Formal Methods in System Design (2021) 57:34–52
https://doi.org/10.1007/s10703-020-00344-2

Model checking boot code from AWS data centers

Byron Cook[1,2] · Kareem Khazem[1,2] · Daniel Kroening[3] · Serdar Tasiran[1] · Michael Tautschnig[1,4] · Mark R. Tuttle[1]

Published online: 15 April 2020

Abstract
This paper describes our experience with symbolic model checking in an industrial setting. We have proved that the initial boot code running in data centers at Amazon Web Services is memory safe, an essential step in establishing the security of any data center. Standard static analysis tools cannot be easily used on boot code without modification owing to issues not commonly found in higher-level code, including memory-mapped device interfaces, byte-level memory access, and linker scripts. This paper describes automated solutions to these issues and their implementation in the C Bounded Model Checker (CBMC). CBMC is now the first source-level static analysis tool to extract the memory layout described in a linker script for use in its analysis.

Keywords Formal verification · Model checking · CBMC · Boot code · Firmware · Linker script · Amazon Web Services (AWS)

1 Introduction

Boot code is the first code to run in a data center; thus, the security of a data center depends on the security of the boot code. It is hard to demonstrate boot code security using standard techniques, as boot code is difficult to test and debug, and boot code must run without the support of common security mitigations available to the operating system and user applications. This industrial experience report describes work to prove the memory safety of initial boot code running in data centers at Amazon Web Services (AWS).

We describe the challenges we faced analyzing AWS boot code, some of which render existing approaches to software verification unsound or imprecise. These challenges include

1. memory-mapped input/output (MMIO) for accessing devices,
2. device behavior behind these MMIO regions,

✉ Kareem Khazem
karkhaz@karkhaz.com

1 Amazon Web Services, Seattle, USA

2 University College London, London, UK

3 University of Oxford, Oxford, UK

4 Queen Mary University of London, London, UK

3. byte-level memory access as the dominant form of memory access, and
4. linker scripts used during the build process.

Not handling MMIO or linker scripts results in imprecision (false positives), and not modeling device behavior is unsound (false negatives).

We describe the solutions to these challenges that we developed. We implemented our solutions in the C Bounded Model Checker (CBMC) [20]. We achieve soundness with CBMC by fully unrolling loops in the boot code. Our solutions automate boot code verification and require no changes to the code being analyzed. This makes our work particularly well-suited for deployment in a continuous validation environment to ensure that memory safety issues do not reappear in the code as it evolves during development. We use CBMC, but any other bit-precise, sound, automated static analysis tool could be used.

2 Related work

There are many approaches to finding memory safety errors in low-level code, from fuzzing [2] to static analysis [24,30,36,54] to deductive verification [21,32].

A key aspect of our work is soundness and precision in the presence of very low-level details. Furthermore, full automation is essential in our setting to operate in a continuous validation environment. This makes some form of model checking most appealing.

CBMC is a bounded model checker for C, C++, and Java programs, available on GitHub [13]. It features bit-precise reasoning, and it verifies array bounds (buffer overflows), pointer safety, arithmetic exceptions, and assertions in the code. A user can bound the model checking done by CBMC by specifying for a loop a maximum number of iterations of the loop. CBMC can check that it is impossible for the loop to iterate more than the specified number of times by checking a *loop-unwinding assertion*. CBMC is sound when all loop-unwinding assertions hold. Loops in boot code typically iterate over arrays of known sizes, making it possible to choose loop unwinding limits such that all loop-unwinding assertions hold (see Sect. 5.6). BLITZ [16] or F-Soft [34] could be used in place of CBMC. SATABS [19], Ufo [3], Cascade [58], Blast [8], CPAchecker [9], Corral [31,39,40], and others [18,43] might even enable unbounded verification. Our work applies to any sound, bit-precise, automated tool.

Note that boot code makes heavy use of pointers, bit vectors, and arrays, but not the heap. Thus, memory safety proof techniques based on three-valued logic [41] or separation logic as in [7] or other techniques [1,22] that focus on the heap are less appropriate since boot code mostly uses simple arrays.

KLEE [12] is a symbolic execution engine for C that has been used to find bugs in firmware. Davidson et al. [25] built the tool FIE on top of KLEE for detecting bugs in firmware programs for the MSP430 family of microcontrollers for low-power platforms, and applied the tool to nearly a hundred open source firmware programs for nearly a dozen versions of the microcontroller to find bugs like buffer overflow and writing to read-only memory. Corin and Manzano [23] used KLEE to do taint analysis and prove confidentiality and integrity properties. KLEE and other tools like SMACK [49] based on the LLVM intermediate representation do not currently support the linker scripts that are a crucial part of building boot code (see Sect. 4.5). They support partial linking by concatenating object files and resolving symbols, but fail to make available to their analysis the addresses and constants assigned to symbols in linker scripts, resulting in an imprecise analysis of the code.

