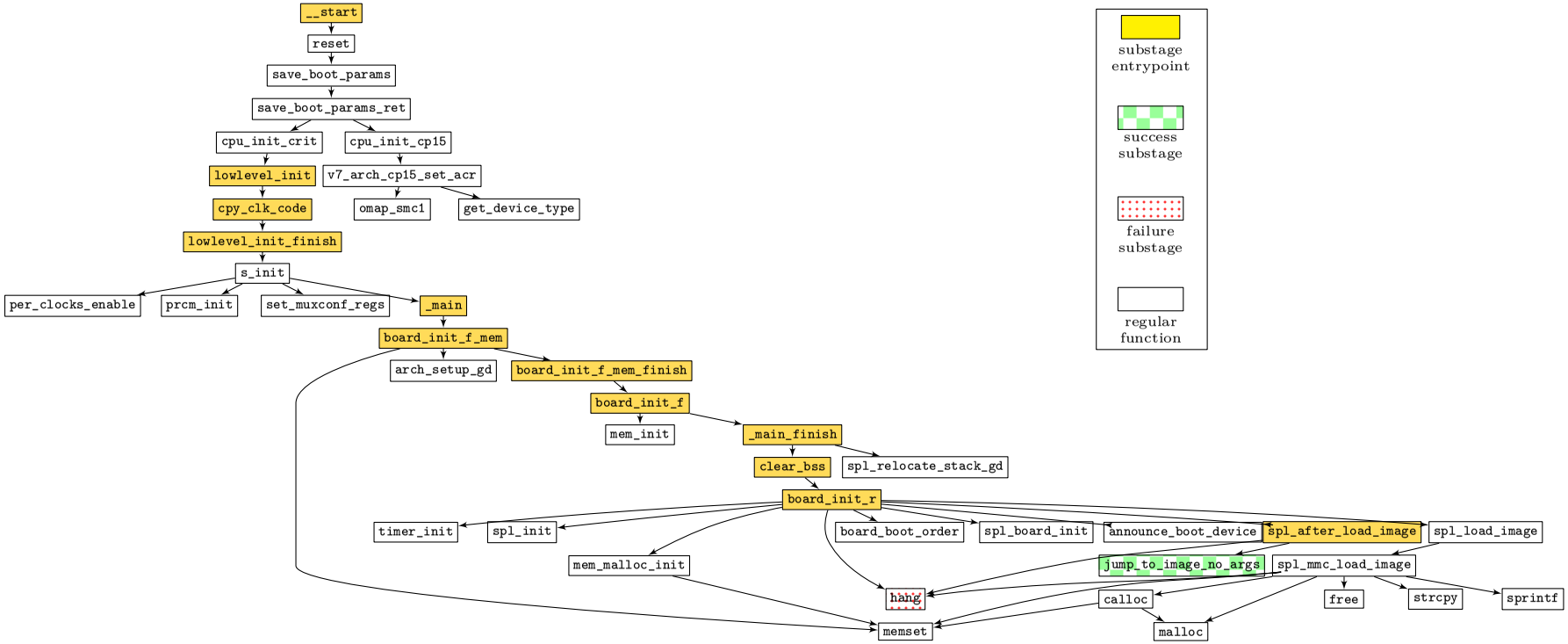


Figure 5.24: Condensed function call graph of the BBxM's U-Boot SPL. Colored-in nodes indicate that the function is also a substage entrypoint.

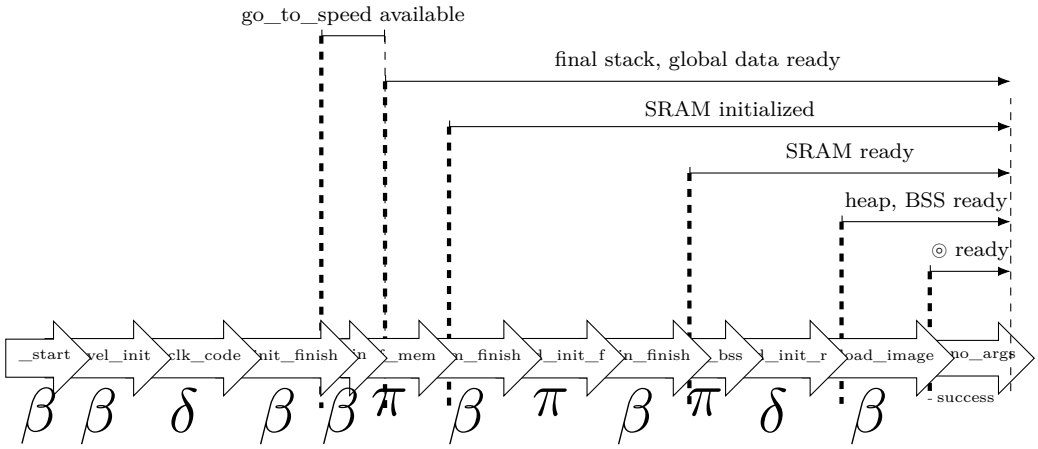


Figure 5.25: Substage sequence defined by BeagleBoard U-Boot policy. Each arrow contains a substage name and points to the policy's subsequent substage. The label below each substage's arrow denotes the substage's type. Important features common to one or more substage are displayed above the substage arrows. ◎ symbol is a shorthand for *target*.

Figure 5.25 depicts a high-level view of these fourteen substages. Memory regions are reclassified during nine of the substage transitions, including a small number of variables (each treated as a tiny region) that are *temporarily* retyped from **b** to **g** (from BOOKKEEPING to GLOBAL) for a single *patching* or *loading* substage. The regions definitions and transitions for subsequent substages are described in table 5.7. Each unique memory map labeled with the substage which brings it into effect can be found in figures 5.26 through 5.28.

5.5.2 BBxM SPL's policy architecture

A high-level (bipartite graph) view of the BBxM's policy is depicted in figure 5.29. This graph also highlights objects/regions that get relocated over the course of execution, more specifically the stack and a special *global data* structure, by appending a counter subscript to the region's name indicating the number of times it has been previously relocated.

Given the RBWAC$^{\mu}$ policy definition (table 5.3 on 101) and this policy instance's region and substage definitions, it should be relatively straightforward to recognize

Table 5.6: BBxM SPL substage definitions

#	entrypoint	type	description
0	`_start`	$\boldsymbol{\beta}$	saves parameters, initializes hardware, moves stack
1	`lowlevel_init`	$\boldsymbol{\beta}$	initializes hardware
2	`cpy_clk_code`	$\boldsymbol{\delta}$	relocates `go_to_speed` function, moves stack
3	`lowlevel_init_finish`	$\boldsymbol{\beta}$	initializes hardware, environment for executing C, moves stack to final location
4	`_main`	$\boldsymbol{\beta}$	initializes hardware, including SRAM
5	`board_init_f_mem`	$\boldsymbol{\pi}$	zeros out region to hold global data
6	`board_init_f_mem_finish`	$\boldsymbol{\beta}$	initializes hardware
7	`board_init_f`	$\boldsymbol{\pi}$	tests SRAM
8	`_main_finish`	$\boldsymbol{\beta}$	more initialization
9	`clear_bss`	$\boldsymbol{\pi}$	zeros out BSS region, initializes heap
10	`board_init_r`	$\boldsymbol{\delta}$	loads target image and header metadata from filesystem on non-volatile storage
11	`spl_after_load_image`	$\boldsymbol{\beta}$	final sanity checks
12	`_jump_to_image_no_args`	*success*	successful boot
-	hang	*failure*	failure to boot

Table 5.7: BBxM SPL region definition transitions

entrypoint	type	figure	regions reclassified at entry
`_start`	$\boldsymbol{\beta}$	5.26a	initial typing
`lowlevel_init`	$\boldsymbol{\beta}$	5.26b	former stack (set by ROM) labeled **r**, new stack labeled **s**
`cpy_clk_code`	$\boldsymbol{\delta}$		no change
`lowlevel_init_finish`	$\boldsymbol{\beta}$	5.26c	stack moved, relocated `go_to_speed` retyped **r**
`_main`	$\boldsymbol{\beta}$	5.27a	stack moved
`board_init_f_mem`	$\boldsymbol{\pi}$	5.27b	global data **p**, stack moved, future image region **f**
`board_init_f_mem_finish`	$\boldsymbol{\beta}$	5.27c	global data **g**, SRAM testing **p**
`board_init_f`	$\boldsymbol{\pi}$		"revision" symbol temporarily **g**
`_main_finish`	$\boldsymbol{\beta}$	5.28a	SRAM testing **p** if in BSS region, **r** otherwise.
`clear_bss`	$\boldsymbol{\pi}$		no change
`board_init_r`	$\boldsymbol{\delta}$	5.28b	BSS now **b**, heap now **g**, a FAT, heap metadata temporarily **g**
`spl_after_load_image`	$\boldsymbol{\beta}$	5.28c	image and image header set **r**

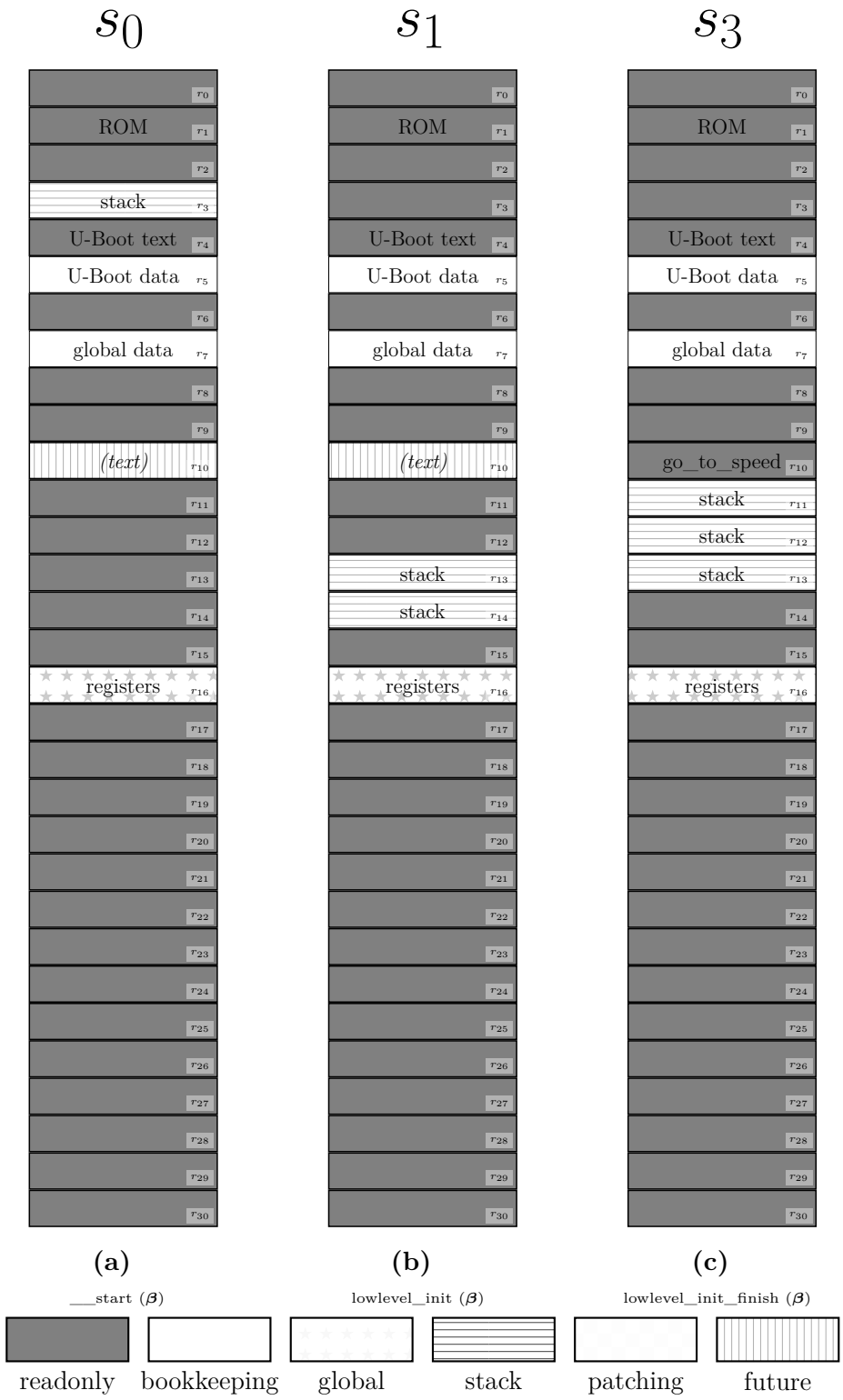


Figure 5.26: BBxM U-Boot SPL substage region definitions (1 of 3). Each region's name is displayed on lower right-hand of its position in the memory map diagram, shading indicates region's type.

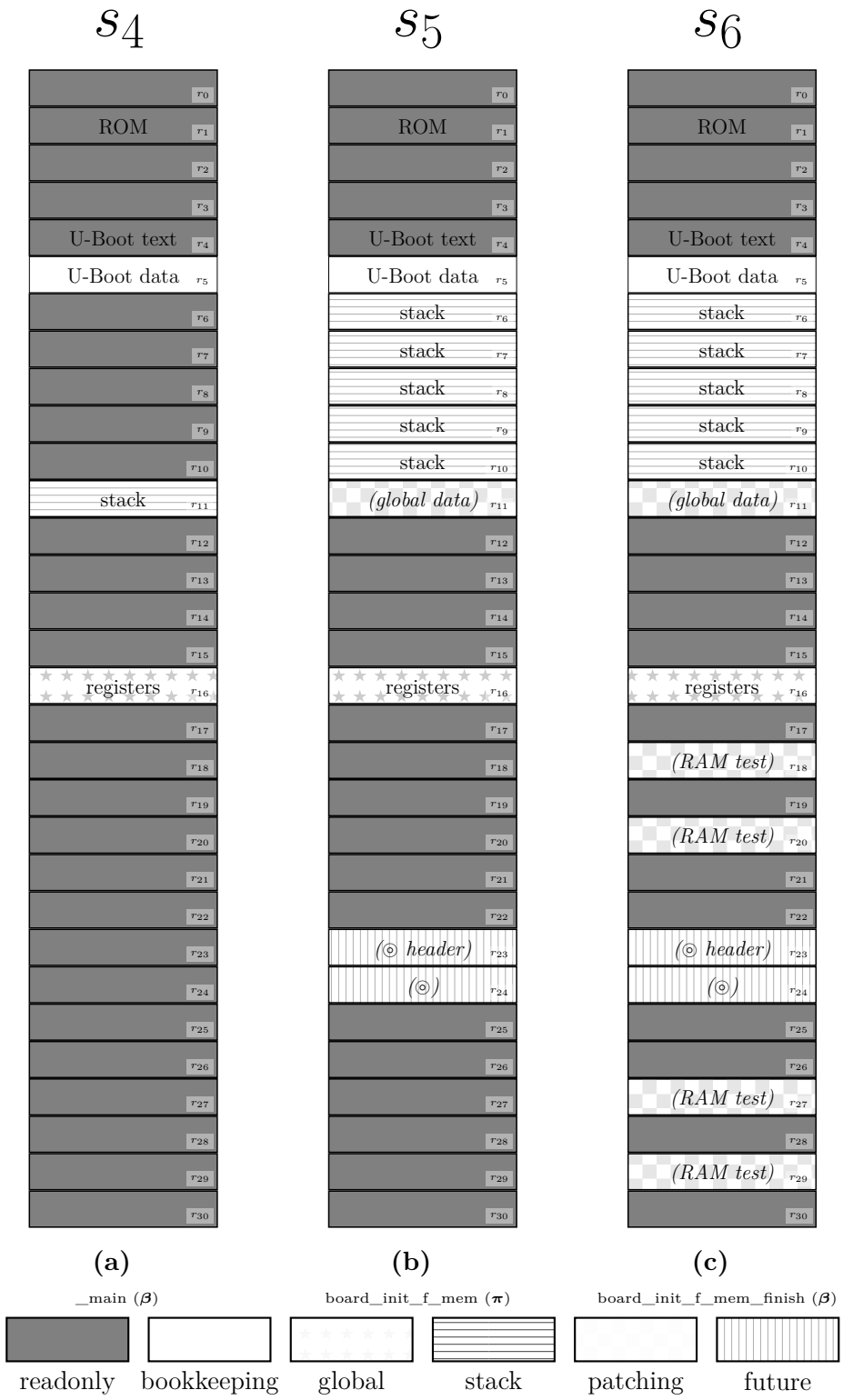


Figure 5.27: BBxM U-Boot SPL substage region definitions (2 of 3). Each region's name is displayed on lower right-hand of its position in the memory map diagram, shading indicates region's type.

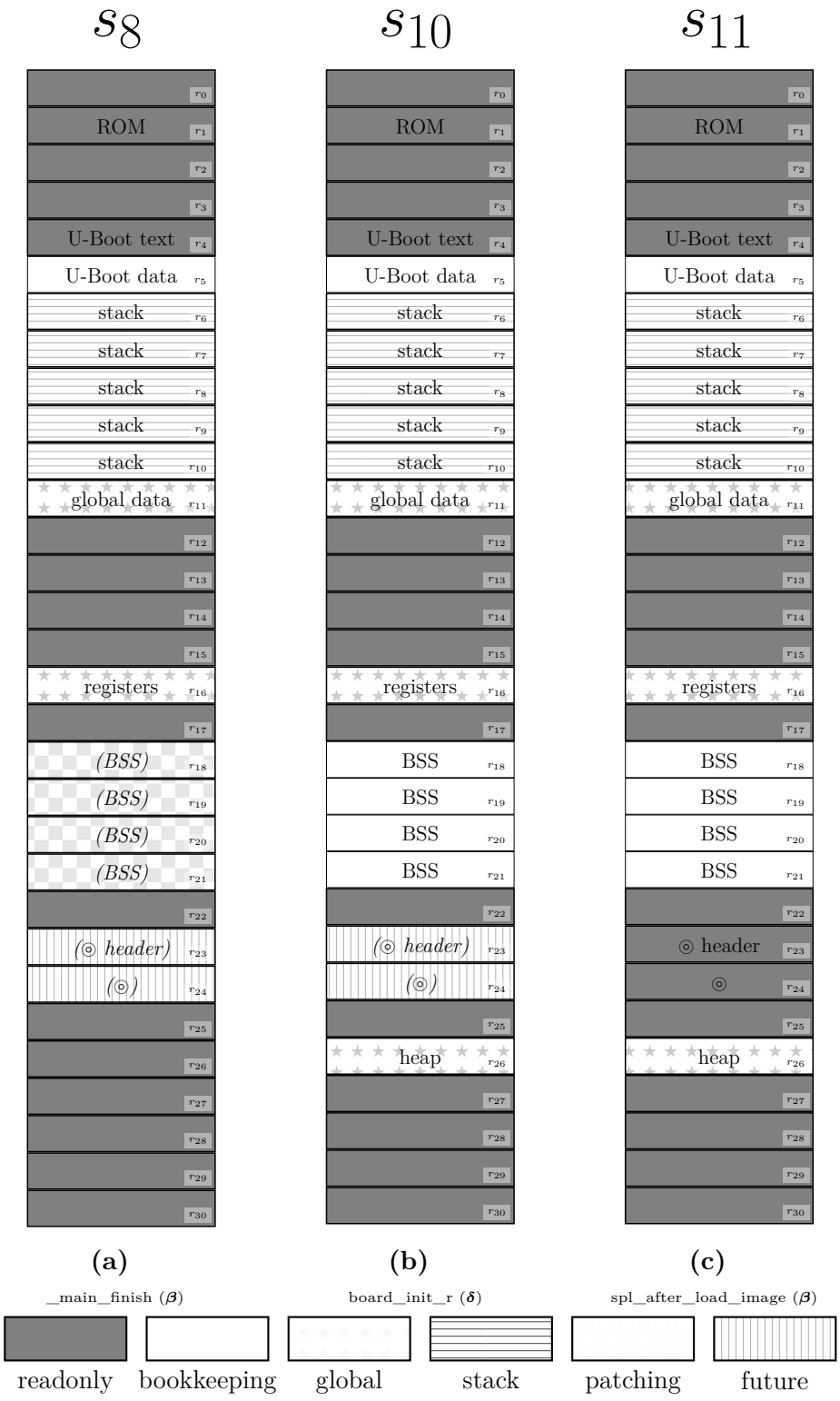


Figure 5.28: BBxM U-Boot SPL substage region definitions (3 of 3). Each region's name is displayed on lower right-hand of its position in the memory map diagram, shading indicates region's type.