S^{2}E [15] is a symbolic execution engine for x86 binaries built on top of the QEMU [6] virtual machine and KLEE. S^{2}E has been used on firmware. Parvez et al. [46] use symbolic execution to generate inputs targeting a potentially buggy statement for debugging. Kuznetsov et al. [38] used a prototype of S^2E to find bugs in Microsoft device drivers. Zaddach et al. [59] built the tool Avatar on top of S^2E to check security of embedded firmware. They test firmware running on top of actual hardware, moving device state between the concrete device and the symbolic execution. Bazhaniuk et al. [5,28] used S^2E to search for security vulnerabilities in interrupt handlers for System Management Mode on Intel platforms. Experts can use S^2E on firmware. One can model device behavior (see Sect. 4.2) by adding a device model to QEMU or using the signaling mechanism used by S^2E during symbolic execution. One can declare an MMIO region (see Sect. 4.1) by inserting it into the QEMU memory hierarchy. Both require understanding either QEMU or S^2E implementations. Our goal is to make it as easy as possible to use our work, primarily by way of automation.

Ferreira et al. [29] verify a task scheduler for an operating system, but that is high in the software stack. Klein et al. [35] prove the correctness of the seL4 kernel, but that code was written with the goal of proof. Dillig et al. [26] synthesize guards ensuring memory safety in low-level code, but our code is written by hand. Rakamarić and Hu [50] developed a conservative, scalable approach to memory safety in low-level code, but the models there are not tailored to our code that routinely accesses memory by an explicit integer-valued memory address. Redini et al. [51] built a tool called BootStomp on top of angr [57], a framework for symbolic execution of binaries based on a symbolic execution engine for the VEX intermediate representation for the Valgrind project, resulting in a powerful testing tool for boot code, but it is not sound.

3 Boot code

We define *boot code* to be the code in a cloud data center that runs from the moment the power is turned on until the BIOS starts. It runs before the operating system's boot loader that most people are familiar with. A key component to ensuring high confidence in data center security is establishing confidence in boot code security. Enhancing confidence in boot code security is a challenge because of unique properties of boot code not found in higher-level software. We now discuss these properties of boot code, and a path to greater confidence in boot code security.

3.1 Boot code implementation

Boot code starts a sequenced boot flow [4] in which each stage locates, loads, and launches the next stage. The boot flow in a modern data center proceeds as follows: (1) When the power is turned on, before a single instruction is executed, the hardware interrogates banks of fuses and hardware registers for configuration information that is distributed to various parts of the platform. (2) *Boot code* starts up to boot a set of microcontrollers that orchestrate bringing up the rest of the platform. In a cloud data center, some of these microcontrollers are feature-rich cores with their own devices used to support virtualization. (3) The BIOS familiar to most people starts up to boot the cores and their devices. (4) A boot loader for the hypervisor launches the hypervisor to virtualize those cores. (5) A boot loader for the operating system launches the operating system itself. The security of each stage, including operating system launched for the customer, depends on the integrity of all prior stages [27].

Ensuring boot code security using traditional techniques is hard. Visibility into code execution can only be achieved via debug ports, with almost no ability to single-step the code for debugging. UEFI (Unified Extensible Firmware Interface) [56] provides an elaborate infrastructure for debugging BIOS, but not for the boot code below BIOS in the software stack. Instrumenting boot code may be impossible because it can break the build process: the increased size of instrumented code can be larger than the size of the ROM targeted by the build process. Extracting the data collected by instrumentation may be difficult because the code has no access to a file system to record the data, and memory available for storing the data may be limited.

Static analysis is a relatively new approach to enhancing confidence in boot code security. As discussed in Sect. 2, most work applying static analysis to boot code applies technology like symbolic execution to binary code, either because the work strips the boot code from ROMs on shipping products for analysis and reverse engineering [38,51], or because code like UEFI-based implementations of BIOS loads modules with a form of dynamic linking that makes source code analysis of any significant functionality impossible [5,28]. But with access to the source code—source code without the complexity of dynamic linking—meaningful static analysis at the source code level is possible.

3.2 Boot code security

Boot code is a foundational component of data center security: it controls what code is run on the server. Attacking boot code is a path to booting your own code, installing a persistent root kit, or making the server unbootable. Boot code also initializes devices and interfaces directly with them. Attacking boot code can also lead to controlling or monitoring peripherals like storage devices.

The input to boot code is primarily configuration information. The run-time behavior of boot code is determined by configuration information in fuses, hardware straps, one-time programmable memories, and ROMs.

From a security perspective, boot code is susceptible to a variety of events that could set the configuration to an undesirable state. To keep any malicious adversary from modifying this configuration information, the configuration is usually locked or otherwise write-protected. Nonetheless, it is routine to discover during hardware vetting before placing hardware on a data center floor that some BIOS added by a supplier accidentally leaves a configuration register unlocked after setting it. In fact, configuration information can be intentionally unlocked for the purpose of patching and then be locked again. Any bug in a patch or in a patching mechanism has the potential to leave a server in a vulnerable configuration. Perhaps more likely than anything is a simple configuration mistake at installation. We want to know that no matter how a configuration may have been corrupted, the boot code will operate as intended and without latent exposures for potential adversaries.