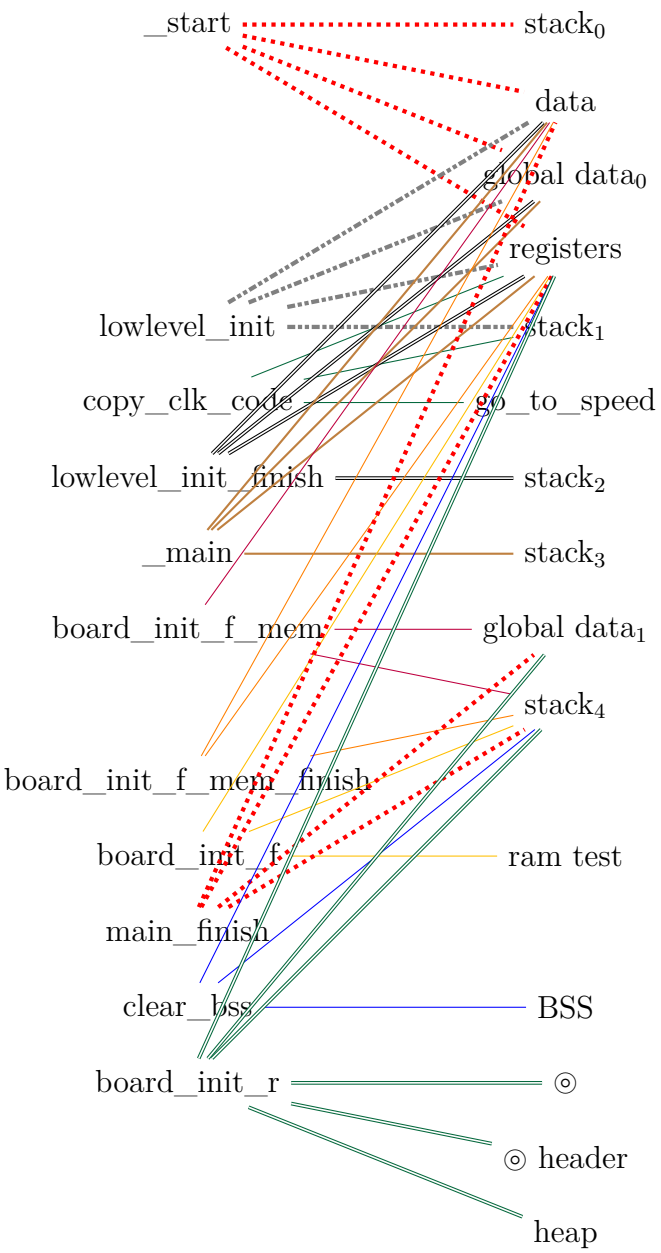


Figure 5.29: Explicitly defined semantic relationships between substages and regions in BBxM's U-Boot SPL policy. Nodes on left represent substages, nodes on the right represent regions (a subscript denotes iteration of relocated region), and an edge between a substage and region denotes write access is allowed.

that property $\boldsymbol{p_0}$ (defined on p. 114) – regions of memory containing in-use code (text) cannot be overwritten – holds for its code and self-relocated code. The region r_5 which continuously holds the bootloader's text is also always typed as **r** (READ-ONLY). The region (r_{10}) to where the loader self-relocates `go_to_speed` is simply *not writable* during any substage in which it may potentially be called, and in particular the `lowlevel_init_finish` substage, s_3. More specifically, $\texttt{typeof\_substage}(s_3) = \boldsymbol{\beta}$, $\texttt{typeof\_region}(r_{10}, s_3) = \mathbf{r}$, and according to $\mathbb{P}^{\mu}$ (as defined in figure 5.10 on page 96), $\nexists\, p \in \mathbb{P}^{\mu} \mid p(s_3, r_{10}, \texttt{write}) = \text{allow}$.

We can also observe how the regions holding the BSS (the consecutive ranges r_{18}, r_{19}, r_{20}, and r_{21}), heap (r_{26}), and global data (r_{11}) are *not writable by bookkeeping stages* until their respective contents have been initialized, i.e., that property $\boldsymbol{p_3}$ holds. For example, the BSS contained in the address range (b_{18}, e_{21}) is not initialized until s_8. It is the case that $\forall\, s_i \in \mathbb{S} \mid i \leq 8 \;\wedge\; \texttt{typeof\_substage}(s_i) = \boldsymbol{\beta}$ (bookkeeping substages that occur *before* the BSS is ready) and $\forall\, p \in \mathbb{P}^{\mu}$:
$\nexists\, r_j = (b_j, e_j) \in \texttt{used\_regions}(s_i)$ where
$\texttt{typeof\_substage}(s_i) = \boldsymbol{\beta} \;\wedge\; (b_j \geq b_{18} \;\vee\; e_j < e_{21}) \;\wedge\; p(s_i, r_j, \;\texttt{write}) = \text{allow}$.
Similar assertions can be made for the heap (also initialized during substage s_8) and the final global data structure (initialized during substage s_6).

$\boldsymbol{p_4}$ (overall reduction in the amount of writable memory at any given point during execution) clearly holds given the fact that there is more than one combination of $r^* \in \mathbb{R}$ and $s^* \in \mathbb{S}$ such that $P^{\mu}(s^*, r^*, \texttt{write}) = \text{deny}$.

We can use similar arguments to establish $\boldsymbol{p_1}$ (that regions reserved for future substage images cannot be written to during bookkeeping phases) and $\boldsymbol{p_2}$ (that loading and patching phases cannot corrupt data intended for bookkeeping) also hold.

This RBWAC policy we constructed for the U-Boot BBxM SPL protects against unintentional bugs and greatly curtails the control maliciously crafted target input (not just a target's image, but also file system metadata) may have over the loader by

heavily reducing the range of addresses that can be written while the target is being loaded, parsed, and interpreted (c.f. the ELF loader weird machine in [191]). This not only exemplifies the *principle of least privilege*, but it also enhances our understanding of what behaviors the executing bootloader is *capable* of exhibiting.

5.6 RBWAC$^{\mu}$ policy language

An RBWAC policy must contain both substage and region definitions. I implemented a simple RBWAC$^{\mu}$ policy interpreter for this thesis that parses policies structured with YAML markup. This interpreter requires two separate YAML files: one that defines named **regions**, and another that defines **substages** and *substage transitions*. Special source code markup (in the form of empty macro statements) is used to define functions which are entrypoints to *failure* substages for convenience.

5.6.1 Region definitions

The YAML-formatted RBWAC policy language defines and labels **regions** of memory in a hierarchical manner, as a forest of trees. Each region defined must be assigned a name and the list of memory addresses it contains. A single region may map to one or more ranges of addresses. This is reminiscent of *tree-structured overlays*, a technique developed in the earlier days of computing when memory was a scarce resource [139]. This technique involved partitioning and loading a program that does not fit in the system's available physical memory so that memory could be temporally shared among the program's separated units. The treatment of memory resources as trees is also a characteristic of modern memory allocaters such as *Vmem* [32], and linker scripts such as those used by the GNU compilation toolchain.

A *region* defined in the policy language can contain zero or more **subregions**, which themselves are also regions but must be located strictly *within the address ranges range of their parent*. Sibling subregions may overlap with respect to each other. A region that has no parents is referred to as a **root region**. All root regions

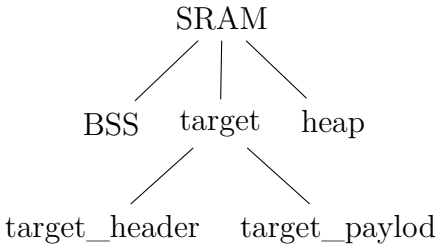


Figure 5.30: Example BBxM U-Boot SPL region definitions for external RAM. Each node denotes a region. Child nodes are named subregions located within their parent's region.

must be assigned unique names, these names act as unique identifiers. Subregions that share a common parent must have unique names with respect to each other so the can be uniquely identified with respect to their parents. For example, if there is a root parent region named **parent** with two subregions $\boldsymbol{c_0}$ and $\boldsymbol{c_1}$, these subregions can be globally referenced with respect to their parent as **parent.**$\boldsymbol{c_0}$ and **parent.**$\boldsymbol{c_1}$.

For example, the case study's bootloader separates the SRAM (its external RAM) into three major areas: the BSS, the target image, and the heap. The target image region can be further divided into two subregions: the target's header and the target's payload. This partitioning of the SRAM can be represented as the tree in figure 5.30. We can uniquely refer to each of the SRAM's subregions with respect to their ancestors: SRAM, SRAM.BSS, SRAM.target, SRAM.heap, SRAM.target.target_header, SRAM.target.target_payload. In accordance with to our language's rules regarding subregions, the SRAM.BSS, SRAM.target, and SRAM.heap regions must all be located within the bounds of the SRAM region. Similarly, the SRAM.target.target_header and SRAM.target.target_payloads must be located within SRAM.target.

However, because U-Boot tests the SRAM by writing and reading bytes from it at a sequence of addresses that overlap with both the SRAM.BSS and addresses in the SRAM that fall outside the BSS, heap, and target, the region layout we have described and illustrated in 5.30 is incomplete. So that the this SRAM testing phase can only modify these few addresses, there are a few additional SRAM subregions

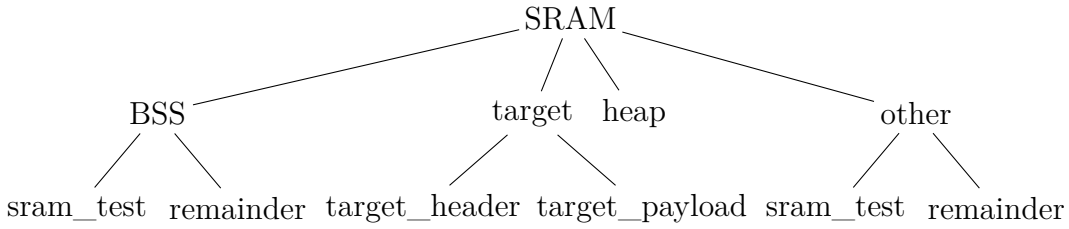


Figure 5.31: More detailed BBxM U-Boot SPL region definitions for external RAM. Each node denotes a region. Child nodes are subregions nested within their parent's region

defined the case study's policy. The actual RBWAC policy's SRAM's regions are more accurately represented by the tree in figure 5.31.

Multiple region definitions among different branches may have overlapping addresses. The RBWAC enforcement mechanism ensures that no two overlapping regions among different branches are *in-scope* during a single substage. The concept of region scope is discussed in section 5.6.2 on page 129 when I discuss substage definitions. This allows us to define multiple region trees to describe a range of addresses that get repurposed during execution. Any tree can be brought into or removed from scope *as a whole* when its corresponding address ranges are repurposed.

As a kind of syntactic sugar, a region's definition may include its initial type that automatically used when introduced into scope. For example, if `SRAM.target.target_payload`'s initial type is defined as READ-ONLY, its respective memory ranges are assigned the ***r*** type when it is brought into scope during a substage transition.

A region with children can be assigned an `include_children` attribute which governs whether its children are automatically brought into scope along with it, this attribute is `False` by default. If `include_children` is `True`, then when the region is brought into scope, all of its children will also recursively be brought into scope. If at any point during this recursive traversal, a subregion with `include_children` explicitly set to false `False` is encountered, the recursive inclusion of children is

discontinued for that branch.

For example, if figure 5.31 `SRAM.target`'s `include_children` attribute is `True`, then `SRAM.target.target_header` and `SRAM.target.target_payload` will automatically be brought into scope when `SRAM.target` is introduced. If `SRAM.BSS`'s `include_children` attribute is `False`, then when `SRAM.BSS` is brought into scope, its two children will remain out-of-scope until explicitly included.

Type is also recursively applied to children when a region is brought into scope unless a child explicitly declares its own type.

For ease of use, a region's address range can be defined using absolute addresses, named constants, ELF symbols, ELF sections, or relative to other named regions with simple arithmetic for convenience (similar to most macro, assembly, and linker script conventions). A region's address range can also be declared to expand to fit the remainder of its parents region that are not covered by any of its siblings. For example, if we directly specify the address ranges of `SRAM.BSS` and `SRAM.BSS.sram_test` when defining the regions in figure 5.31, we can specify that there is sibling region called `SRAM.BSS.remainder` that contains all of the addresses in `SRAM.BSS` that are *not* in `SRAM.BSS.sram_test`. Additionally, a region can also divided into subregions based on an externally produced description – a feature I used to define a separate subregion corresponding to each register listed in the BBxM processor's technical reference.

5.6.2 Substage definitions and region type transitions

In the YAML-formatted policy substage description, substages are defined in the same order in which they should be executed during boot. Substages that denote failure are not included in the file of substage definitions, and are instead simply defined by a special annotation (in the form of an empty macro) at the substage's entrypoint (function definition) in the source code.

A substage definition must include its name (function entrypoint), type ($\boldsymbol{\beta}$, $\boldsymbol{\pi}$, or

$\boldsymbol{\delta}$), and may optionally include region typing and region scope transition statements. Any chunks of addresses that are not within the range of any in-scope region are automatically typed as **r** (READ-ONLY) for this substage.

`new_regions`, `reclassified_regions`, `undefined_regions`, and `allowed_symbols` are *optional* substage attributes. The `new_regions` attribute contains a list of regions that should be brought into scope and typed according to their definition (as specified in section 5.6.1). The first substage that is entered when the bootloader is invoked *must explicitly* bring regions to which it wants to write into scope. For example, our case study's first substage's `new_region` attribute includes the `Registers`, `RAM.stack0`, `RAM.scratch_space`, and `RAM.uboot_data` regions because some of these regions are modified during this first substage.

Any region (and its children if `include_children` is `True`) listed in a substage's `new_regions` remains in-scope through later substages *until* they are included in a later substage's `undefined_regions` attribute. For example, the `RAM.stack0` region is listed in the first substage's `new_regions` attribute so it is brought into scope as this substage is entered. However, it is also listed in the subsequent substage's (`lowlevel_init`'s) `undefined_regions` attribute because the stack is moved before this substage is entered, and thus is removed from scope upon entry to `lowlevel_init`. When a region is listed in a substage's `undefined_regions` attribute, it and its children (regardless of its `include_children` attribute) are recursively removed from scope before that substage is entered, and consequently its address ranges become READ-ONLY until a region that overlaps with those address ranges is brought into scope again.

When a region is brought into scope, its address ranges are typed accordingly for the policy's access decisions. A region's type (and its children) may be changed when a substage is entered by listing it and its new type in the substage's `redefined_regions` attribute. This new type continues to be assigned to the region until the region is

redefined by a future substage (via `redefined_regions`) or is removed from scope (via `undefined_regions`). For example, after U-Boot initializes the BBxM's external RAM during the `_main` substage, the SRAM-based `SRAM.BSS.sram_test` is brought into scope as READ-ONLY (via the `new_regions` attribute) when it enters the subsequent `board_init_f_mem` substage. This region remains typed READ-ONLY until the bootloader is ready to test the external RAM during the later `board_init_f` *patching* substage. This retyping happens as the *bookkeeping* `board_init_f_mem_finish` stage is entered by including (`SRAM.BSS.sram_test`, PATCHING) in `board_init_f_mem_finish`'s `redefined_regions` attribute. Eventually the `board_init_f` *patching* substage is entered which may write to the `SRAM.BSS.sram_test` PATCHING region as it finishes initializing the external RAM.

Pseudocode written in a logical Python-like programming style which calculates region scope changes during substage transitions can be found in figure 5.32. This substage and region definition scheme allows us to *statically* check for conflicting or overlapping region types for a given substage and calculate the set of addresses to where writes are allowed during that substage.

For added convenience, any substage of type $\boldsymbol{\delta}$ or $\boldsymbol{\pi}$ may define an `allowed_symbols` attribute, which contains of list of variables defined in the bootloader's source code that need to be in a writable region for just that substage. The pseudocode in figure 5.35 demonstrates how these temporary allowances are included a policy's decision as whether to allow a write at a particular address to occur during the current substage.

5.7 RBWAC$^{\mu}$ instance and enforcement challenges

Although I have shown that RBWAC$^{\mu}$ is both useful and usable, it is not all sunshine and rainbows. For example, RBWAC$^{\mu}$ (as well as RBWAC) does not generically protect against all memory corruptions within legitimately related substages and regions and it may not catch unintended substage transitions caused by memory

```
def update_region_scope(region_scope,
                        new_regions,
                        undefined_regions,
                        reclassified_regions):
    for r in undefined_regions:
        region_scope.remove(r)
    for r in new_regions:
        region_scope.add(r)
    for (r, type) in reclassified_regions:
        if not region_scope.contains(r):
            raise Exception("Region not in scope")
        else:
            region_scope.update_type(r, type)
    # signal an error if there is a problem with the region definitions
    # such as an interval overlap
    # also assigns the type 'read only' to any intervals that are not
    # explicitly contained within a region
    region_scope.verify_intervals()
    return region_scope
```

Figure 5.32: RBWAC$^{\mu}$ policy language's region scoping pseudocode

corruption. Also, the tools I have developed to both instrument the bootloader and enforce a policy impose a considerable overhead on bootloader execution – the U-Boot SPL stage that normally takes a *fraction of a second* on QEMU-emulated hardware takes *minutes* to run with my instrumentation that uses breakpoints to enforce policy. Finally, it is generally not practical (and potentially unsafe) to run a debugger in production software for enforcing security policies. Nevertheless, this all is merely a prototype to demonstrate RBWAC$^{\mu}$ feasibility.

Heap non-determinism. It is not always possible to statically predict where all data structures will be located at runtime. Even if the bootloader's heap is deterministic, we cannot always depend on a particular heap-allocated data structure to being allocated at the same address every time the bootloader is executed. One way to address the issue of mediating writes to heap-allocated objects is to create separate heaps for data modified by bookkeeping, loading, and patching substages.

```
regions_yml: region_definitions values
region_definitions: region region | ()

region:
     name ":" region_type addresses include csv subregions
   | ()

subregions: "subregions:" region | ()
addresses: "addresses:" address_values
address_values: addr_list | "remainder"
csv: "csv: " STRING | ()
values: "values:"  value_entry value_entry | ()
addr_list: addr_range addr_list | ()
addr_range: "[" addr ", " addr "]"
addr: INTEGER | STRING

region_type:
     "bookkeeping"
   | "future"
   | "patching"
   | "readonly"
   | "stack"
   | "global"
   | ()

include:
    "include_children: True"
  | "include_children: False"
  | ()

value_entry: STRING |  INT
name: STRING
```

Figure 5.33: Grammar for RBWAC$^{\mu}$'s region definition language

```
substages_yml: substage_entrypoint substages_yml | ()

substage_entrypoint:
    name ":" type_decl reclassified undefined symbols new

type_decl: "substage_type:" substage_type
reclassified: "reclassified_regions:" reclass reclass | ()
reclass: name ":" substage_type | ()
undefined: "undefined_regions:"  list | ()
symbols: "allowed_symbols:" list
new: "new_regions:" list | ()
list: name list | ()
substage_type: "bookkeeping" | "loading" | "patching"
name: STRING
```

Figure 5.34: Grammar for RBWAC$^{\mu}$'s substage definition language

```
def is_address_writable(address,
                        current_substage,
                        region_scope):
    if not bookkeeping_substage(current_substage):
        if current_substage.allowed_symbols.contains(address):
            return True
    region = region_scope.region_containing_address(address)
    return allowed(current_substage, region, WRITE)
```

Figure 5.35: Pseudocode implementing RBWAC$^{\mu}$ type policy

Bootloader console environments. Some bootloaders provide mechanisms that allow a user to manually instruct it how and where to load an image to boot and to re-specify these parameters if it fails to boot the target. A linear substage description may not be able to capture such behaviors at a desired granularity, but this possible shortcoming can be addressed by future work as additions to the type system.

Variability in target image storage. Furthermore, many bootloaders allow for flexibility in where and how the target image is stored (whereas my case study focused only on a target stored on an SD card), meaning that further care must be taken to separate bookkeeping from loading logic for each of the supported target locating and loading methods.

Lazy device initialization. Similarly, it may be difficult to capture the behaviors of just-in-time-style (lazy) device initialization at a desirable granularity given the constraints of the RBWAC class described in this thesis. Again, such considerations can be addressed in future work with help from the set of tools I developed for this thesis.

Loading images as libraries, not targets. Some bootloaders, such as those designed for general-purpose computers, may load an arbitrary number of images from various sources (such as the extension ROM that can be found in network cards connected to a PCI-type bus). This poses challenges for writing a useful RBWAC$^{\mu}$ policy instance. Furthermore, these other loaded images themselves may include code to be executed in the context of the main bootloader, which levies yet another challenge in defining and enforcing RBWAC. Nevertheless, although these challenges may not be as easily addressed in by the RBWAC class I presented here, I believe that they are not inherent limitations to RBWAC *as a whole* and that extensions to its type system will better address such issues.

5.7.1 Static analysis of loaders – or –

"How are loaders different from all other software[8]"

It may be useful to employ more static analysis techniques in the future in order to not rely so much on dynamic policy checking. However, in order to understand the changes we may face when applying a static analysis technique to a loader, we must be aware of what makes loaders, and especially bootloaders, different from other software.

All other software may potentially be executed multiple times, whereas upon successful boot, bootloaders execute through completion only once.

All other software may execute in either an unprivileged hardware operating mode or in both a privileged and unprivileged operating mode, whereas bootloaders are only executed in a privileged hardware execution mode.

All other software may assume that all addressable memory is ready for use, whereas bootloaders must tiptoe around what little addressable memory is available as it initializes memory that is not yet available.

All other software has a stack available for them to use, whereas bootloaders may not be able to make this assumption.

Although we may not need our static analysis to be cognizant of these differences if we are only focusing on a loader's write operations, static analysis tools that are not aware of such low-level details and requirements are not as useful.

For example, the well-known and well-used Frama-C analysis framework is designed to work with systems software but it mainly performs symbolic analysis on source code. It does not understand assembly code nor does it naturally understand relocation, and so it is best used to analyze bookkeeping phases. Even so, it requires any portions of the loader's source code written in assembly to be modeled in C. Although Frama-C requires an architectural model of the hardware itself, this information is not enough

[8]Introduction is inspired by the Four Questions (מה נשתנה) traditionally asked by the youngest capable child during a Passover seder meal.

to usefully model a bootloader's behavior because bootloaders are greatly affected by the behavior of their memory-mapped registers.

In order to perform a useful Frama-C analysis of the U-Boot SPL, I had to manually contribute these missing pieces. More details on how I worked with Frama-C to produce a value analysis of U-Boot is documented in section 4.4.1 (page 64), appendix C (page 165), and Frama-C analysis-related source code markup can be found in appendix D (page 185).

Therefore, I developed a Frama-C plugin (source code is available in appendix C.2 on page 173) that processes Frama-C's value analysis results and extracts all information on writes whose destination address has been (at least) partially resolved. Using my plugin, I found that Frama-C was able to meaningfully resolve the destination addresses of 593 out of the 1,596 write instructions present in the U-Boot binary image[9]. Out of these 593 statically-resolved write destinations, only 43 of these resolved destinations have a resolved write destination range greater than four bytes (the size of a word). All of the 43 less-narrowly-resolved writes are located in hardware-specific portions of the source code, making it possible to check their behavior against the processor's technical reference. These 593 statically-resolved write instructions ultimately make up about 13% of all memory writes that occur during a successful execution of the bootloader.

Because U-Boot's substages are implemented as non-returning continuations and includes the call stack from which the result was calculated, it is possible to determine the substage during which each of these writes occur. Figure 5.36 shows an example of some of the output from my Frama-C write destination analysis plugin formatted as:

"`[dst] [<start_addr>,<end_addr>] <left_value> in <source_line> .. -><callstack>`",

such that,

[9]Although Frama-C does not have an understanding of U-Boot as a compiled binary, as it operates at a source code level, using U-Boot's debugging symbols (and gdb's `info line`) command) it is possible to uniquely identify which write instruction corresponds to a given line in the source code.

```
[dst] [0x48002264, 0x48002266] *((unsigned short volatile *)1207968356) in
↪  /tmp/tmpQXzkhn/board/ti/beagle/beagle.c:526 ..
↪  ->frama_go->lowlevel_init->cpy_clk_code->lowlevel_init_finish->s_init-
↪  >set_muxconf_regs

[dst] [0x490580D0, 0x490580D4] *((unsigned int volatile *)reg) in
↪  /tmp/tmpQXzkhn/drivers/gpio/omap_gpio.c:71 ..
↪  ->frama_go->lowlevel_init->cpy_clk_code->lowlevel_init_finish->s_init-
↪  >_main->board_init_f_mem->board_init_f_mem_finish->board_init_f->⌋
↪  mem_init->do_sdrc_init->get_board_mem_timings->get_board_revision->⌋
↪  gpio_direction_input->_set_gpio_direction

[dst] [0x4020FED8, 0x4020FEDC] gd->arch.omap_boot_device in
↪  /tmp/tmpQXzkhn/arch/arm/cpu/armv7/omap-common/boot-common.c:105 ..
↪  ->frama_go->lowlevel_init->cpy_clk_code->lowlevel_init_finish->s_init-
↪  >_main->board_init_f_mem->board_init_f_mem_finish->board_init_f->⌋
↪  _main_finish->clear_bss->board_init_r->spl_board_init->⌋
↪  save_omap_boot_params

[dst] [0x80000400, 0x80000404] *((unsigned int volatile *)(addr + 1024U))
↪  in /tmp/tmpQXzkhn/arch/arm/cpu/armv7/omap-common/mem-common.c:39 ..
↪  ->frama_go->lowlevel_init->cpy_clk_code->lowlevel_init_finish->s_init-
↪  >_main->board_init_f_mem->board_init_f_mem_finish->board_init_f->⌋
↪  mem_init->do_sdrc_init->write_sdrc_timings->mem_ok
```

Figure 5.36: Example output generated by Frama-C write destination analysis plugin containing the call stack and possible write destination ranges for various memory writes performed in the U-Boot source code in the form of:
`[dst] [<start_addr>, <end_addr>] <left_value> in <source_line> .. -><callstack>`

- `start_addr` and `end_addr` mark the range of destination addresses to which this statement may write
- `left_value` is how Frama-C visually represents this resolved destination range (as a dereferenced left-hand-side value)
- `source_line` is the location of this write in the source code
- `callstack` is the call stack (separated by "->") from which this value was calculated. Note that first function on this list is always `frama_go`, a function that contains a C translation of U-Boot's entrypoint.

For any write instruction destination that cannot be statically resolved, it is easy

to imagine supplementing this static analysis with a compilation stage to inject range checks that guard that particular write.

5.7.2 Rearchitecting systems to address RBWAC challenges

Although rearchitecting a system from the ground up is not a particularly practical solution, certain architectural modifications could help us better apply an RBWAC policy to its bootloaders.

For example, if we could set memory access permissions at a word granularity as proposed in the Mondrian memory protection paper [227], the bootloader itself could setup region access permissions during substage transitions. The proposed SAFE system architecture which tags each addressed word with metadata, dynamically-defined "authorities", and call gates that allow transitions between these authorities (beyond user and system modes) for privilege isolation, may be able to cleanly enforce substage transitions and access controls [52].

These proposed architectural changes are forms of **tagged architectures** that allow for "self-identifying data" and include instruction sets that incorporate these data tags [77]. Tagging is not completely absent from commonplace modern architectures. For example, page tables used for memory virtualization have metadata that are essentially a page-granularity tags. In fact, *grsecurity* made use of otherwise-unused page metadata to implement the equivalent of a no-executable bit for x86-based hardware that lacked hardware-based memory execution protections [196].

5.8 Other applications of RBWAC

Up to this point, RBWAC has been exhibited as a system designed to reduce a bootloader's attack surface by enforcing memory accesses based on how the bootloader is *intended* to operate. Yet, there are other potential uses for RBWAC and its instrumentation suite. As demonstrated when I described how I developed a policy for U-Boot, RBWAC can be used to extract information on how a bootloader behaves

and gain an understanding of its overall control flow. Therefore, one can use it to build and test models that represent how they believe the loader functions or how they intend it to function. These same techniques and tools can be applied to all types of software for blackbox and whitebox testing as well as reverse-engineering. They can also be used as a general testing/fuzzing aid, to catch more subtle bugs (e.g., off-by-one errors) that surface as the target software processes malformed input.

6

Future directions

Although my thesis covers a lot of ground, there is still plenty left to explore. Throughout this thesis, I aimed to achieve a balance between practicality and academic inquiry; the thesis, therefore, opens many questions in both realms. It acts as an exploratory step towards trustworthy loaders and bootloaders, and demonstrates that is feasible to build a stronger chain-of-trust based on simple properties and in a manner that requires minimal changes to legacy code. Yet, there are many questions, ranging from theoretical to practical, that must be addressed before we can build verifiably trustworthy loaders. Throughout the course of working on this dissertation, I encountered an number of tangentially-related questions worth exploring which I also highlight in this chapter.

6.1 Theory

Loaders act like machines – typically written in Turing-complete programming models – that ultimately translate a Turing machine its reads in as input to a Turing machine that becomes executed in its place, sometimes performing self-modification along the

way. Although there is research on modeling and verifying self-modifying code, most of it has either focused more generally on self-modification or on self-modification observed in malicious software. Therefore, I see value in extending this work to focus more specifically on the kind of self-modification performed by loaders and linkers – a form of self-modification that aims to *preserve* (as opposed to *obfuscate*) the semantics of the target image. Another important question is whether a loader needs to be Turing-complete, or if it can be expressed in sub-Turing models that lend themselves to easier verification. Yet another potential direction one could explore is to develop a framework of Hoare-style reasoning, similar to Cai, Shao, and Vaynberg's *Certified Self-Modifying Code* [46], to model and verify behaviors, such as relocation, that are common across all types of loaders.

6.2 Performance

Although my thesis did not focus much on performance issues, performance must be addressed if the techniques I introduced are to be adopted. I have addressed some performance concerns with potential improvements in section 5.7 (page 131), however it is worth reiterating here that future work can endeavor to reduce runtime performance costs via a combination of static analysis and injecting runtime policy checks instead of relying on external instrumentation. It may also be worth considering potential hardware extensions as a more general solution.

One also may want to study techniques from loaders past – more specifically the overlay method many older loaders employed that allowed the system to execute a binary image larger than the amount of memory available. Overlays were managed by an *overlay manager* which was a paging/dynamic linking-like mechanism statically built into a binary's image that copies binary's own unmapped segments into memory when needed by overwriting unneeded segments. It may be worthwhile to build a bootloader that, in the same manner of an overlay manager, incrementally loads

pieces of itself as needed while discarding regions with which it is finished. This would naturally limit the amount of available code/data[1] and thus the loader's own capabilities at any given time.

6.3 "It's not a bus, it's a network![2]"

Memory regions can naturally be represented as nodes on a tree whose layout and relationships represent the address ranges enclosed by each region. What may not be surprising is that a processor's physical addresses can be modeled the same way. When a processor reads an address's value, there are electronic mechanisms in place – buses – that dictate the semantics of this address and provide the CPU access to the addresses' value. When we dig deeper into the true meaning of a physical address, we find that this bus, often taken for granted by software engineers due to its transparency, is a complex network in and of itself. Addressing may *seem* simple from the point of view of a software developer because instruction sets present a consistent interface when working with the abstraction of a *physical address*, however this simplicity is merely an illusion. Hardware buses will continue to grow more complex, especially with the ever-increasing popularity and complexity of SoCs (system on a chips). SoCs contain a conglomeration of discrete IP (intellectual property) components connected by various configurable buses and networking protocols such as the Advanced Microcontroller Bus Architecture (AMBA), the Advanced Extensible Interface (AXI), and the Wishbone Bus – all of which seem to be relatively unknown within the computer security community. Given the large body of existing work on networks and distributed systems, we should try to apply this body of knowledge to these buses so we can build safer systems.

[1] C.f. Linux's marking of some sections of code as init-only, discarded them at the end of system initialization.

[2] Exclaimed Sergey Bratus during a lecture Travis Goodspeed gave to Sergey's class on embedded devices.

6.4 Other tangentially-related research questions

What happens on the other size of a *physical address*'s abstraction barrier is dependent on the hardware itself, but an address access request can potentially traverse multiple types of buses, each of which potentially have their own private physical, link, and networking layers – all of which imply that buses are managed by embedded processors. Buses *are* networks, complete with internal nodes and edges, they are not merely a network's edges. Not only that, but their routers are *smart* and *stateful*. Many of the more-recently discovered UEFI-related vulnerabilities (especially those related to x86's System Management Mode) take advantage of these networks being misconfigured. I believe that there is value is more deeply exploring the relationships between buses, networks, and routing of physical addresses.

There is another layer of complexity that those who work with systems software rarely need to consider – physical addresses represent how the processor (and *software developer*) interact with this network, but the primary processor is *not* the only node that is allowed to initiate read or write requests on this network. Other hardware components (that have their own private processor, firmware, and volatile memory) may themselves interact with other endpoints (other hardware or the processor itself) over the same network. These endpoints may have a completely different view of the network, with an orthogonal addressing scheme imposed by the bus itself. This not only makes it harder to reason about the network, but allows for the possibility that different endpoints have conflicting views of the network. These conflicts should be formally studied, and bootloaders may be a natural entrypoint to such research since it is typically the bootloader that configures any non-hardwired topology. It may be useful to pull from the well-established field of distributed systems in to identify potential issues, especially as these buses become faster, more powerful, and more complex.

7

Conclusion

7.1 Concluding thoughts

Specifying formal properties of loaders, and especially bootloaders, may be underappreciated, but loader failures are certainly conspicuous when things go wrong. Not only is it important to have correctly-implemented and secure loaders for our own sanity, it is also important that loaders live up to their position as a foundation on which we build our system's trust. My thesis illustrates why current loaders are not as trustworthy as they are treated by documenting the many ways loaders have failed to act in a trustworthy way. I then strive to improve the trustworthiness of loaders by developing the necessary tools to instrument a loader, analyzing its behaviors, and proposing a novel typing-based method that allows us to semantically model a loader's intended behavior – one that targets and mediates a loader's memory accesses in a meaningful way. Finally, I designed and implemented languages and mechanisms that allow for loader memory accesses policies to be defined and enforced – ultimately demonstrating the feasibility and applicability of my method of modeling and media-

tion by applying it U-Boot, a well-known and extensively used bootloader.

My thesis starts by describing the role loaders play in a system – which act as a necessary bridge between a system's hardware and software ecosystems, and exist in the realm of the application binary interface (ABI). We have always implicitly trusted loaders to operate correctly and in an benign manner. However, an increasing amount of *explicit* trust is now being placed on bootloaders throughout this past decade as an increasing number of loaders have been tasked with judging the trustworthiness of the binary images they load. Examples of this explicit trust include *Trusted Boot* and *Measured Boot*, and their designs hardly address the internals of the bootloader itself[1]. My thesis addressed this pressing need to design and implement *provably* trustworthy loaders by proposing a region-based type system that meaningfully govern a loader's behaviors.

In order to motivate and inform my type system's design, I extensively discuss weaknesses exhibited by real-world loaders across a variety of devices (in chapter 2 and provide a table of publicly reported vulnerabilities (CVEs) in appendix B). I then discuss many of the challenges that make loader analysis and verification difficult in chapter 3. Chapter 4 addresses some of these challenges using tools I implemented and discusses how these tools were used to extract and describe the U-Boot SPL bootloaders behaviors as it executed the BeagleBoard-xM (BBxM), a case study of how these tools operate in practice.

The meat of my thesis's proposed policy framework is presented in chapter 5, which formally describes a typed region policy framework called RBWAC that is capable of modeling a loader's intended behaviors – distinguishing phases when the loader is performing internal/bookkeeping tasks, from phases when the loader is copying a subsequent substage to memory, and from those when the loader is patching a subsequent stage. By explicitly separating these three types of intended behaviors,

[1]Even the well-known UEFI bootloader specification only minimally addresses bootloader internals.

we are able to better build loaders that do not suffer from the kinds of weaknesses loaders exhibit (as discussed in chapter 2). My policy discussion in chapter 5 presents three policy definitions of increasingly complexity, two for a toy-sized, but realistic, bootloader, and the third, most complex example, dealt with the BBxM's U-Boot SPL boot stage. This demonstrates both the feasibility and usefulness of my thesis's proposed policy framework. I then conclude this policy chapter with a discussion of other possible uses for my thesis's tools and policy framework. Finally, in chapter 6, I present a few ideas of how this work can evolve and be extended, as well as other possible research questions inspired by this thesis.

7.2 Final thoughts

My thesis has merely scratched the surface of to build safer bootloaders. There are may potential directions this work can proceed, but, before I continue with this work, it is important that I identify which of these ideas resonate with the wider research community and industry. I would also like to see which components of this work can be adopted into existing infrastructures and engineering environments. I hope that those from academia and industry who have taken time to study this work will consider its possibilities respond with their own interpretations and derivations.
In the spirit of open access, I have made the source code for my tools available at: https://github.com/bx/bootloader_instrumentation_suite

The U-Boot source code, complete with my patches is available at: https://github.com/bx/u-boot-extended

My patched version of *QEMU* that is fully compatible with U-Boot and is capable of tracing watchpoints is available at https://github.com/bx/qemu-linaro-patched

A patched version of *openocd* that supports BBxM JTAG debugging is available at

https://github.com/bx/openocd-patched

A

RBWAC-inspired U-Boot discoveries

Throughout my process of working with QEMU and U-Boot to develop an RBWAC policy, I encountered many interesting quirks and (by)products of the its development process that I discuss this chapter.

A.1 BBxM hardware and documentation

The BBxM has been around since 2010, and the SoC around which it was built, the TI (Texas Instruments) am37x, has been around just as long. Despite their age, I have noticed various quirks in both the hardware and its documentation that have withstood the test of time as well as a number of revisions.

A.1.1 Documentation's register tables

TI publicly released extensive documentation for the am37x, over 3000 pages long, which included tables that identify many (if not all) of its memory-mapped registers.

My register table scraping tools that were run on this document found multiple (20+) instances where these tables suggest that two disparate registers are located at the same physical address. These are likely bugs in the documentation.

A.1.2 ARM TrustZone security extensions

TI's documentation uses the term "public" to describe the processor's on-chip-ROM-based bootloader, an on-chip-RAM-based stack, and other bootloading-related objects. What is not initially apparent is *why* the adjective *public* is being used in such contexts. However, when we read between the documentation's lines, we may find ourselves pondering on why its entrypoint which is not where it should be for an ARM processor (either at 0x00000000 or 0xFFFF0000) but instead is at 0x14000. Perhaps the SoC design has some abnormalities, or perhaps there *is* a *non-public* portion of the ROM whose entrypoint is at an expected address.

It appears that am37x makes use of its ARM processor's TrustZone security extensions and configures these security extensions so that a portion of its on-chip ROM (which likely starts at address 0x00000000) is not-directly visible from the so-called public ROM and, consequently, any standard hardware debugging tools. Indeed, TrustZone-specific ARM instructions are present in the BBxM's U-Boot implementation (and unsurprisingly, they are written by a TI employee[1]). Indeed, it turns out that the SoC's documentation contains some information on how to interact with this non-public portion of the boot ROM, although its not easy to find, as it appears as a side-note displayed in a box marked *caution* located in the chapter on device initialization.

It is often the case that information about a microprocessor's internal boot ROM is not made public, however it is *almost* surprising that TI's documentation does not directly state the existence of the am37x's "secure" boot ROM given the large number of details they provide on the chip's boot ROM. Unfortunately, it is not uncommon

[1] U-Boot commit 45bf05854bc94ed8bae9e9114292895b990327ea

for chip manufacturers to incorporate black boxes into a chip's ROM that act as highly privileged and trusted interfaces (e.g., system management mode and the Intel Management Engine).

This tale about the BBxM's "secret" use of TrustZone brings up a few thoughts worth pondering, in particular:

- How do such unknown-unknowns affect the resulting strength of a security policy?
- How do we architect a security policy that incorporates any little knowledge we have of these implicitly-trusted black-box interfaces (in this case provided by the chip manufacturer)?
- Certain hardware-based security mechanisms, such as TrustZone, are sort of like hardware fuses – they are mechanisms can only be used once. If multiple stakeholders in a given manufacturing chain want to implement systems that make use of these security extensions, the stakeholder who is earliest in the manufacturing get first dibs, ultimately locking other stakeholders out from making direct use of these security extensions. Therefore, such security extensions will more likely be used by the chip manufacturer then an OEM and especially then the end-user.

A.2 QEMU

QEMU's implementation of a BBxM emulator includes an implementation of the device's on-chip boot ROM. This implementation comes in the form of an array of bytes which represent how the boot ROM appears in memory. Figure A.1 shows a small portion of this byte array found in the QEMU source code.

Although this implementation technique is interesting in and of itself, what is also interesting is that the QEMU's original ROM implementation is not compatible with all of the bootloaders that directly run on hardware. QEMU's ROM implementation works with x-loader – the original initial bootloader used by the BBxM – not with

```
1     0x08, 0x0c, 0x40, 0xe2, /* sub r0, r0, #2048   @ 2kB UND stack */
2     0xd3, 0xf0, 0x21, 0xe3, /* msr cpsr_c, #0xd3   @ enter SVC mode */
3     0x00, 0xd0, 0xa0, 0xe1, /* mov sp, r0   @ 23kB left for SVC stack */
4     0xdf, 0xf0, 0x21, 0xe3, /* msr cpsr_c, #0xdf   @ enter SYS mode */
5     0x40, 0x04, 0xa0, 0xe3, /* mov r0, #0x40000000 @ r0 -> vba */
6     0x05, 0x09, 0x80, 0xe2, /* add r0, r0, #0x14000 */
7     0x10, 0x0f, 0x0c, 0xee, /* mcr p15, 0, r0, c12, c0, 0 */
8     0x60, 0x00, 0x80, 0xe2, /* add r0, r0, #0x60  @ r0 -> monitor vba */
9     0x30, 0x0f, 0x0c, 0xee, /* mcr p15, 0, r0, c12, c0, 1 */
10    0x1c, 0x00, 0x10, 0xe5, /* sub r0, [r0, #1c] @ r0 -> booting
      ↪  parameter struct */
11    0x01, 0xf0, 0xa0, 0xe1, /* mov pc, r1 */
```

Code fragment from QEMU linaro's hw/misc/omap3_boot.c

Figure A.1: Sample of QEMU's BBxM ROM implementation – instructions embedded in an array of bytes

the U-Boot SPL. What makes x-loader compatible with QEMU's emulated BBxM ROM but not the U-Boot SPL is that QEMU's ROM stores its parameter data (as shown in figure A.2) within the boot ROM's own read-only region even though it is unlikely that the hardware's on-chip boot ROM is even capable of storing parameters in ROM. x-loader is perfectly happy to read its boot parameters from a region that is not technically writable, whereas U-Boot refuses to interpret its parameters if the are not located within the device's on-chip RAM (as we can see in its source code in figure A.3). I resolved this issue by patching QEMU so that its ROM implementations stores its parameters in (its emulated) on-chip RAM, thus appeasing U-Boot.

These seemingly subtle differences in implementation can (and do) have a real impact. They are not just "benign" example of *red pills* because they can also impact behavior in hard-to-predict and hard-to-ignore ways. And yet, in this particular instance, we cannot firmly say that QEMU's ROM implementation is incorrect – although it makes the most sense that their emulation stores such data in writable memory, the chip's technical reference says nothing specific about where these parameters may be located. Likewise, U-Boot's sanity checks are extremely reasonable – it makes sense to verify whether its parameters are stored in writable memory.

Table 25-47. Booting Parameter Structure

Offset	Field	Size [Bytes]	Description
0x00	Booting message	4	Last received booting message
0x04	Current booting device	1	Code of device used for booting: 0x01: XIP memory 0x02: NAND 0x03: OneNAND 0x04: DOC 0x05: MMC/SD2 0x06: MMC/SD1 0x07: XIP memory with wait monitoring 0x10: UART 0x11: HS USB Other: Reserved
0x05	Reserved	1	Reserved

Table 25-47. Booting Parameter Structure (continued)

Offset	Field	Size [Bytes]	Description
0x06	Reset reason	1	Current reset reason bit mask (bit = 1, event present): [0]: Power-on reset (POR) [1]: Global software reset [2]: Reserved [3]: Reserved [4]: WDT2 reset [5]: Reserved [6]: External warm reset [7]: VDD1 voltage manager reset [8]: VDD2 voltage manager reset [9]: ICEPick™ reset [10]: ICECrusher™ reset Other bits: Reserved
0x07	CH flags	1	Configuration header sections flag. Each section is described by 1 bit. A set bit indicates that the section was executed: [0]: CHSETTINGS [1]: CHRAM [2]: CHFLASH [3]: CHMMC Other bits: Reserved
0x08	Device descriptor	4	Pointer to the device descriptor structure. This pointer is required when current booting device driver functions are called.

Figure A.2: Layout of parameters passed from the on-chip boot ROM to the subsequent bootloader as published in the am37x technical specification [202]

```
1 void save_omap_boot_params(void)
2 {
3         u32 boot_params = *((u32 *)OMAP_SRAM_SCRATCH_BOOT_PARAMS);
4         struct omap_boot_parameters *omap_boot_params;
5         int sys_boot_device = 0;
6         u32 boot_device;
7         u32 boot_mode;
8
9         if ((boot_params < NON_SECURE_SRAM_START) ||
10            (boot_params > NON_SECURE_SRAM_END))
11                return;
```

Code fragment from U-Boots arch/arm/cpu/armv7/omap-common/boot-common.c

Figure A.3: U-Boot sanity checking the location of the parameters passed to it from the boot ROM

A.3 U-Boot

U-Boot is an open source project that has been in existence since 1999. U-Boot, as it stands today, is a result of the branching and merging of fairly independent incarnations (often targeted to a specific architecture) of U-Boot trees. In fact, U-Boot has developed a formal process of tracking these so-called "custodian trees"[2]. Because of this, U-Boot's source code harbors a variety of coding styles and design patterns that address similar engineering tasks. There is also a large amount of code reuse in U-Boot, including hardware-specific code that is reused by related hardware. This code reuse is aided by preprocessor macros, a complex build environment, static libraries, and what appears to be copy-and-paste development techniques. One example of a possible copy-and-paste bug found in the source code is highlighted in figure 5.16 on 105 where I note inaccuracies in the markup that explains its relocation process in section 5.4.1 on page 102.

A.3.1 Code bloat

These development techniques also result in both dead and unnecessary code which can be problematic given the scarcity of RAM available early in the boot process of some hardware. I personally ran into size/bloat issues while trying to build a version of the SPL with debugging symbols.

We can get an idea of how much bloat is present in U-Boot my comparing the sizes of the U-Boot SPL generated for the BBxM (`MLO-uboot`) and the equivalent OMAP3-specific x-loader[3] bootloader image (`MLO-xload`):

[2] http://www.denx.de/wiki/view/U-Boot/CustodianGitTrees#Philosophy_of_custodian_trees

[3] x-loader is TI's OMAP chip family specific bootloader and was what was eventually merged into U-Boot to support the BBxM.

```
> ls -s1h
total 76K
 52K MLO-uboot
 24K MLO-xload
```

The U-Boot SPL image is nearly **twice** the size of x-loader's image!

One chunk of unnecessary code in U-Boot's SPL is the function `cpy_clk_code` which relocates the `go_to_speed` function. According to comments in the source code, `go_to_speed` is relocated to a region of memory that is faster to access. More specifically, the comment above `cpy_clk_code`[4] which says the following:

```
/*************************************************
 * cpy_clk_code: relocates clock code into SRAM where
                 its safer to execute
 * R1 = SRAM destination address.
 *************************************************/
```

`go_to_speed`[5] is annotated with the following comments:

```
/*****************************************
 * go_to_speed: -Moves to bypass, -Commits clock dividers, -puts dpll
        at speed
 * -executed from SRAM.
 * R0 = CM_CLKEN_PLL-bypass value
 * R1 = CM_CLKSEL1_PLL-m, n, and divider values
 * R2 = CM_CLKSEL_CORE-divider values
 * R3 = CM_IDLEST_CKGEN - addr dpll lock wait
 *
 * Note: If core unlocks/relocks and SDRAM is running fast already it
```

[4]Found in U-Boot's arch/arm/cpu/armv7/omap3/lowlevel_init.S
[5]Also, in U-Boot arch/arm/cpu/armv7/omap3/lowlevel_init.S

```
 * gets confused. A reset of the controller gets it back. Taking away
 * its L3 when its not in self refresh seems bad for it. Normally,
 * this code runs from flash before SDR is init so that should be ok.
 ****************************************/
```

In the case of the BBxM, `cpy_clk_code` unnecessarily relocates `go_to_speed` because it is relocated into an address ranged backed by the same memory (and thus the same speed) as its original location. Additionally, `go_to_speed` is never executed by the SPL – a property that can be statically deduced.

A.3.2 Undefined registers

Using my PDF register table parsing tool I generated a RBWAC policy that specifically targeted registers, to ensure that read-only registers are not written to by the bootloader and that the bootloader does not write to non-existent registers. Because of this, I discovered three memory addresses that U-Boot believes to be mapped to writable registers but are not listed in the device's technical reference. It is not clear as to whether these registers are merely missing from the technical reference or that these write accesses are indeed subtle (potentially code reuse-related) bugs. Regardless of where the truth lies, given that this code that writes potentially undefined locations has been unchanged since 2009 (or 2008 if we take into account its presence in the x-loader implementation), we can see how RBWAC can also act as a useful software testing and auditing tool.

A.3.3 Typing issues

Frama-C was able to identify multiple type violations that were not otherwise known, which are discussed in section C.1.4 (page 170).

A.3.4 Linkmap scripts and tricks

U-Boot implements a custom link map script which instructs the compilation toolchain's static linker in how to construct and order sections in its generated ELF image. This script manipulates the static linker into storing select static variables, sorted alphanumerically by symbol name, in a special separate section they call `.u_boot_list`. This trick allows for important hardware-specific data structures, such as those that hold hardware information and driver function pointers, to be declared anywhere in the codebase, but for the data structures themselves to be coalesced (by the linker) into a single ELF section that contains them in a series of doubly-linked lists. This is implemented in a linkmap script (arch/arm/cpu/armv7/omap-common/u-boot-spl.lds) by instructing the linker to sort and combine all sections whose names match `.u_boot_list*_i2c_*` into a single section named `.u_boot_list` and locate the result in the region defined by `.sram` using the following statements –

```
.u_boot_list : {
        KEEP(*(SORT(.u_boot_list*_i2c_*)));
} >.sram
```

Where `sram` is defined in the same file as:

```
MEMORY { .sram : ORIGIN = CONFIG_SPL_TEXT_BASE,\
                 LENGTH = CONFIG_SPL_MAX_SIZE }
```

Entries in a linkmap list are declared and traversed via macros (located in include/linker_lists.h) such as:

```
#define ll_entry_declare(_type, _name, _list)                          \
        _type _u_boot_list_2_##_list##_2_##_name __aligned(4)          \
                        __attribute__((unused,                         \
                        section(".u_boot_list_2_"#_list"_2_"#_name)))
```

B

Loader-related vulnerabilities

The table below contains a summary of loader-related weaknesses found in the database of Common Vulnerabilities and Exposures (CVEs) in the National Vulnerability Database published by NIST (National Institute of Standards and Technology), as well as Vulnerability Notes (VNs) published by US-CERT (United States Computer Emergency Readiness Team). This table is by no means complete – not just because not all known vulnerabilities are disclosed as a CVE and/or VN, but also because CVE records do not consistently contain enough information to discern if a vulnerability is loader-related. This table lists CVEs and VNs that were either mentioned in a publication (which is cited in the *citations* column of the table) or found while manually searching the CVE database for records that contained at least one of the following words: boot, loader, crafted, bypass, UEFI, BIOS, firmware. It is not possible for me to definitely classify all CVEs that appear to be loader-related due to the brevity of some database records, and so my assignation of a CVE's underlying weakness is sometimes judgment I made based on the available information.

Many of the vulnerabilities I pulled from these databases are expressed in a source-

code granularity – integer overflows, buffer overflows, out-of-bounds (with respect to a buffer) reads, null pointer dereferences, missing permissions checks, etc. Such features are most often what a CVE entry describes and focuses on, but they also fail to model a software instance's underlying weaknesses and deficiencies in capturing the software's higher-level goals and intentions[1]. A higher-level intent violation cannot always be captured as a single source code-level bug – and this table, which draws most of its content from CVEs, merely provides insight into the general landscape of loader vulnerabilities. Section 5.2.6 (on 92) more formally defines the meaning of *type confusion*, *verification failure*, and *ordering failure*.

Table B.1: Example loader-related vulnerabilities

ID	Citations	Weakness	Software
CVE-2000-0729		verification failure	FreeBSD
CVE-2004-1070	[198]	type confusion	Linux
CVE-2004-1071	[198]	type confusion	Linux
CVE-2004-1072	[198]	type confusion	Linux
CVE-2004-1073	[198]	type confusion	Linux
CVE-2006-0741	[198]	type confusion	Linux
CVE-2006-6165	[81, 102]	type confusion	FreeBSD
CVE-2007-3912	[65]	verification failure	Debian
CVE-2007-4315	[160, 161]	verification failure	AMD driver
CVE-2007-4993	[232]	type confusion	XEN
CVE-2007-5549		verification failure	Cisco IOS
CVE-2010-0482		verification failure	Windows
CVE-2010-0486		verification failure	Windows
CVE-2010-0487		verification failure	Windows

continued on next page

[1]MITRE is working to address such omissions with their Common Weakness Enumeration (CWE), which seeks to formalize, standardize, and classify types of software weaknesses.

Table B.1 – *continued from previous page*

ID	Citations	Weakness	Software
CVE-2010-0830		type confusion	glibc
CVE-2010-4346	[112, 147]	type confusion	Linux
CVE-2011-2503	[141]	verification failure	SystemTap
CVE-2011-2883		verification failure	Citrix
CVE-2012-0151	[87]	verification failure	Windows
CVE-2012-2625		verification failure	PyGrub
CVE-2012-3485	[209]	type confusion	Tunnelbick
CVE-2013-0977	[225]	type confusion	iOS
CVE-2013-2195		type confusion	XEN libelf
CVE-2013-2598	[176]	type confusion	LittleKernel
CVE-2013-3582	[120, 223]	verification failure	BIOS
CVE-2013-3900	[177]	verification failure	Windows
CVE-2013-3949		enforcement failure	OS X
CVE-2014-1273	[83, 225]	verification failure	iOS
CVE-2014-2961	[43, 119, 220]	enforcement failure	UEFI
CVE-2014-3880	[78]	type confusion	FreeBSD
CVE-2014-3714		type confusion	XEN
CVE-2014-4325	[70]	enforcement failure	LittleKernel
CVE-2014-4455	[137]	type confusion	iOS
CVE-2014-4707	[98]	enforcement failure	LittleKernel
CVE-2014-4859	[121, 218]	type confusion	tianocore
CVE-2014-4864	[121, 218]	type confusion	tianocore
CVE-2014-7840	[148]	verification failure	QEMU
CVE-2014-8271	[217]	type confusion	tianocore

continued on next page

Table B.1 – *continued from previous page*

ID	Citations	Weakness	Software
CVE-2014-8273	[221, 228]	type confusion	BIOS
CVE-2014-8838		verification failure	OS X
CVE-2014-9793	[115]	type confusion	Android
CVE-2014-9795	[114, 197]	type confusion	LittleKernel
CVE-2014-9796	[128]	type confusion	LittleKernel
CVE-2014-9798	[134]	type confusion	Android
CVE-2014-9801	[132]	type confusion	Android
CVE-2014-9802	[133]	type confusion	Android
CVE-2015-0949	[118, 219]	type confusion	BIOS
CVE-2015-1145		verification failure	OS X
CVE-2015-1146		verification failure	OS X
CVE-2015-2830	[117]	enforcement failure	Linux
CVE-2015-3709	[23]	verification failure	kextd
CVE-2015-3802	[137]	verification failure	iOS
CVE-2015-3803	[137]	verification failure	iOS
CVE-2015-3805	[137]	verification failure	iOS
CVE-2015-3806	[137]	verification failure	iOS
CVE-2015-5281	[60]	verification failure	Grub2
CVE-2015-5839	[137, 169]	type confusion	iOS
CVE-2015-6128		verification failure	Windows
CVE-2015-7055	[224]	enforcement failure	iOS
CVE-2015-7079	[224]	enforcement failure	iOS
CVE-2015-8888	[129]	verification failure	iOS
CVE-2015-8890	[135]	type confusion	Linux

continued on next page

Table B.1 – *continued from previous page*

ID	Citations	Weakness	Software
CVE-2015-8891	[144]	type confusion	LittleKernel
	[184]	verification failure	QNX RTOS
	[94, 165, 168]	verification failure	OpenBSD
CVE-2015-8892	[116]	verification failure	Android
CVE-2015-8893	[130]	type confusion	LittleKernel
CVE-2015-8967	[181]	enforcement failure	Linux
CVE-2016-0014		?	Windows
CVE-2016-0016		?	Windows
CVE-2016-0018		?	Windows
CVE-2016-0020		?	Windows
CVE-2016-0041		?	Windows
CVE-2016-0042		?	Windows
CVE-2016-0160		enforcement failure	Windows
CVE-2016-0428		verification failure	Solaris
CVE-2016-0807		?	elf_utils
CVE-2016-1000		?	Windows
CVE-2016-1738		verification failure	iOS dyld
CVE-2016-2050		type confusion	libdwarf
CVE-2016-2226		type confusion	GNU libiberty
CVE-2016-3850	[131]	type confusion	LittleKernel
CVE-2016-4488	[27]	type confusion	GNU binutils
CVE-2016-4489	[28]	type confusion	GNU libiberty
CVE-2016-4490	[29]	type confusion	GNU libiberty
CVE-2016-4491	[30]	type confusion	GNU libiberty

continued on next page

Table B.1 – *continued from previous page*

ID	Citations	Weakness	Software
CVE-2016-4492	[26]	type confusion	GNU libiberty
CVE-2016-4493	[26]	type confusion	GNU libiberty
CVE-2016-5027	[140]	type confusion	libdwarf
CVE-2016-5031	[2]	type confusion	libdwarf
CVE-2016-5034	[2]	type confusion	libdwarf
CVE-2016-5035	[2]	type confusion	libdwarf
CVE-2016-5247		verification failure	BIOS
CVE-2016-7247		verification failure	Windows
CVE-2016-7275		?	Office
CVE-2016-7292		verification failure	Windows
CVE-2016-7410	[2]	type confusion	libdwarf
CVE-2016-7511	[2]	type confusion	libdwarf
CVE-2016-8680	[145]	type confusion	dwarfdump
CVE-2016-8681	[146]	type confusion	dwarfdump
CVE-2016-9379	[233]	verification failure	pygrub
CVE-2016-9380	[233]	verification failure	pygrub
CVE-2016-10254		verification failure	elfutils
CVE-2016-10255		verification failure	elfutils
CVE-2017-0039		verification failure	Windows
CVE-2017-7210	[172]	type confusion	objdump
CVE-2017-7607		type confusion	elfutils
CVE-2017-7608		type confusion	elfutils
CVE-2017-7609		verification failure	elfutils
CVE-2017-7610		type confusion	elfutils

continued on next page

Table B.1 – *continued from previous page*

ID	Citations	Weakness	Software
CVE-2017-7611		type confusion	elfutils
CVE-2017-7612		type confusion	elfutils
CVE-2017-7613		verification failure	elfutils
CVE-2017-8421	[108]	verification failure	objdump
CVE-2017-8396	[164]	type confusion	libbfd
CVE-2017-9038		type confusion	binutils
CVE-2017-9039		type confusion	binutils
CVE-2017-9040		type confusion	binutils
CVE-2017-9041		type confusion	binutils
CVE-2017-9042		type confusion	binutils
CVE-2017-9043		type confusion	binutils
CVE-2017-9044		type confusion	binutils
VU#127284 (not posted)	[231]	type confusion	BIOS
VU#255726 (not posted)	[120]	enforcement failure	BIOS

C

U-Boot Frama-C value analysis

C.1 Running a Frama-C analysis on U-Boot

The well-known and widely-used Frama-C analysis framework has been designed to work with systems software. Performing a Frama-C analysis on the U-Boot SPL for the BBxM was not a straight-forward task. For example, because Frama-C does not analyze assembly code, I manually translated portions of U-Boot written in assembly into C. Frama-C both had trouble with a number of type declarations in U-Boot, as well as issues with the alignment of some of its write operations. I will discuss how I addressed these and other analysis difficulties in this appendix, provide statistics on what kind of information I derived from Frama-C, as well as discuss how I performed the analysis.

C.1.1 Frama-C value analysis statistics

My primary goal of working with Frama-C was to perform a *value analysis* on U-Boot in order to calculate the set of possible values of all variables. Because U-Boot's

source included a number of important memory addresses stored as variables, a value analysis would likely highlight a number of write operations whose destination address could be statically calculated. In other words, using Frama-C terminology, I was interested in how many lvalues (which can be thought of as the address of the variable on the left-hand side of an expression/the expression result's storage location) could be calculated by a value analysis. It turns out that Frama-C statically resolves about 13% of all write instructions evaluated during a successful boot. This and other value analysis-related statistics are summarized in table C.1.

Table C.1: Frama-C BBxM U-Boot SPL value analysis statistics

Total number write instructions	1,596
Statically-resolved writes	593 (37%)
Writes whose resolved destination range is > 4 bytes	43
Runtime writes performed by these pre-resolved instructions	$\sim$13%

C.1.2 Frama-C ARM architecture support

As of when I performed this analysis, Frama-C did not natively support the ARM architecture. Therefore, I needed to implement an ARMv7-based model of the BBxM processor architecture's endiandness and fundamental data types for Frama-C – this can be found in section C.3.

C.1.3 U-Boot source code post-preprocessing tool

U-Boot has a fairly complex build process and thus I decided to provide Frama-C with the source code produced by the C preprocessor instead of U-Boot's "raw" source code. However, for many reasons discussed in this chapter, Frama-C is unable to analyze Frama-C's preprocessed code as-is. Because I wanted to minimize the number of changes I made to U-Boot's source code, I created a tool that instead patches the intermediary/preprocessed source produced by the preprocessor. This

tool performs simple string manipulations based on markup (in the form of empty preprocessor macros) that I insert into the source code and produces a patched set of (post-)preprocessed files for Frama-C to analyze.

C.1.4 Incorporating assembly into the Frama-C analysis

The U-Boot bootloader performs a number of important operations that are implemented in assembly, all of which I manually translated into C for Frama-C.

Because Frama-C cannot naively analyze the U-Boot's assembly-based entry point, I translated its entrypoint into a C function named `frama_go` which is also where I instruct Frama-C to begin its analysis. In this C-based translation, I use global variables to act as the instruction set's registers. I use C-preprocessor line control macros[1] to instruct the compilation tools from where in the source code each translated assembly statement originated.

A globally-referenced pointer to U-Boot's *global data* structure, which stores pointers to all of U-Boot's bookkeeping data such drivers, is stored in a specially-reserved general-purpose register, `r9`[2], and is chiefly addressed using `DECLARE_GLOBAL_DATA_PTR` macro-generated inline assembly code (defined in arch/arm/include/asm/global_data.h) which "stores" the global data's address as a pointer named `gd`:

```
#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r9")
```

As-is, Frama-C treats this statement's assembly instruction as having no side-effects. Also, because the `gd` variable is declared to be `volatile`, by default Frama-C assumes that its value is non-deterministic which results in any value written to *gd not* getting propagated through the value analysis, and consequently a less-than-useful value

[1] Line control macros inform the C compiler as to where a line of source code has originated which is useful for debugging purposes. Such file/line-number information are typically transparently managed by the compilation toolchain. These macros allow me to tell the compilation tools to act as if a particular line of source code originated in a different file. More specifically, I use them to identify the location of the assembly code from which a line of C code was translated. Information on GNU line control macros can be found in the GCC manual or at https://gcc.gnu.org/onlinedocs/cpp/Line-Control.html.

[2] The `r9` register is reserved by making use of the compiler's *-ffixed-r9* option.

analysis. To address this volatility issue, I modeled the global data structure using ACSL (ANSI/ISO C Specification Language) *ghost variables*. **ACSL** is source-code markup language for specifying formal properties that guide analysis and are only visible to Frama-C and in practice are written as comments in the source code being analyzed. ACSL **ghost variables** are C-like variables and statements that are only visible to Frama-C.

In order to address the issues of the global data structure being accessed using assembly, post-preprocessor tools transforms all `DECLARE_GLOBAL_DATA_PTR` statements into statements that instead define `gd` as a plain, uninitialized pointer so all lines containing the string
`register volatile gd_t *gd asm ("r9")` become `gd_t *gd;`. I then use ACSL ghost statements to populate the value of gd. This is achieved using the following ACSL:

```
//@ volatile gd reads read_gd writes write_gd;
```

This statement instructs Frama-C to use the value generated by the ghost function `read_gd` whenever `gd` is read and to call the ghost function `write_gd` whenever `gd` is written. `read_gd` and `write_gd` are implemented by the following ACSL statements:

```
/*@ ghost gd_t *gdghost;
    gd_t *read_gd(gd_t **p) {return gdghost;}
    gd_t *write_gd(gd_t **p, gd_t *x) {return gdghost = x;}
  */
```

The `gdghost` ghost variable is used to store the *global data* address in place of the `r9` register. `gdghost` is used in conjunction with `read_gd` and `write_gd` to emulate the job of the `r9` register for the Frama-C analysis.

Unfortunately Frama-C does not natively support the double indirection I use to implement these ghost functions (as of the time I performed this analysis), and

thus required I make two minor changes to its annotation type checker. This patch is shown in section C.5.

Another piece of U-Boot code written in assembly I ported to C is a function named `get_cpu_id` (implemented in arch/arm/cpu/armv7/omap3/sys_info.c) which uses a special ARM instruction to retrieve information from (and about) the processor. As a workaround, I simply used my post-preprocessing tool to replace the function's inline assembly with a statement that produces the value that the BBxM's processor always returns, according to its technical reference. The following shows U-Boot's implementation of `get_cpu_id` with my markup macro that instructs post-preprocessor tool to replace the statement following the macro.

```
1 u32 get_cpu_id(void) {
2     struct ctrl_id *id_base;
3     u32 cpuid = 0;
4 #define ___FRAMAC_cpuid_spl_PATCH
5     __asm__ __volatile__("mrc p15, 0, %0, c0, c0, 0":"=r"(cpuid));
```

The macro in line 4 in the above source code is instructs my post-preprocessor to replace the statement on line 5 with `cpuid = 0x3;`.

A final bit of assembly that needs to be replaced is part of a test the bootloader performs that checks from where in its address spaces it is currently executing, it implemented in arch/arm/cpu/armv7/omap3/sys_info.c and shown below with my markup macro.

```
1 static u32 get_base(void) {
2   u32 val;
3 #define ___FRAMAC_val_spl_PATCH
4   __asm__ __volatile__("mov %0, pc \n":"=r"(val)::"memory");
5   val &= 0xF0000000;
6   val >>= 28;
7   return val;
8 }
```

The macro on line 3 instructs my post-preprocessor to replace the inline assembly on line 4 with `val = get_base;`, so that the address of `get_base` (the entrypoint of the currently executing function) is written to `val` in place of the program counter.

Patching type violations

Frama-C has identified multiple type violations in the U-Boot source code which cause it to prematurely halt analysis and are, therefore, patched by my post-preprocessor. More specifically, it identified two instances of statements within `void` functions that return values. For example, the `void` function `puts()` (implemented in common/console.c) includes the following statements:

```
1 if (!gd->have_console)
2 #define ___FRAMAC_noreturn_spl_PATCH
3     return pre_console_putc(c);
```

My macro on like 2 instructs the post-preprocessor to remove the string "return" from the return statement on line 3.

There is also an instance in which a `void` function (`__bad_unaligned_access_size`) is called in a manner where it is expected to return a value (in include/linux/unaligned/generic.h), more specifically:

```
1 #define ___FRAMAC_void_to_int_spl_PATCH
2 extern void __bad_unaligned_access_size(void);
3
4 #define __get_unaligned_le(ptr) ((__force typeof(*(ptr)))({ \
5   __builtin_choose_expr(sizeof(*(ptr)) == 1, *(ptr), \
6   __builtin_choose_expr(sizeof(*(ptr)) == 2, get_unaligned_le16((ptr)), \
7   __builtin_choose_expr(sizeof(*(ptr)) == 4, get_unaligned_le32((ptr)), \
8   __builtin_choose_expr(sizeof(*(ptr)) == 8, get_unaligned_le64((ptr)), \
9   __bad_unaligned_access_size())))); \
10 }))
```

The function declaration on line 2 states that `__bad_unaligned_access_size` is `void`, yet its use on line 9 implies that it returns a value. The markup macro on line 1

instructs my post-preprocessor to replace the function declaration's `void` return type with `int`, thus pacifying Frama-C.

C.1.5 Alignment issues

Frama-C's value analysis process occasionally has issues handling pointers whose values were calculated via bit manipulation operations. This result in alignment-issues that affect its analysis. In order to address these issues, my post-preprocessor tool makes the following adjustments:

The statement `offset = (offset & 15) << 27 | (offset & 0x300) << 17;` (in arch/arm/cpu/armv7/omap3/sdrc.c) inhibits useful analysis when `offset` is later treated as a pointer, therefore my post-preprocessor tool statically sets the value of `offset` to zero, which conveniently happens to always be `offset`'s value.

C.1.6 Static linker-generated structures

Frama-C also has issues performing value analyses on U-Boot's linker-generated lists as described in section A.3.4 on page 157. More specifically, Frama-C has alignment-related troubles with the macros when it locates the beginning of a list using `ll_entry_start()` and instances where the length of a list is calculated using `ll_entry_count()`. My post-preprocessor replaces such problematic statements with the value they ultimately produce – values which (fortunately) can be statically calculated by hand.

C.1.7 Recursion

Frama-C cannot soundly analyze recursive function calls. Fortunately, there is only one such call in the BBxM U-Boot SPL source code, and it occurs while parsing a storage device's DOS-based partition table (`get_partition_info_extended` implemented in disk/part_dos.c) –

```
return get_partition_info_extended(dev_desc, lba_start,
                                   ext_part_sector == 0 ? lba_start : relative,
                                   part_num, which_part, info, disksig);
```

My post-preprocessor replaces this recursive call with a simple return statement.

C.1.8 Frama-C execution options

To analyze the U-Boot SPL, I execute Frama-C with the following value-analysis options set:

- **`-no-initialized-padding-locals`** and **`-val-initialization-padding-globals=no`**: Disables implicit initialization of padding bits for local and global variables.
- **-absolute-valid-range 0x10000000-0xffffffff** indicates that all absolute addresses must fall in this range and any accesses outside this range are deemed to be invalid. This range in particular includes all volatile memory and registers.
- **-val-builtin malloc:Frama_C_malloc_fresh,free:Frama_C_free**: asks Frama-C to use its builtin definitions of `malloc` and `free` instead of using the ones provided by U-Boot. I decided to opt for this because Frama-C runs into alignment issues while analyzing U-Boot's built-in `malloc` implementation.
- **-slevel=1**: instructs the value analysis plugin to superimpose up to one state while unrolling control flow – the default value is 0. A value of 1 allows for more precision. Counter-intuitively, a value of 1 speeds up the analysis process – value analysis takes less than eight minutes when `slevel` is 1, but it takes more than 50 hours (and then crashes) when it is set to its default value, 0[3].

[3]I assume this behavior is indicative of a bug in Frama-C's analysis, however addressing such a bug is clearly out-of-scope of this thesis.

C.2 Frama-C destination analysis plugin source

```
open Cil_types
open Cil

module SS = Set.Make(String)
module Funcall_info = struct
  type funcallinfo = {
      lval:Cil_types.lval;
      exp:Cil_types.exp;
      lexloc:Cil_types.location;
      lvalloc:Locations.location;
      lvalfulladdr:Integer.t;
      instr:Cil_types.instr;
      min:Abstract_interp.Int.t option;
      max:Abstract_interp.Int.t option;
    }

  let form_callstack_string cs =
    List.fold_right (fun c s -> (match c with (f, _) ->
                                  s ^ "->" ^
                                    (Ast_info.Function.get_name
                                    ↪  f.fundec))) cs ""
  let build_callinfo s kinstr =
    if (Db.Value.is_computed()) && (Db.Value.is_reachable_stmt s)  then
      (match Db.Value.get_stmt_state_callstack ~after:true s with
         None -> SS.empty
       | Some(state) -> Value_types.Callstack.Hashtbl.fold
                          (fun cs state r ->
                            (if Cvalue.Model.is_reachable state then
                               SS.add (form_callstack_string cs) r
                             else
                               SS.empty))
                           state SS.empty)
    else
      SS.empty
end

let help_msg = "Resolves as many memory write destinations as possible"

module Self = Plugin.Register
                (struct
                  let name = "write destination resolver"
                  let shortname = "dst"
                  let help = help_msg
                end)
```

```
module Enabled = Self.False
                   (struct
                     let option_name = "-dst"
                     let help = "when on (off by default), " ^ help_msg
                   end)
module More_enabled = Self.False
                   (struct
                     let option_name = "-dst-more"
                     let help = "print more dst info, " ^ help_msg
                   end)

module Output_file = Self.String
                       (struct
                         let option_name = "-dst-output"
                         let default = "-"
                         let arg_name = "output_file"
                         let help =
                           "file where the message is output (default:
                           ↪  console)"
                       end)

module Location_helper= struct
  let loc_to_loc_and_size loc =
    (Locations.loc_to_loc_without_size loc, Locations.loc_size loc)

  let loc_bytes_to_addr_int l =
    try
      match l with
        (Locations.Location_Bytes.Map(m), _) -> (
        match Locations.Location_Bytes.M.find Base.null m with
          Ival.Set([|i|]) -> i
        | _ -> Integer.zero) (* zero or more than one results *)
      | _ -> Integer.zero (* no location map *)
    with Not_found -> Integer.zero

  let get_min_max l =
    try
      (match l with
         (llv, lsz) ->
         (match ((Cvalue.V.project_ival llv), (Int_Base.project lsz)) with
            (v, sz) ->
            (match Ival.min_and_max v with
               (Some(min), Some(max)) ->
                (Some(min),
                 Some(Integer.add max
                                 (Integer.native_div sz (Integer.of_int
                                 ↪  8))))
```

```
            | _ -> (None, None))))
    with Cvalue.V.Not_based_on_null -> (None, None)
end

module Instr_info = struct
  type instrinfo = {
      lval:Cil_types.lval;
      exp:Cil_types.exp;
      lexloc:Cil_types.location;
      lvalloc:Locations.location;
      lvalfulladdr:Integer.t;
      instr:Cil_types.instr;
      min:Abstract_interp.Int.t option;
      max:Abstract_interp.Int.t option;
      callinfo:SS.t;
    }

  let lexloc_string info =
    let ({Lexing.pos_fname=f1; Lexing.pos_lnum=l1; _}, _) = info.lexloc in
    Printf.sprintf "%s:%d"  f1 l1

  let get_lvalloc info =
    info.lvalloc

  let instr_string info =
    let s = Printer.pp_instr Format.str_formatter info.instr in
    Format.flush_str_formatter s

  let eval_lval lval kinstr =
    !Db.Value.lval_to_loc ~with_alarms:CilE.warn_none_mode kinstr lval

  let callstack_str info =
    if SS.is_empty info.callinfo then
      ""
    else
      SS.choose info.callinfo

  let lval_string info =
    let s = Printer.pp_lval Format.str_formatter info.lval in
    Format.flush_str_formatter s

  let build_instrinfo st kinstr =
    if Db.Value.is_computed() then
      match st.skind with
        Instr (Set(lv, e, location) as s) ->
         (let lvl = eval_lval lv kinstr in
          (let (min, max) = Location_helper.get_min_max
```

```
                                (Location_helper.loc_to_loc_and_size lvl) in
            (let  ii = {
                  lval = lv;
                  exp = e;
                  lexloc = location;
                  lvalloc = lvl;
                  lvalfulladdr = Location_helper.loc_bytes_to_addr_int
                                   (Location_helper.loc_to_loc_and_size lvl);
                  instr = s;
                  min = min;
                  max = max;
                  callinfo = Funcall_info.build_callinfo st kinstr;
                }
             in Some(ii))))
      | _ -> None
    else
      None
end

let print_msg =
  object (self : 'self)
    val mutable tofile = not Output_file.is_default()
    val mutable file_chan = if Output_file.is_default() then
                              stdout
                            else
                              open_out (Output_file.get())

    method ival_string ival =
      (let s = Abstract_interp.Int.pretty
                 Format.str_formatter ival in
         Format.flush_str_formatter s;)

    method print msg =
      if tofile then
        Printf.fprintf file_chan "%s\n" msg
      else
        Self.result "%s" msg
    method print_range info =
      (let {Instr_info.min=min; Instr_info.max=max; _} = info in
       match (min, max) with
         Some(min'), Some(max') ->
          self#print (Printf.sprintf "[%s, %s] %s in %s .. %s\n"
                                     (self#ival_string min')
                                     (self#ival_string max')
                                     (Instr_info.lval_string info)
                                     (Instr_info.lexloc_string info)
                                     (Instr_info.callstack_str info));
```

```
      | _ -> ();)
    method print_more info enabled =
      if enabled then
        (let s = Locations.pretty Format.str_formatter
                                  (Instr_info.get_lvalloc info) in
         self#print (Printf.sprintf "%s = %s (%s) .. %s \n"
                                    (Instr_info.instr_string info)
                                    (Format.flush_str_formatter s)
                                    (Instr_info.lexloc_string info)
                                    (Instr_info.callstack_str info)););
      ()
    method close =
      if tofile then
        close_out file_chan
  end

class print_dsts print_obj more = object (self: 'self)

  inherit Visitor.frama_c_inplace

  method! vstmt_aux s =
    (match Instr_info.build_instrinfo s self#current_kinstr with
       Some(info) -> (print_obj#print_range info;
                      print_obj#print_more info more)
     | _ -> ());
    Cil.DoChildren
end

let run () =
  if Enabled.get() then
    Visitor.visitFramacFileSameGlobals (new print_dsts print_msg
                                            (More_enabled.get()))
                                       (Ast.get ());
  print_msg#close

let () = Db.Main.extend run
```

C.3 ARM architecture definition for Frama-C

```
  open Cil_types

  let arm = {
      version          = "arm machdep";
    compiler         = "gcc";
    cpp_arch_flags = [];
    (* All types but char and long long are 16 bits *)
```

```
    sizeof_short     = 2;
    sizeof_int       = 4;
    sizeof_long      = 4;
    sizeof_longlong  = 8;
    sizeof_float     = 4;
    sizeof_double    = 8;
    sizeof_longdouble = 8;
    sizeof_ptr       = 4;
    sizeof_void      = 4;
    sizeof_fun       = 4;
    wchar_t = "int";
    alignof_str = 1;
    alignof_fun = 1;
    char_is_unsigned = false;
    underscore_name = false;
    const_string_literals = false;
    alignof_aligned = 8;
    has__builtin_va_list = true;
    __thread_is_keyword = true;
    alignof_short    =  2;
    alignof_int      = 4;
    alignof_long     = 4;
    alignof_longlong = 8;
    alignof_float    = 4;
    alignof_double   = 8;
    alignof_longdouble = 8;
    alignof_ptr      = 4;
    little_endian = true;
    size_t = "unsigned int";
    ptrdiff_t = "int";

  }

let () = File.new_machdep "arm" arm
```

C.4 C representation of U-Boot assembly code

```
#include <common.h>

DECLARE_GLOBAL_DATA_PTR;

#include <configs/ti_armv7_common.h>
#include <spl.h>
#include <asm/arch-omap3/clock.h>

u32 *sp;
```

```
u32 *r1;
u32 *r0;
#define ___FRAMAC_go_to_speed_spl_ADDR_PATCH
u32 *framac_go_to_speed = 0;
#define ___FRAMAC_lowlevel_init_spl_ADDR_PATCH
u32 *framac_end = lowlevel_init;
#define ___FRAMAC___bss_end_spl_ADDR_PATCH
u32 *framac__bss_end = 0;
#define ___FRAMAC___bss_start_spl_ADDR_PATCH
u32 *framac__bss_start = 0;
u32 *framac_ret;
void cpy_clk_code(u32 *ptr);
void lowlevel_init_finish();
ulong board_init_f_mem(ulong top);
int _main();
int _main_finish();
void s_init();
u32 spl_relocate_stack_gd();
void frama_go();
int lowlevel_init();
struct omap_boot_parameters bxparams = {0, 8, 0, 0, 2, 0};
struct omap_boot_parameters **bxparamptr = 0x4020E024;

/*@ terminates \false;
  @ ensures \false;
*/
#define ___FRAMAC_artifical_spl_ENTRYPOINT
void frama_go()
{
#line 23 "arch/arm/cpu/armv7/omap-common/lowlevel_init.S"
  *bxparamptr =  (u32 *) &bxparams;
#line 50 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
  lowlevel_init();
#line 44 "frama_c_tweaks.c"
}
/*@ terminates \false;
  @ ensures \false;
*/
int lowlevel_init() {
  sp = LOW_LEVEL_SRAM_STACK;
#line 188 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
  *sp = &frama_go;
#line 198 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
  cpy_clk_code(SRAM_CLK_CODE);
#line 55 "frama_c_tweaks.c"
}
/*@ terminates \false;
```

```
  @ ensures \false;
*/
void lowlevel_init_finish()
{
#line 206 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
  s_init(); // not defined in any header
#line 64 "frama_c_tweaks.c"
}

/*@ terminates \false;
  @ ensures \false;
*/
void cpy_clk_code(u32 *ptr)
{
  r0 = framac_go_to_speed;
  while (ptr < framac_end) {
#line 53 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
    *ptr = *r0;
#line 52 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
    ptr++;
#line 53 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
    r0++;
#line 80 "frama_c_tweaks.c"
  }
#line 60 "arch/arm/cpu/armv7/omap3/lowlevel_init.S"
  lowlevel_init_finish();
#line 84 "frama_c_tweaks.c"
}
/*@ terminates \false;
  ensures \false;
*/
int _main()
{
  sp = CONFIG_SYS_INIT_SP_ADDR; // ti_armv7_commoh.h
  //  bic         sp, sp, #7        /* 8-byte alignment for ABI compliance
   ↪  */
#line 86 "arch/arm/lib/crt0.S"
  board_init_f_mem(sp); //common.h
#line 95 "frama_c_tweaks.c"
  return -1; // shouldn't happen
}
ulong board_init_f_mem_finish(ulong framac_ret)
{
  sp = framac_ret;
#line 94 "arch/arm/lib/crt0.S"
  board_init_f(0);  //common.h
#line 103 "frama_c_tweaks.c"
```

```
  return -1;
}
/*@ terminates \false;
  ensures \false;
*/
int _main_finish() {
#line 140 "arch/arm/lib/crt0.S"
  framac_ret = (u32) spl_relocate_stack_gd(); // not in any header but
  ↪  defined in spl/spl.c
#line 112 "frama_c_tweaks.c"
  if (framac_ret != 0) {
    sp = framac_ret;
  }
#line 145 "arch/arm/lib/crt0.S"
  clear_bss();
#line 118 "frama_c_tweaks.c"
}
/*@ terminates \false;
  @ ensures \false;
*/
int clear_bss(){
  // clear out bss, spl.h
#line 158 "arch/arm/lib/crt0.S"
  r1 = framac__bss_end; //@ assert \valid(r1);
#line 149 "arch/arm/lib/crt0.S"
  r0 = framac__bss_start; //@ assert \valid(r0);
  //@ loop pragma WIDEN_HINTS r0, 0x80000000, 0x80030144;
#line 130 "frama_c_tweaks.c"
  while (r0 != r1) { //@assert \pointer_comparable(r0, r1);
    //@ assert \valid(r0+(0..4));
#line 167 "arch/arm/lib/crt0.S"
    *r0 = 0;
    r0++;
#line 136 "frama_c_tweaks.c"
  }
#line 184 "arch/arm/lib/crt0.S"
  board_init_r(gd, gd->relocaddr); //common.h
#line 140 "frama_c_tweaks.c"
}

dpll_param *get_36x_core_dpll_param()
{
#define ___FRAMAC_core_36x_dpll_param_spl_ADDR_PATCH
  dpll_param *core_36x_dpll_param;
  return core_36x_dpll_param;
}
```

```
dpll_param *get_36x_per_dpll_param()
{
#define ___FRAMAC_per_36x_dpll_param_spl_ADDR_PATCH
  dpll_param *per_36x_dpll_param;
  return per_36x_dpll_param;
}

dpll_param *get_36x_per2_dpll_param()
{
#define ___FRAMAC_per2_36x_dpll_param_spl_ADDR_PATCH
  dpll_param *per2_36x_dpll_param;
  return per2_36x_dpll_param;
}

dpll_param *get_36x_iva_dpll_param()
{
#define ___FRAMAC_iva_36x_dpll_param_spl_ADDR_PATCH
  dpll_param *iva_36x_dpll_param;
  return iva_36x_dpll_param;
}

dpll_param *get_36x_mpu_dpll_param()
{
#define ___FRAMAC_mpu_36x_dpll_param_spl_ADDR_PATCH
  dpll_param *mpu_36x_dpll_param;
  return mpu_36x_dpll_param;
}

dpll_param *get_core_dpll_param()
{
#define ___FRAMAC_core_dpll_param_spl_ADDR_PATCH
  dpll_param *core_dpll_param;
  return core_dpll_param;
}

dpll_param *get_per_dpll_param()
{
#define ___FRAMAC_per_dpll_param_spl_ADDR_PATCH
  dpll_param *per_dpll_param;
  return per_dpll_param;
}

dpll_param *get_per2_dpll_param()
{
#define ___FRAMAC_per2_dpll_param_spl_ADDR_PATCH
  dpll_param *per2_dpll_param;
  return per2_dpll_param;
```

```
}

dpll_param *get_iva_dpll_param()
{
#define ___FRAMAC_iva_dpll_param_spl_ADDR_PATCH
  dpll_param *iva_dpll_param;
  return iva_dpll_param;
}

dpll_param *get_mpu_dpll_param()
{
#define ___FRAMAC_mpu_dpll_param_spl_ADDR_PATCH
  dpll_param *mpu_dpll_param;
  return mpu_dpll_param;
}
```

C.5 Frama-C source code patch

```
diff -r -u frama-c-Phosphorus-20170501.orig/src/kernel_services/
↪  ast_queries/logic_typing.ml
↪  frama-c-Phosphorus-20170501.patched/src/kernel_services/ast_queries/
↪  logic_typing.ml
--- frama-c-Phosphorus-20170501.orig/src/kernel_services/ast_queries/
↪  logic_typing.ml        2017-06-01 04:02:17.000000000
↪  -0400
+++ frama-c-Phosphorus-20170501.patched/src/kernel_services/ast_queries/
↪  logic_typing.ml        2017-08-13 08:30:31.903248755
↪  -0400
@@ -4044,7 +4044,6 @@
               (not (isVoidType ret || is_varg_arg))
               && isPointerType arg1
               && Cil_datatype.Typ.equal (typeOf_pointed arg1) ret_type
-              && Cil.typeHasAttributeDeep "volatile" ret
             ->  (* matching prototype: T fct (T *arg1) when T has some
              ↪  volatile attr*)
             checks_tsets_type fct ret_type (* tsets should have type: T *)
           | _ ->
@@ -4076,7 +4076,6 @@
               && isPointerType arg1
               && Cil_datatype.Typ.equal arg2 ret_type
               && Cil_datatype.Typ.equal (typeOf_pointed arg1) ret_type
-              && Cil.typeHasAttributeDeep "volatile" ret
             ->
             (* matching prototype: T fct (T *arg1, T arg2)
                when T has some volatile attr *)
```

D

BBxM U-Boot SPL stage RBWAC policy definition

D.1 Region definitions

```
regions:
  ROM:
    type: "readonly"
    addresses: [0x40000000, 0x4001C000]
  RAM:
    addresses: [0x40200000, 0x40210000]
    include_children: True
    type: "readonly"
    subregions:
      rom_stack:
        type: "stack"
        addresses: ["ROM_STACK_START", "ROM_STACK_END"]
      downloaded_image:
        addresses: ["ROM_STACK_END", 0x4020FFB0]
        subregions:
          downloaded_image_text:
            type: "readonly"
```

```
            addresses: ["RAM.downloaded_image.start", ".text.end"]
          downloaded_image_data:
            type: "bookkeeping"
            addresses: [".text.end", "__image_copy_end"]
          unused:
            type: "readonly"
            addresses: ["__image_copy_end", "SCRATCH_SPACE_ADDR"]
          scratch_space:
            type: "bookkeeping"
            addresses: ["SCRATCH_SPACE_ADDR",
             ↪  "OMAP5_SCRATCH_SPACE_END"]
          remainder:
            type: "readonly"
            addresses: ["OMAP5_SCRATCH_SPACE_END", 0x4020F000]
          public_stack:
            type: "readonly"
            include_children: True
            addresses: [0x4020F000, 0x4020FFB0]
            subregions:
              beginning:
                type: "readonly"
                addresses: [0x4020F000, 0x4020f840]
              cpy_clk_code:
                type: "future"
                addresses: [0x4020f840, "0x4020f840 + lowlevel_init -
                 ↪  cpy_clk_code"]
              stack_rest0:
                addresses: ["RAM.downloaded_image.public_stack.」
                 ↪  cpy_clk_code.end",
                 ↪  0x4020ff20]
              stack_rest1:
                addresses: [ 0x4020ff20, "RAM.downloaded_image.end"]
      rest0:
        addresses: ["RAM.downloaded_image.end", 0x4020fffc]
        type: "readonly"
      rest1:
        addresses: [0x4020fffc, RAM.end]
        type: "readonly"

RAM1:
  include_children: True
  type: "readonly"
  addresses: ["RAM.start", "RAM.end"]
  subregions:
    begin:
```

```
      type: "readonly"
      addresses: ["RAM.start", ".text.end"]
    downloaded_image_data:
      type: "bookkeeping"
      addresses: [".text.end", "__image_copy_end"]
    later_stack:
      type: "readonly"
      addresses: ["__image_copy_end", "0x4020f840 + lowlevel_init -
      ↪  cpy_clk_code"]
    global_data:
      type: "stack"
      addresses: ["0x4020f840 + lowlevel_init - cpy_clk_code",
      ↪  0x4020ff20]
    end:
      type: "readonly"
      addresses: [ 0x4020ff20, "RAM.end"]


SRAM:
  type: "readonly"
  addresses: [0x80000000, 0xc0000000]
  include_children: True
  subregions:
    bss:
      addresses: [0x80000000, 0x80030144]
      subregions:
        sram_test:
          type: "patching"
          addresses: [[0x80000000, 0x80000008], [0x80000400,
          ↪  0x80000404]]
        bss_rest:
          addresses: "remainder"
    after_bss:
        addresses: [0x80030144, "CONFIG_SYS_TEXT_BASE -
        ↪  sizeof_struct_image_header"]
        type: "readonly"
    image_header:
      type: "future"
      addresses: ["CONFIG_SYS_TEXT_BASE -
      ↪  sizeof_struct_image_header", "CONFIG_SYS_TEXT_BASE"]
    image:
      type: "future"
      addresses: ["CONFIG_SYS_TEXT_BASE", "CONFIG_SYS_TEXT_BASE +
      ↪  main.image_size"]
    post_image:
```

```
        addresses: ["CONFIG_SYS_TEXT_BASE + main.image_size",
        ↪  "CONFIG_SYS_SPL_MALLOC_START"]
      heap:
        type: "readonly"
        addresses: ["CONFIG_SYS_SPL_MALLOC_START",
        ↪  "CONFIG_SYS_SPL_MALLOC_START +
        ↪  CONFIG_SYS_SPL_MALLOC_SIZE"]
      nonbss:
        type: "readonly"
        addresses: ["CONFIG_SYS_SPL_MALLOC_START +
        ↪  CONFIG_SYS_SPL_MALLOC_SIZE", 0xc0000000]
        include_children: True
        subregions:
          sram_test:
            type: "patching"
            addresses: [[0xA0000400, 0xA0000404], [0xA0000000,
            ↪  0xA0000008]]
          rest:
            addresses: "remainder"

  Registers:
    csv: "regs.csv"
    type: "global"
    include_children: True
  missing_control_padconf0:
    type: "global"
    addresses: [0x48002150, 0x48002154]
  missing_control_padconf1:
    type: "global"
    addresses: [0x48002154, 0x48002158]
  missing_protection_mech_0:
    type: "global"
    addresses: [0x68010060, 0x68010068]

values:
  SRAM_STACK: 0x4020fffc
  SCRATCH_SPACE_ADDR: 0x4020E000
  OMAP5_SCRATCH_SPACE_END: 0x4020E030
  ROM_STACK_START: 0x40200000
  ROM_STACK_END: 0x40200800
  CONFIG_SYS_SPL_MALLOC_START: 0x80208000
  CONFIG_SYS_SPL_MALLOC_SIZE: 0x100000
  CONFIG_SYS_TEXT_BASE: 0x80100000
  sizeof_struct_image_header: 64
```

D.2 Substage definitions and region retyping rules

```
_start:
  substage_type: "bookkeeping"
  new_regions: ["ROM", "RAM", "Registers", "missing_control_padconf0",
   ↪  "missing_control_padconf1", "missing_protection_mech_0"]

lowlevel_init:
  substage_type: "bookkeeping"
  reclassified_regions:
    RAM.rest0: "stack"
    RAM.rest1: "stack"
    RAM.rom_stack: "readonly"

cpy_clk_code:
  substage_type: "loading"

lowlevel_init_finish:
  substage_type: "bookkeeping"
  reclassified_regions:
    RAM.downloaded_image.public_stack.stack_rest0: "stack"
    RAM.downloaded_image.public_stack.stack_rest1: "stack"
    RAM.downloaded_image.public_stack.cpy_clk_code: "readonly"
    RAM.rest1: "readonly"


_main:
   substage_type: "bookkeeping"
   undefined_regions: ["RAM.rom_stack",
    ↪  "RAM.downloaded_image.scratch_space",
    ↪  "RAM.downloaded_image.public_stack", "RAM.rest1", "RAM.rest0",
    ↪  "RAM.downloaded_image.unused",
    ↪  "RAM.downloaded_image.public_stack.beginning",
    ↪  "RAM.downloaded_image",
    ↪  "RAM.downloaded_image.public_stack.stack_rest1",
    ↪  "RAM.downloaded_image.public_stack.stack_rest0",
    ↪  "RAM.downloaded_image.downloaded_image_text",
    ↪  "RAM.downloaded_image.downloaded_image_data",
    ↪  "RAM.downloaded_image.public_stack.cpy_clk_code"]
   reclassified_regions:
     RAM1.global_data: "stack"
     RAM1.later_stack: "readonly"
     RAM1.end: "readonly"
```

```
   new_regions: ["RAM1"]

board_init_f_mem:
  substage_type: "patching"
  reclassified_regions:
     RAM1.global_data: "patching"
     RAM1.later_stack: "stack"
     SRAM.bss.sram_test: "readonly"
  new_regions: ["SRAM"]

board_init_f_mem_finish:
   substage_type: "bookkeeping"
   reclassified_regions:
     RAM1.global_data: "global"
     SRAM.bss.sram_test: "patching"
     SRAM.image: "future"
     SRAM.image_header: "future"

board_init_f:
  substage_type: "patching"
  allowed_symbols: ["revision"]

_main_finish:
  substage_type: "bookkeeping"
  reclassified_regions:
    SRAM.bss: "patching"
    SRAM.nonbss: "readonly"
  undefined_regions: ["SRAM.bss.sram_test"]

clear_bss:
  substage_type: "patching"

board_init_r:
  substage_type: "loading"
  reclassified_regions:
    SRAM.bss: "bookkeeping"
    SRAM.heap: "global"
  allowed_symbols: ["mem_malloc_start", "mem_malloc_end",
  ↪  "mem_malloc_brk", "_u_boot_list_2_i2c_2_omap24_0",
  ↪  "spl_boot_list", "mmc_devices", "mem_malloc_brk",
  ↪  "current_mallinfo", "sbrk_base", "av_", "max_sbrked_mem",
  ↪  "max_total_mem", "cur_dev", "fat_registered", "spl_image",
  ↪  "cur_dev_num", "do_fat_read_at_block",
  ↪  "get_contents_vfatname_block", "cur_part_info"]
```

```
spl_after_load_image:
  substage_type: "bookkeeping"
  reclassified_regions:
    SRAM.image: "readonly"
    SRAM.image_header: "readonly"

jump_to_image_no_args:
  substage_type: "bookkeeping"
```

D.3 U-Boot source code

All bootloader analysis was based on revision `fa85e826c` of the U-Boot git repository, which can be retrieved from `http://git.denx.de/u-boot.git`. The source code I ultimately worked with, including all patches, is available at fork located at `https://github.com/bx/u-boot-extended`

Bibliography

[1] @MuscleNerd, "Evolution of the iPhone baseband and unlocks". Presented at: *HITB Amsterdam*. 2012 (cit. on p. 23).

[2] D. A., *David a's libdwarf vulnerabilities*. May 4, 2016. URL: https://www.prevanders.net/dwarfbug.html (visited on 06/20/2017) (cit. on p. 163).

[3] M. Abadi, ET AL., "Control-flow integrity". In: *Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353 (cit. on p. 45).

[4] M. Abadi, ET AL., "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information Systems Security*. Vol. 13. #1. ACM, Nov. 2009, pp. 4:1–4:40 (cit. on p. 45).

[5] Adobe, *Security advisory: revocation of Adobe code signing certificate*. 2012. URL: https://www.adobe.com/support/security/advisories/apsa12-01.html (visited on 06/05/2014) (cit. on p. 19).

[6] Aleph One, "Smashing the stack for fun and profit". In: *Phrack*. Vol. 7. #49. 1996 (cit. on p. 20).

[7] E. Alkassar, N. Schirmer, AND A. Starostin, "Formal pervasive verification of a paging mechanism". In: *Theory and Practice of Software, International Conference on Tools and Algorithms For the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan, AND J. Rehof. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 109–123 (cit. on p. 50).

[8] B. Anckaert, M. Madou, AND K. D. Bosschere, "A model for self-modifying code". In: *International Conference on Information Hiding*. IH'06. Springer-Verlag, 2006, pp. 232–248 (cit. on pp. 48, 49).

[9] D. Arora, ET AL., "Architectural support for safe software execution on embedded processors". In: *International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS'06. ACM, 2006, pp. 106–111 (cit. on p. 45).

[10] ashamsu, *purplera1n*. GitHub repository. Oct. 3, 2014. URL: https://github.com/ashamsu/purplera1n (visited on 05/15/2017) (cit. on p. 25).

[11] "ASUS Eee PC and other series: BIOS SMM privilege escalation vulnerabilities". In: *Bugtraq Mailing List*. Aug. 7, 2009. URL: http://seclists.org/bugtraq/2009/Aug/58 (cit. on p. 31).

[12] AT&T, *UNIX System V release 4: programmer's guide: ANSI C and programming support tools*. Prentice-Hall, 1990 (cit. on p. 11).

[13] axi0mX, *alloc8*. GitHub repository. Apr. 9, 2017. URL: https://github.com/axi0mX/alloc8 (visited on 05/15/2017) (cit. on p. 27).

[14] G. Back, "DataScript- a specification and scripting language for binary data". In: *Generative Programming and Component Engineering*. Ed. by D. Batory, C. Consel, AND W. Taha. Vol. 2487. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 66–77 (cit. on p. 47).

[15] J. Bangert, AND S. Bratus, "ELF eccentricities". Presented at: *CONFidence*. 2013 (cit. on p. 43).

[16] J. Bangert, AND N. Zeldovich, "Nail: a practical interface generator for binary formats". In: *Securirty & Privacy LangSec Workshop*. IEEE, 2014 (cit. on p. 47).

[17] J. Bangert, ET AL., *ELFbac: using the loader format for intent-level semantics and fine-grained protection*. Tech. rep. #TR2013-727. Dartmouth College, Computer Science, May 2013 (cit. on p. 72).

[18] D. W. Baron, *Assemblers and loaders*. 3rd ed. Elsevier North-Holland, 1978 (cit. on p. 13).

[19] M. Bazaliy, "Pegasys internals". Presented at: *Chaos Communication Congress*. 2016 (cit. on p. 27).

[20] O. Bazhaniuk, ET AL., "Attacking and defending BIOS in 2015". Presented at: *REcon*. 2015 (cit. on pp. 4, 31).

[21] O. Bazhaniuk, ET AL., "Symbolic execution for BIOS security". In: *Workshop on Offensive Technologies*. WOOT'15. USENIX, 2015 (cit. on p. 48).

[22] Bberryusa, *Safe C library*. Sorceforge project. URL: http://sourceforge.net/p/safeclib (visited on 06/06/2014) (cit. on p. 45).

[23] I. Beer, *OS X kextd bad path checking and toctou allow a regular user to load an unsigned kernel extension*. Apr. 29, 2015. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=353 (cit. on p. 161).

[24] S. Bing, "BIOS boot hijacking and VMware vulnerabilities digging". Presented at: *Power of Community*. Nov. 16, 2007 (cit. on p. 31).

[25] T. Bletsch, X. Jiang, AND V. Freeh, "Mitigating code-reuse attacks with control-flow locking". In: *Annual Computer Security Applications Conference*. ACSAC'11. ACM, 2011, pp. 353–362 (cit. on p. 45).

[26] M. Bohme, "Bug 70926 – libiberty demangler segfaults (5)". In: *GCC Bugzilla*. May 3, 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70926 (cit. on p. 163).

[27] M. Böhme, "Bug 70481 – [regression] libiberty demangler segfaults". In: *GCC Bugzilla*. Mar. 31, 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70481 (cit. on p. 162).

[28] M. Böhme, "Bug 70492 – libiberty demangler segfaults (2)". In: *GCC Bugzilla*. Apr. 1, 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70492 (cit. on p. 162).

[29] M. Böhme, "Bug 70498 – libiberty demangler segfaults (3)". In: *GCC Bugzilla*. Apr. 1, 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70498 (cit. on p. 162).

[30] M. Böhme, "Bug 70909 – libiberty demangler segfaults (4)". In: *GCC Bugzilla*. May 2, 2016. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=70909 (cit. on p. 162).

[31] G. Bonfante, J.-Y. Marion, AND D. Reynaud-Plantey, "A computability perspective on self-modifying programs". In: *International Conference on Software Engineering and Formal Methods*. SEFM'09. IEEE, 2009, pp. 231–239 (cit. on p. 49).

[32] J. Bonwick, AND J. Adams, "Magazines and vmem: extending the slab allocator to many CPUs and arbitrary resources". In: *USENIX Technical Conference*. USENIX, 2001, pp. 15–33 (cit. on p. 126).

[33] E. C. Brady, "IDRIS: systems programming meets full dependent types". In: *Workshop on Programming Languages Meets Program Verification*. PLPV'11. ACM, 2011, pp. 43–54 (cit. on p. 46).

[34] S. Bratus, AND J. Bangert, "ELFs are dorky, elves are cool". In: *POC or GTFO*. Vol. 0x0. Aug. 2013 (cit. on pp. 11, 43).

[35] S. Bratus, J. Bangert, AND R. Shapiro, "Revisiting "trusting trust" for binary toolchains". In: *Chaos Communication Congress*. 2013 (cit. on p. 19).

[36] S. Bratus, M. E. Locasto, AND M. L. Patterson, "Exploit programming: from buffer overflows to "weird machines" and theory of computation". In: *USENIX; Login*. USENIX, 2011, pp. 13–21 (cit. on p. 36).

[37] E. Brewer, ET AL., "Thirty years is long enough: getting beyond C". In: *Hot Topics In Operating Systems*. HotOS'05. USENIX, 2005, pp. 14–14 (cit. on p. 46).

[38] BSDaemon, coideloko, AND D0nAnd0n, "System management mode hack: using SMM for other purposes". In: *Phrack*. Vol. 12. 2008 (cit. on p. 31).

[39] Y. Bulygin, "Evil Maid just got angrier: why full-disk encryption with TPM is insecure on many system". Presented at: *CanSecWest*. 2013 (cit. on p. 31).

[40] Y. Bulygin, A. Furtak, AND O. Bazhaniuk, "A tale of one software bypass of Windows 8 secure boot". Presented at: *Black Hat USA*. 2013 (cit. on p. 31).

[41] bushing, AND marcan, "Console hacking 2008: WII fail". Presented at: *Chaos Communication Congress*. 2008 (cit. on p. 31).

[42] J. Butterworth, C. Kallenberg, AND X. Kovah, "BIOS chronomancy: fixing the static root of trust for measurement". Presented at: *NoSuchCon*. 2013 (cit. on p. 31).

[43] J. Butterworth, ET AL., "Setup for failure: defeating UEFI/Win8 secure boot". Presented at: *Hack In Paris*. June 2015 (cit. on pp. 31, 160).

[44] C. Cadar, D. Dunbar, AND D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Conference on Operating Systems Design and Implementation*. OSDI'08. USENIX, 2008, pp. 209–224 (cit. on p. 47).

[45] C. Cadar, ET AL., "EXE: automatically generating inputs of death". In: *ACM Transactions on Information and System Security*. Vol. 12. #2. ACM, 2008, p. 10 (cit. on p. 51).

[46] H. Cai, Z. Shao, AND A. Vaynberg, "Certified self-modifying code". In: *SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'07. ACM, 2007, pp. 66–77 (cit. on pp. 48, 49, 142).

[47] M. Castro, M. Costa, AND T. Harris, "Securing software by enforcing data-flow integrity". In: *Symposium on Operating Systems Design and Implementation*. OSDI'06. USENIX, 2006, pp. 147–160 (cit. on p. 45).

[48] S. K. Cha, ET AL., "Unleashing Mayhem on binary code". In: *IEEE Security & Privacy*. IEEE, May 2012, pp. 380–394 (cit. on p. 46).

[49] P. Chen, ET AL., "IntFinder: automatically detecting integer bugs in x86 binary program". In: *International Conference on Information and Communications Security*. Ed. by S. Qing, C. J. Mitchell, AND G. Wang. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 336–345 (cit. on p. 51).

[50] B. V. Chess, "Improving computer security using extended static checking". In: *IEEE Security & Privacy*. IEEE, 2002, pp. 160–173 (cit. on p. 45).

[51] P. Chiffier, "UEFI and PCI bootkits". Presented at: *PACSEC*. 2013 (cit. on p. 31).

[52] S. Chiricescu, ET AL., "SAFE: a clean-slate architecture for secure systems". In: *International Conference on Technologies For Homeland Security*. HST'13. IEEE, 2013, pp. 570–576 (cit. on p. 139).

[53] J. Chirimar, C. A. Gunter, AND J. G. Riecke, "Proving memory management invariants for a language based on linear logic". In: *LISP and Functional Programming*. LFP'92. ACM, 1992, pp. 139–150 (cit. on p. 50).

[54] Chronic, AND iPhone Dev Team, "ARM7 go". In: *The IPhone Wiki*. 2009. URL: https://www.theiphonewiki.com/wiki/ARM7_Go (cit. on p. 29).

[55] chronic, ET AL., "0x24000 segment overflow". In: *The IPhone Wiki*. 2009. URL: https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow (cit. on p. 24).

[56] W. Chuang, S. N. B. Calder, AND R. Jhala, "Bounds checking with taint-based analysis". In: *International Conference on High Performance Embedded Architectures and Compilers*. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 71–86 (cit. on p. 46).

[57] J. Condit, ET AL., "Dependent types for low-level programming". In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, 2007, pp. 520–535 (cit. on pp. 45, 46).

[58] S. D. K. Coogan, AND G. Townsend, *On the semantics of self-unpacking malware code*. Tech. rep. University of Arizona, Computer Science, 2008 (cit. on p. 49).

[59] B. Cook, E. Koskinen, AND M. Vardi, "Temporal property verification as a program analysis task". In: *International Conference on Computer Aided Verification*. CAV'11. Springer-Verlag, 2011, pp. 333–348 (cit. on p. 50).

[60] S. Cornelius, "CVE-2015-5281 grub2: modules built in on EFI builds that allow loading arbitrary code, circumventing secure boot". In: *Red Hat Bugzilla*. Sept. 17, 2016. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1264103 (cit. on p. 161).

[61] D. E. Corporation, *Digital pdp11/70 processor handbook*. Digital Equipment Corporation, 1976 (cit. on p. 13).

[62] C. Cowan, ET AL., "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks". In: *USENIX Security Symposium*. Vol. 81. SSYM'98. USENIX, 1998, pp. 346–355 (cit. on p. 46).

[63] US-CERT, AND NIST, "CVE-2014-4455". In: *National Vulnerability Database*. Nov. 18, 2014 (cit. on p. 26).

[64] P.-E. Dagand, A. Baumann, AND T. Roscoe, "Filet-o-fish: practical and dependable domain-specific languages for OS development". In: *SIGOPS Operating Systems Review*. Vol. 43. #4. ACM, Jan. 2010, pp. 35–39 (cit. on p. 46).

[65] T. de Grenier de Latour, "Checkrestart: arbitrary root-privileged command execution". In: *Debian Bug Tracking System*. Sept. 1, 2009. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=440411 (cit. on p. 159).

[66] R. DeLine, AND M. Fahndrich, "Enforcing high-level protocols in low-level software". In: *SIGPLAN Notices*. Vol. 36. #5. ACM, 2001, pp. 59–69 (cit. on p. 46).

[67] J. Devietti, ET AL., "Hardbound: architectural support for spatial safety of the C programming language". In: *SIGARCH Computer Architecture News*. Vol. 36. #1. ACM, Mar. 2008, pp. 103–114 (cit. on p. 52).

[68] D. Dhurjati, ET AL., "Memory safety without runtime checks or garbage collection". In: *SIGPLAN Conference on Language, Compiler, and Tool For Embedded Systems*. LCTES'03. ACM, 2003, pp. 69–80 (cit. on p. 45).

[69] I. S. Diatchki, AND M. P. Jones, "Strongly typed memory areas programming systems-level data structures in a functional language". In: *SIGPLAN Workshop on Haskell*. Haskell'06. ACM, 2006, pp. 72–83 (cit. on p. 46).

[70] P. Dixit, "Fastboot boot command bypasses signature verification (CVE-2014-4325)". In: *CodeAurora*. Aug. 4, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=2c00928b4884fdb0b1661bcc530d7e68c9561a2f (cit. on p. 160).

[71] W. Drewry, AND T. Ormandy, "Flayer: exposing application internals". In: *Workshop on Offensive Technologies*. WOOT'07. USENIX, 2007, pp. 1–9 (cit. on p. 51).

[72] L. Duflot, "Security issues related to pentium system management mode". Presented at: *CanSecWest*. 2006 (cit. on p. 31).

[73] L. Duflot, ET AL., "Getting into the SMRAM: SMM reloaded". Presented at: *CanSecWest*. 2009 (cit. on p. 31).

[74] L. Duflot, D. Etiemvle, AND O. Grumelard, *Using CPU system management mode to circumvent operating system security functions*. Tech. rep. DCSSI (Central Information Systems Security Division) (cit. on p. 31).

[75] L. Duflot, ET AL., "ACPI and SMI handlers: some limits to trusted computing". In: *Journal In Computer Virology*. Vol. 6. #4. Springer-Verlag, 2010, pp. 353–374 (cit. on p. 31).

[76] D. Evans, AND D. Larochelle, "Improving security using extensible lightweight static analysis". In: *IEEE Software*. Vol. 19. #1. IEEE, 2002, pp. 42–51 (cit. on p. 45).

[77] E. A. Feustel, "On the advantages of tagged architecture". In: *IEEE Transactions on Computers*. Vol. C-22. #7. IEEE, 1973, pp. 644–656 (cit. on p. 139).

[78] FreeBSD, *Triple-fault when executing from a threaded process*. May 3, 2014. URL: https://www.freebsd.org/security/advisories/FreeBSD-EN-14:06.exec.asc (cit. on p. 160).

[79] J. Freeman, *Exploit (& fix) android "Master Key"*. blog. 2013. URL: http://www.saurik.com/id/17 (visited on 06/05/2014) (cit. on pp. 19, 43, 94).

[80] J. Freeman, *Yet another android Master Key bug*. 2013. URL: http://www.saurik.com/id/19 (visited on 06/11/2014) (cit. on pp. 19, 43, 94).

[81] D. G., *Finger 79/tcp # McDonald, Dowd and Schuh challenge: part 2*. Nov. 13, 2006 (cit. on p. 159).

[82] X. Ge, M. Payer, AND T. Jaeger, "An evil copy: how the loader betrays you". In: *Network & Distributed System Security Conference*. NDSS'17. Internet Society, Mar. 2017 (cit. on p. 20).

[83] geohot, *geohot (@tomcr00se) presents an evasion7 writeup*. URL: http://geohot.com/e7writeup.html (cit. on p. 160).

[84] geohot, "iBoot environment variable overflow". In: *The IPhone Wiki*. 2009. URL: https://www.theiphonewiki.com/wiki/IBoot_Environment_Variable_Overflow (cit. on p. 25).

[85] geohot, "Limera1n exploit". In: *The IPhone Wiki*. URL: https://www.theiphonewiki.com/wiki/Limera1n_Exploit (cit. on p. 28).

[86] S. Ghose, ET AL., "Architectural support for low overhead detection of memory violations". In: *Conference on Design, Automation and Test In Europe*. DATE'09. European Design AND Automation Association, 2009, pp. 652–657 (cit. on p. 52).

[87] I. Glucksmann, "Injecting custom payload into signed Windows executables". Presented at: *REcon*. 2012 (cit. on pp. 19, 160).

[88] M. Gorobets, ET AL., "Attacking hypervisors via firmware and hardware". Presented at: *Black Hat USA*. 2015 (cit. on p. 31).

[89] D. Grossman, ET AL., "Region-based memory management in Cyclone". In: *SIGPLAN Programming Language Design and Implementation*. PLDI'02. ACM, 2002, pp. 282–293 (cit. on pp. 47, 71).

[90] M. Harren, AND G. C. Necula, "Using dependent types to certify the safety of assembly code". In: *International Static Analysis Symposium*. Lecture Notes in Computer Science. Springer, 2005, pp. 155–170 (cit. on p. 46).

[91] L. Harrison, AND K. Li, "Arms race: the story of (in)-secure bootloaders". Presented at: *Shmoocon*. 2014 (cit. on p. 30).

[92] N. Hasabnis, A. Misra, AND R. Sekar, "Light-weight bounds checking". In: *Symposium on Code Generation and Optimization*. CGO'12. ACM, 2012, pp. 135–144 (cit. on p. 46).

[93] J. Heasman, "Implementing and detecting a PCI rootkit". Presented at: *Black Hat DC*. 2007 (cit. on p. 31).

[94] A. Hernandez, "OpenBSD $\leq$ 5.5 local kernel panic". IOActive, 2014. URL: http://www.ioactive.com/pdfs/IOActive_Advisory_OpenBSD_5_5_Local_Kernel_Panic.pdf (cit. on p. 162).

[95] A. Hernández, "Melkor – an ELF file format fuzzer". Presented at: *Black Hat USA*. 2014 (cit. on p. 51).

[96] J. Hill, "SHAttered dreams: adventures in BootROM land". Presented at: *HITB Malaysia*. 2013 (cit. on p. 28).

[97] A. Huang, "Keeping secrets in hardware: the Microsoft xbox case study". In: *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES'02. Springer-Verlag, 2003, pp. 213–227 (cit. on p. 31).

[98] Huawei, *Security advisory- bootrom menu and boot menu vulnerabilities on huawei campus switches*. May 7, 2015. URL: http://www.huawei.com/en/psirt/security-advisories/hw-334629 (cit. on p. 160).

[99] T. Hudson, "Thunderstrike". Presented at: *Chaos Communication Congress*. Dec. 2014 (cit. on p. 31).

[100] T. Hudson, AND L. Rudolph, "Thunderstrike: EFI firmware bootkits for Apple MacBooks". In: *International Systems and Storage Conference*. SYSTOR'15. ACM, 2015, pp. 15:1–15:10 (cit. on p. 31).

[101] M. I. Husain, ET AL., *How to bypass verified boot security in chromium OS*. 2012. URL: http://arxiv.org/abs/1202.5282 (cit. on p. 19).

[102] In Cognito, "Lack of environment sanitization in the FreeBSD, OpenBSD, NetBSD dynamic loaders". In: *Bugtraq Mailing List*. Nov. 22, 2006. URL: http://seclists.org/bugtraq/2006/Nov/451 (cit. on p. 159).

[103] Information Sciences Institute, *Transmission control protocol DARPA internet program protocol specification*. RFC #793. Internet Requests for Comments, Sept. 1981 (cit. on p. 10).

[104] IOActive, *Melkor ELF fuzzer*. GitHub repository. Sept. 26, 2014. URL: https://github.com/IOActive/Melkor_ELF_Fuzzer (visited on 09/04/2017) (cit. on p. 51).

[105] iPhone Dev Team, "Diags". In: *The IPhone Wiki*. 2008. URL: https://www.theiphonewiki.com/wiki/Diags (cit. on p. 28).

[106] the iPhone wiki, "usb_control_msg(0x21, 2) exploit". In: *The IPhone Wiki*. 2009. URL: https://www.theiphonewiki.com/wiki/Usb_control_msg(0x21,_2)_Exploit (cit. on p. 28).

[107] B. Jacobs, J. Smans, AND F. Piessens, "Verification of unloadable modules". In: *International Symposium on Formal Methods*. Lecture Notes in Computer Science. 2011, pp. 402–406 (cit. on p. 50).

[108] jgj212, "Bug 21440 – malicious PE with invalid extended relocation can cause binutils/objdump 2.28 to allocate any-size big memory". In: *Sourceware Bugzilla*. Apr. 27, 2017. URL: https://sourceware.org/bugzilla/show_bug.cgi?id=21440 (cit. on p. 164).

[109] T. Jim, ET AL., "Cyclone: a safe dialect of C". In: *USENIX Annual Technical Conference*. USENIX, 2002, pp. 275–288 (cit. on pp. 6, 45).

[110] R. Johnson, AND D. Wagner, "Finding user/kernel pointer bugs with type inference". In: *USENIX Security Symposium*. Vol. 2. SSYM'04. USENIX, 2004 (cit. on p. 46).

[111] S. C. Johnson, *Lint, a C program checker*. Tech. rep. #65. Bell Laboratories, 1978, pp. 78–1273 (cit. on p. 45).

[112] Julien, "Taviso's install_special_mapping bypass for mmap_min_addr". In blog: *Xorl%eax,%eax*. Dec. 9, 2009. URL: https://xorl.wordpress.com/2010/12/09/tavisos-install_special_mapping-bypass-for-mmap_min_addr/ (cit. on p. 160).

[113] L. K, "De mysteriis dom: mac EFI rootkits". Presented at: *Black Hat USA*. 2012 (cit. on p. 31).

[114] C. Kadabi, "App: aboot: verify boot image signature". In: *CodeAurora*. June 18, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=fc3b31f81a1c128c2bcc745564a075022cd72a2e (cit. on p. 161).

[115] C. Kadabi, "Platform: msm_shared: fix write protect function". In: *CodeAurora*. Nov. 22, 2013. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=0dcccecc4a6a9a9b3314cb87b2be8b52df1b7a81 (cit. on p. 161).

[116] C. Kadabi, "Platform: msm_shared: validate the decrypted signature". In: *CodeAurora*. Sept. 8, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=fae606b9dd92c021e2419369975264f24f60db23 (cit. on p. 162).

[117] V. Kaigorodov, "CVE-2015-2830 kernel: int80 fork from 64-bit tasks mishandling". In: *Red Hat Bugzilla*. Apr. 2, 2015. URL: httxps://bugzilla.redhat.com/show_bug.cgi?id=1208598 (cit. on p. 161).

[118] C. Kallenberg, AND X. Kovah, "How many million BIOSes would you like to infect?" Presented at: *HITB Amsterdam*. May 28, 2015 (cit. on pp. 31, 161).

[119] C. Kallenberg, ET AL., "All your boot are belong to us". Presented at: *CanSecWest*. 2014 (cit. on p. 160).

[120] C. Kallenberg, ET AL., "Defeating signed BIOS enforcement". Presented at: *HITB Malaysia*. 2013 (cit. on pp. 31, 99, 160, 164).

[121] C. Kallenberg, ET AL., "Extreme privilege escalation on Windows 8/UEFI systems". Presented at: *DEF CON* (cit. on pp. 31, 160).

[122] D. Kaminsky, M. L. Patterson, AND L. Sassaman, "PKI layer cake: new collision attacks against the global X.509 infrastructure". In: *International Conference on Financial Cryptography and Data Security*. Lecture Notes on Computer Science. Springer Berlin Heidelberg, 2010, pp. 289–303 (cit. on p. 43).

[123] S. Kell, "Dynamically diagnosing type errors in unsafe code". In: *SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA'16. ACM, 2016, pp. 800–819 (cit. on p. 45).

[124] *KLEE LLVM execution engine*. GitHub repository. URL: https://klee.github.io/ (visited on 04/12/2017) (cit. on p. 47).

[125] G. Klein, ET AL., "seL4: formal verification of an OS kernel". In: *Symposium on Operating Systems Principles*. SOSP'09. ACM, 2009, pp. 207–220 (cit. on p. 50).

[126] X. Kovah, ET AL., "Setup for failure: defeating secure boot". Presented at: *SyScan Singapore*. 2014 (cit. on p. 31).

[127] N. Kumar, AND V. Kumar, "Vboot kit: compromising Windows Vista security". Presented at: *Black Hat EU*. 2007 (cit. on p. 31).

[128] V. Kumar, "Aboot: check for max page size". In: *CodeAurora*. June 26, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=2e21b3a57cac7fb876bcf43244d7cc3dc1f6030d (cit. on p. 161).

[129] V. Kumar, "App: aboot: add the integer overflow checks on sparse header". In: *CodeAurora*. Mar. 27, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=1321f34f1ebcff61ad7e65e507cfd3e9028af19b (cit. on p. 161).

[130] V. Kumar, "App: aboot: check for buffer over reads". In: *CodeAurora*. Apr. 24, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=800255e8bfcc31a02e89460460e3811f225e7a69 (cit. on p. 162).

[131] V. Kumar, "App: aboot: check for integer overflow". In: *CodeAurora*. Sept. 12, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=030371d45a9dcda4d0cc3c76647e753a1cc1b782 (cit. on p. 162).

[132] V. Kumar, "Lib: fdt: add integer overflow checks". In: *CodeAurora*. Aug. 11, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=cf8f5a105bafda906ccb7f149d1a5b8564ce20c0 (cit. on p. 161).

[133] V. Kumar, "Lib: fdt: add integer overflow checks in fdt header". In: *CodeAurora*. Aug. 12, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=222e0ec9bc755bfeaa74f9a0052b7c709a4ad054 (cit. on p. 161).

[134] V. Kumar, "Platform: msm_shared: check for atags address". In: *CodeAurora*. June 25, 2014. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=b05eed2491a098bf627ac485a5b43d2f4fae2484 (cit. on p. 161).

[135] V. Kumar, "Platform: msm_shared: check for CRC of GPT header and partition entries". In: *CodeAurora*. Apr. 28, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=e22aca36da2bb6f5016f3c885eb8c8ff85c115e4 (cit. on p. 161).

[136] X. Leroy, "Formal verification of a realistic compiler". In: *Communications of the ACM*. Vol. 52. #7. ACM, July 2009, pp. 107–115 (cit. on p. 49).

[137] J. Levin, *The annotated (informal) guide to TaiG – part the 1st*. 2015. URL: http://newosxbook.com/articles/TaiG.html (cit. on pp. 19, 160, 161).

[138] J. Levin, "To sign and protect COPS in OS X and iOS". Presented at: *RSA*. Apr. 2015 (cit. on pp. 26–28).

[139] J. R. Levine, *Linkers and loaders*. Morgan Kaufmann, 1999 (cit. on pp. 14, 71, 126).

[140] lieanu, "Bug 1330237 – NULL dereference bug in _dwarf_decode_s_leb128". In: *Red Hat Bugzilla*. Apr. 25, 2016. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1330237 (cit. on p. 163).

[141] J. Lieskovsky, "CVE-2011-2503 systemtap: signed module loading race condition". In: *Red Hat Bugzilla*. Dec. 6, 2011. URL: https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2011-2503 (cit. on p. 160).

[142] R. W. Lo, K. N. Levitt, AND R. A. Olsson, "MCF: a malicious code filter". In: *Computers and Security*. Vol. 14. #6. Elsevier Advanced Technology Publications, Jan. 1995, pp. 541–566 (cit. on p. 49).

[143] C. Luo, ET AL., "Verifying pointer safety for programs with unknown calls". In: *Journal of Symbolic Computation*. Vol. 45. #11. Academic Press, Dec. 2010, pp. 1163–1183 (cit. on p. 48).

[144] A. Mallavarapu, "App: aboot: check for integer overflow during sparse image flash". In: *CodeAurora*. Mar. 28, 2015. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=4f829bb52d0338c87bc6fbd0414b258f55cc7c62 (cit. on p. 162).

[145] A. Mariš, "Bug 1385686 – (CVE-2016-8680) CVE-2016-8680 libdwarf: out of bounds heap read in _dwarf_get_abbrev_for_code". In: *Red Hat Bugzilla*. Oct. 17, 2016. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1385686 (cit. on p. 163).

[146] A. Mariš, "Bug 1385690 – (CVE-2016-8681) CVE-2016-8681 libdwarf: heap based buffer overflow in _dwarf_get_abbrev_for_code". In: *Red Hat Bugzilla*. Oct. 17, 2016. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1385690 (cit. on p. 163).

[147] P. Matousek, "CVE-2010-4346 kernel: install_special_mapping skips security_file_mmap check". In: *Red Hat Bugzilla*. Dec. 10, 2010. URL: https://bugzilla.redhat.com/show_bug.cgi?id=662189 (cit. on p. 160).

[148] P. Matousek, "CVE-2014-7840 qemu: insufficient parameter validation during ram load". In: *Red Hat Bugzilla*. Nov. 12, 2014. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1163075 (cit. on p. 160).

[149] A. Matrosov, AND Y. Bulygin, "BIOS and secure boot attacks uncovered". Presented at: *Ekoparty*. 2014 (cit. on p. 31).

[150] P. J. McCann, AND C. Satish, "Packet types: abstract specification of network protocol messages". In: *Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication*. SIGCOMM'00. ACM, 2000, pp. 321–333 (cit. on p. 47).

[151] D. Micay, AND N. Matsakis, *Proposal for regions*. URL: https://github.com/graydon/rust/wiki/Proposal-for-regions (visited on 06/06/2014) (cit. on p. 47).

[152] R. Milewicz, ET AL., "Runtime checking C programs". In: *Symposium on Applied Computing*. SAC'15. ACM, 2015, pp. 2107–2114 (cit. on p. 45).

[153] B. P. Miller, L. Fredriksen, AND B. So, "An empirical study of the reliability of unix utilities". In: *Communications of the ACM*. Vol. 33. #12. ACM, Dec. 1990, pp. 32–44 (cit. on p. 51).

[154] C. Miller, "Don't hassle the Hoff: breaking iOS code signing". Presented at: *SyScan*. Oct. 2011 (cit. on p. 19).

[155] MN, "Uroburos – deeper travel into kernel protection mitigation". In blog: *G DATA Security Blog*. Mar. 7, 2014. URL: https://blog.gdatasoftware.com/blog/article/uroburos-deeper-travel-into-kernel-protection-mitigation.html (visited on 06/05/2014) (cit. on p. 19).

[156] MuscleNerd, AND iPhone Dev Team, "Ultrasn0w". In: *The IPhone Wiki*. 2009. URL: https://www.theiphonewiki.com/wiki/Ultrasn0w (cit. on p. 29).

[157] S. Nagarakatte, M. M. K. Martin, AND S. Zdancewic, "Watchdog: hardware for safe and secure manual memory management and full memory safety". In: *International Symposium on Computer Architecture*. June 2012, pp. 189–200 (cit. on p. 52).

[158] S. Nagarakatte, ET AL., "CETS: compiler enforced temporal safety for C". In: *SIGPLAN Notices*. Vol. 45. #8. ACM, June 2010, pp. 31–40 (cit. on p. 45).

[159] S. Nagarakatte, ET AL., "SoftBound: highly compatible and complete spatial memory safety for C". In: *SIGPLAN Notices*. Vol. 44. #6. ACM, June 2009, pp. 245–258 (cit. on p. 46).

[160] R. Naraine, "UPDATE: ATI driver flaw exposes Vista kernel to attackers". In: *Zdnet*. Aug. 9, 2007 (cit. on p. 159).

[161] R. Naraine, "Vista kernel tampering tool released, then mysteriously disappears". In blog: *Zdnet*. Aug. 7, 2007 (cit. on p. 159).

[162] G. C. Necula, S. McPeak, AND W. Weimer, "CCured: type-safe retrofitting of legacy code". In: *SIGPLAN Notices*. Vol. 37. ACM, 2002, pp. 128–139 (cit. on p. 45).

[163] N. Nethercote, AND J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *SIGPLAN Notices*. Vol. 42. #6. ACM, June 2007, pp. 89–100 (cit. on p. 51).

[164] M.-D. Nguyen, "Bug 21432 – heap buffer overflow in objdump". In: *Sourceware Bugzilla*. Apr. 26, 2017. URL: https://sourceware.org/bugzilla/show_bug.cgi?id=21432 (cit. on p. 164).

[165] nitr0us, "OpenBSD 5.5 – local kernel panic". Exploit Database, Oct. 25, 2014. URL: https://www.exploit-db.com/exploits/35058/ (cit. on p. 162).

[166] C. O'Donnel, *GNU_IFUNC*. URL: https://sourceware.org/glibc/wiki/GNU_IFUNC (cit. on p. 15).

[167] Y. Oiwa, ET AL., "Fail-safe ANSI-C compiler: an approach to making C programs secure". In: *International Conference on Software Security: Theories and Systems*. ISSS'02. Springer-Verlag, 2003, pp. 133–153 (cit. on p. 45).

[168] openbsd, "OpenBSD 5.5 errata 13". Oct. 20, 2014. URL: https://ftp.openbsd.org/pub/OpenBSD/patches/5.5/common/013_kernexec.patch.sig (cit. on p. 162).

[169] T. Pangu, "Hacking from iOS 8 to iOS 9". Presented at: *Power of Community*. 2015 (cit. on pp. 26, 161).

[170] K. Pattabiraman, V. Grover, AND B. G. Zorn, "Samurai: protecting critical data in unsafe languages". In: *SIGOPS European Conference on Computer Systems*. Eurosys'08. ACM, 2008, pp. 219–232 (cit. on p. 45).

[171] L. Pedersen, AND H. Reza, "Using Pit to improve security in low-level programs". In: *The Journal of Supercomputing*. Vol. 53. #3. Kluwer Academic Publishers, Sept. 1, 2010, pp. 394–410 (cit. on p. 46).

[172] T. Pham, "Bug 21157 – objdump segfault – off-by-one read". In: *Sourceware Bugzilla*. Feb. 14, 2017. URL: https://sourceware.org/bugzilla/show_bug.cgi?id=21157 (cit. on p. 163).

[173] A. Pnueli, "The temporal logic of programs". In: *Symposium on Foundations of Computer Science*. SFCS'77. IEEE, 1977, pp. 46–57 (cit. on p. 95).

[174] pod2g, "usb_control_msg(0xA1, 1) exploit". In: *The IPhone Wiki*. 2010. URL: https://www.theiphonewiki.com/wiki/Usb_control_msg(0xA1,_1)_Exploit (cit. on p. 28).

[175] pytey, planetbeing, AND MuscleNerd, "Hacking the iPhone". Presented at: *Chaos Communication Congress*. 2008 (cit. on pp. 24, 29, 99).

[176] "Platform: msm_shared: fix write protect function". In: *CodeAurora*. Ed. by Qualcomm. Sept. 6, 2013. URL: https://www.codeaurora.org/projects/security-advisories/loading-image-data-memory-locations-based-untrusted-header-data-lk-bootloader-cve-2013-2598 (cit. on p. 160).

[177] A. Rahbar, "MS13-098: update to enhance the security of authenticode". In blog: *Microsoft Security Research & Defense*. Dec. 10, 2013. URL: https://blogs.technet.microsoft.com/srd/2013/12/10/ms13-098-update-to-enhance-the-security-of-authenticode/ (cit. on p. 160).

[178] N. Redini, ET AL., "BootStomp: on the security of bootloaders in mobile devices". In: *USENIX Security Symposium*. SSYM'17. USENIX, 2017 (cit. on pp. 48, 90).

[179] Retro-Computing Society of Rhode Island, *Original pdp-11/70 front panel*. 2007 (cit. on p. 15).

[180] *Rust programming languague*. URL: https://www.rust-lang.org (visited on 06/05/2014) (cit. on p. 46).

[181] M. Rutland, "arm64: make sys_call_table const". In: *Linux Git Repository*. Jan. 28, 2015. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c623b33b4e9599c6ac5076f7db7369eb9869aa04 (cit. on p. 162).

[182] J. H. Saltzer, AND M. D. Schroeder, "The protection of information in computer systems". In: *Proceedings of the IEEE*. IEEE, 1975 (cit. on p. 116).

[183] F. B. Schneider, "Enforceable security policies". In: *ACM Transactions on Information and System Security*. Vol. 3. #1. ACM, Feb. 2000, pp. 30–50 (cit. on p. 80).

[184] SecurityFocus, "QNX RTOS malformed ELF binary file local denial of service vulnerability". In: *Bugtraq Mailing List*. Jan. 19, 2009. URL: http://www.securityfocus.com/bid/33352/ (cit. on p. 162).

[185] K. Serebryany, ET AL., "AddressSanitizer: a fast address sanity checker". In: *USENIX Annual Technical Conference*. USENIX, 2012, pp. 309–318 (cit. on p. 51).

[186] J. Seward, AND N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision". In: *USENIX Annual Technical Conference*. ATEC'05. USENIX, 2005, pp. 2–2 (cit. on p. 51).

[187] U. Shankar, ET AL., "Detecting format string vulnerabilities with type qualifiers". In: *USENIX Security Symposium*. SSYM'01. USENIX, 2001 (cit. on p. 46).

[188] R. Shapiro, "Calling putchar() from an ELF weird machine". In: *POC or GTFO*. Vol. 0x02. Dec. 2013 (cit. on p. 37).

[189] R. Shapiro, "Returning from ELF to libc". In: *POC or GTFO*. Vol. 0x0. Aug. 2013 (cit. on p. 37).

[190] R. Shapiro, "The evolution of Linux kernel module signing". In blog: *.bxblogs*, 2014 (cit. on p. 19).

[191] R. Shapiro, S. Bratus, AND S. W. Smith, "Weird machines in ELF: a spotlight on the underappreciated metadata". In: *Workshop on Offensive Technologies*. (Washington, D.C.). WOOT'13. USENIX, 2013 (cit. on pp. 37, 40, 81, 126).

[192] J. Shi, ET AL., "ORIENTAIS: formal verified OSEK/VDX real-time operating system". In: *International Conference on Engineering of Complex Computer Systems*. IEEE, 2012, pp. 293–301 (cit. on p. 50).

[193] M. S. Simpson, AND R. K. Barua, "MemSafe: ensuring the spatial and temporal memory safety of C at runtime". In: *Software: Practice and Experience*. Vol. 43. #1. John Wiley & Sons, Jan. 2013, pp. 93–128 (cit. on p. 45).

[194] M. S. Simpson, "Runtime enforcement of memory safety for the C programming language". PhD thesis. University of Maryland, 2011 (cit. on p. 45).

[195] skape, "Locreate: an anagram for relocate". In: *Uninformed*. Vol. 6. Jan. 2007 (cit. on p. 42).

[196] B. Spengler, "PaX: the guaranteed end of arbitrary code execution". Presented at: *G-Con 2*. Oct. 2003 (cit. on p. 139).

[197] S. Srinivasan, "App: aboot: modify the integer overflow check". In: *CodeAurora*. Dec. 16, 2013. URL: https://source.codeaurora.org/quic/la/kernel/lk/commit/?id=ce2a0ea1f14298abc83729f3a095adab43342342 (cit. on p. 161).

[198] P. Starzetz, "Linux ELF loader vulnerabilities". In: *Full Disclosure Mailing List*. Nov. 10, 2004. URL: http://seclists.org/fulldisclosure/2004/Nov/329 (cit. on p. 159).

[199] R. E. Strom, AND S. Yemini, "Typestate: a programming language concept for enhancing software reliability". In: *Transactions on Software Engineering*. Vol. 12. #1. IEEE, Jan. 1986, pp. 157–171 (cit. on p. 81).

[200] A. Tereshkin, AND R. Wojtczuk, "Introducing ring -3 rootkits". Presented at: *Black Hat USA*. 2009 (cit. on p. 31).

[201] Texas Instruments, *AM/DM37x multimedia device technical reference manual*. Sept. 2012 (cit. on p. 61).

[202] Texas Instruments, *AM335x Sitara processors technical reference manual*. Feb. 2015 (cit. on pp. 108, 153).

[203] *The proof*. Tech. rep. seL4. URL: http://sel4.systems/Info/FAQ/proof.pml (visited on 06/10/2017) (cit. on p. 50).

[204] K. Thompson, "Reflections on trusting trust". In: *Communications of the ACM*. Vol. 27. #8. ACM, Aug. 1984, pp. 761–763 (cit. on p. 35).

[205] Tianocore, *EDK II build specification*. Rev. 1.24. Dec. 2014 (cit. on p. 33).

[206] C. Tice, ET AL., "Enforcing forward-edge control-flow integrity in GCC & LLVM". In: *USENIX Security Symposium*. SSYM'14. USENIX, 2014, pp. 941–955 (cit. on p. 45).

[207] M. Tofte, AND J.-P. Talpin, "Region-based memory management". In: *Information and Computation*. Vol. 132. #2. Elsevier, 1997, pp. 109–176 (cit. on p. 46).

[208] M. Tofte, ET AL., "A retrospective on region-based memory management". In: *Higher Order Symbolic Computation*. Vol. 17. #3. Kluwer Academic Publishers, Sept. 2004, pp. 245–265 (cit. on p. 46).

[209] Tunnelblick, "Security problem (race condition in SUID code)". In: *Tunnelblick Issues*. July 21, 2012. URL: https://github.com/Tunnelblick/Tunnelblick/issues/212 (cit. on p. 160).

[210] Unified Extensible Firmware Interface Forum, *UEFI shell specification*. Version 2.2 (cit. on p. 33).

[211] UpstandingHackers, *Hammer*. GitHub repository. URL: https://github.com/UpstandingHackers/hammer (visited on 06/06/2014) (cit. on p. 47).

[212] J. Vanegue, AND S. K. Lahiri, "Modern static security checking of C/C++ programs". Presented at: *REcon*. June 14, 2012 (cit. on p. 48).

[213] A. Vecerka, "Linear and temporal logic programming language". In: *WSEAS International Conference on Applied Computer Science*. ACS'06. World Scientific, Engineering Academy, AND Society, 2006, pp. 226–230 (cit. on p. 50).

[214] R. Venkitaraman, AND G. Gupta, "Static program analysis of embedded executable assembly code". In: *International Conference on Compilers, Architecture, and Synthesis For Embedded Systems*. CASES'04. ACM, 2004, pp. 157–166 (cit. on p. 48).

[215] *Verifying boot*. Tech. rep. Google. URL: https://source.android.com/security/verifiedboot/verified-boot (visited on 06/10/2017) (cit. on p. 48).

[216] M. Voelter, ET AL., "Mbeddr: an extensible C-based programming language and IDE for embedded systems". In: *Systems, Programming, and Applications: Software For Humanity*. SPLASH'12. ACM, 2012, pp. 121–140 (cit. on p. 46).

[217] US-CERT, *Vulnerability note VU#533140*. Jan. 5, 2015 (cit. on pp. 4, 160).

[218] US-CERT, *Vulnerability note VU#552286*. Aug. 7, 2014 (cit. on pp. 4, 160).

[219] US-CERT, *Vulnerability note VU#631788*. Mar. 20, 2015 (cit. on p. 161).

[220] US-CERT, *Vulnerability note VU#758382*. June 9, 2014 (cit. on pp. 4, 160).

[221] US-CERT, *Vulnerability note VU#766164*. Jan. 5, 2015 (cit. on pp. 4, 161).

[222] US-CERT, *Vulnerability note VU#577140*. July 30, 2015 (cit. on p. 4).

[223] US-CERT, *Vulnerability note VU#912156*. Aug. 15, 2013 (cit. on pp. 4, 160).

[224] T. Wahg, H. Xu, AND X. Chen, "Pangu 9 internals". Presented at: *Black Hat USA*. 2016 (cit. on p. 161).

[225] D. Wang, ET AL., "Swiping through modern security features". Presented at: *HITB Amsterdam*. 2013 (cit. on pp. 19, 25, 99, 160).

[226] C. Wetherell, *Etudes for programmers*. Prentice-Hall, 1978 (cit. on p. 10).

[227] E. Witchel, J. Cates, AND K. Asanović, "Mondrian memory protection". In: *SIGARCH Computer Architecture News*. Vol. 30. #5. ACM, Dec. 2002, pp. 304–316 (cit. on p. 139).

[228] R. Wojtczuk, AND C. Kallenberg, "Attacks on UEFI security". Presented at: *Chaos Communication Congress*. 2014 (cit. on pp. 31, 161).

[229] R. Wojtczuk, AND J. Rutkowska, "Attacking Intel trusted execution technology". Presented at: *Black Hat DC*. Feb. 2009 (cit. on p. 31).

[230] R. Wojtczuk, AND J. Rutkowska, *Attacking SMM memory via Intel CPU cache poisoning*. Tech. rep. Invisible Things Lab, 2009 (cit. on p. 31).

[231] R. Wojtczuk, AND A. Tereshkin, "Attacking Intel BIOS". Presented at: *Black Hat USA*. July 20, 2009 (cit. on pp. 31, 164).

[232] Xen, "[Xen-bugs] [bug 1068] new: guest root can escape to domain through grub.conf and pygrub". In: *Xen-Bugs Mailing List*. Sept. 22, 2007. URL: http://old-list-archives.xenproject.org/archives/html/xen-bugs/2007-09/msg00040.html (cit. on p. 159).

[233] Xen, "Delimiter injection vulnerabilities in pygrub". In: *Xen Security Advisory*. Nov. 22, 2016. URL: http://xenbits.xen.org/xsa/advisory-198.html (cit. on p. 163).

[234] W. Xu, AND a. R. S. Daniel C. DuVarney, "An efficient and backwards-compatible transformation to ensure memory safety of C programs". In: *SIGSOFT Software Engineering Notes*. Vol. 29. #6. ACM, Oct. 2004, pp. 117–126 (cit. on p. 45).

[235] E. Yahav, ET AL., "Verifying temporal heap properties specified via evolution logic". In: *European Conference on Programming*. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 204–222 (cit. on p. 50).

[236] F. Yamaguchi, ET AL., "Modeling and discovering vulnerabilities with code property graphs". In: *IEEE Security & Privacy*. IEEE, 2014 (cit. on p. 48).

[237] H. Yang, ET AL., "Scalable shape analysis for systems code". In: *Computer Aided Verification*. Ed. by A. Gupta, AND S. Malik. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 385–398 (cit. on p. 48).

[238] J. Zaddach, ET AL., "Avatar: a framework to support dynamic security analysis of embedded systems' firmwares". In: *Network & Distributed System Security Symposium*. NDSS'14. Internet Society, Feb. 2014 (cit. on pp. 51, 67).

[239] F. Zhou, ET AL., "SafeDrive: safe and recoverable extensions using language-based techniques". In: *Symposium on Operating Systems Design and Implementation*. OSDI'06. USENIX, 2006, pp. 45–60 (cit. on p. 46).