The attack surface we focus on in this paper is memory safety, meaning there are no buffer overflows, no dereferencing of null pointers, and no pointers pointing into unallocated regions of memory. Code written in C is known to be at risk for memory safety, and boot code is almost always written in C, in part because of the direct connection between boot code and the hardware, and sometimes because of space limitations in the ROMs used to store the code.

There are many techniques for protecting against memory safety errors and mitigating their consequences at the higher levels of the software stack. Languages other than C are less prone to memory safety errors. Safe libraries can do bounds checking for standard library functions. Compiler extensions to compilers like `gcc` and `clang` can help detect buffer overflow when it happens (which is different from keeping it from happening). Address space layout randomization makes it harder for the adversary to make reliable use of a vulnerability. None of these mitigations, however, apply to firmware. Firmware is typically built using the tool chain that is provided by the manufacturer of the microcontroller, and firmware typically runs before the operating system starts, without the benefit of operating system support like a virtual machine or randomized memory layout.

4 Boot code verification challenges

Boot code poses challenges to the precision, soundness, and performance of any analysis tool. The C standard [33] says, "A volatile declaration may be used to describe an object corresponding to an MMIO port" and "what constitutes an access to an object that has volatile-qualified type is implementation-defined." Any tool that seeks to verify boot code must provide means to model what the C standard calls *implementation-defined behavior*. Of all such behavior, MMIO and device behavior are most relevant to boot code. In this section, we discuss these issues and the solutions we have implemented in CBMC.

4.1 Memory-mapped I/O

Boot code accesses a device through *memory-mapped input/output* (MMIO). Registers of the device are mapped to specific locations in memory. Boot code reads or writes a register in the device by reading or writing a specific location in memory. If boot code wants to set the second bit in a configuration register, and if that configuration register is mapped to the byte at location 0x1000 in memory, then the boot code sets the second bit of the byte at 0x1000. The problem posed by MMIO is that there is no declaration or allocation in the source code specifying this location 0x1000 as a valid region of memory. Nevertheless accesses within this region are valid memory accesses, and should not be flagged as an out-of-bounds memory reference. This is an example of implementation-defined behavior that must be modeled to avoid reporting false positives.

To facilitate analysis of low-level code, we have added to CBMC a built-in function

```
__CPROVER_allocated_memory(address, size)
```

to mark ranges of memory as valid. Accesses within this region are exempt from the out-of-bounds assertion checking that CBMC would normally do. The function declares the half-open interval [`address`, `address` + `size`) as valid memory that can be read and written. This function can be used anywhere in the source code, but is most commonly used in the test harness. (CBMC, like most program analysis approaches, uses a test harness to drive the analysis.)

4.2 Device behavior

An MMIO region is an interface to a device. It is unsound to assume that the values returned by reading and writing this region of memory follow the semantics of ordinary read-write memory. Imagine a device that can generate unique ids. If the register returning the unique id is mapped to the byte at location 0x1000, then reading location 0x1000 will return a different value every time, even without intervening writes. These side effects have to be modeled. One easy approach is to 'havoc' the device, meaning that writes are ignored and reads return nondeterministic values. This is sound, but may lead to too many false positives. We can model the device semantics more precisely, using one of the options described below.

If the device has an API, we havoc the device by making use of a more general functionality we have added to CBMC. We have added a command-line option

```
--remove-function-body device_access
```

to CBMC's `goto-instrument` tool. When used, this will drop the implementation of the function `device_access` from compiled object code. If there is no other definition of `device_access`, CBMC will model each invocation of `device_access` as returning an unconstrained value of the appropriate return type. Now, to havoc a device with an API that includes a read and write method, we can use this command-line option to remove their function bodies, and CBMC will model each invocation of read as returning an unconstrained value.

At link time, if another object file, such as the test harness, provides a second definition of `device_access`, CBMC will use this definition in its place. Thus, to model device semantics more precisely, we can provide a device model in the test harness by providing implementations of (or approximations for) the methods in the API.

If the device has no API, meaning that the code refers directly to the address in the MMIO region for the device without reference to accessor functions, we have another method. We have added two function symbols

```
__CPROVER_mm_io_r(address, size)
__CPROVER_mm_io_w(address, size, value)
```

to CBMC to model the reading or writing of an address at a fixed integer address. If the test harness provides implementations of these functions, CBMC will use these functions to model every read or write of memory. For example, defining

```
char __CPROVER_mm_io_r(void *a, unsigned s) {
  if(a == 0x1000)
    return 2;
  else
    return nondet_char();
}
```

will return the value `2` upon any access at address 0x1000, and return a non-deterministic value in all other cases.

In both cases—with or without an API—we can thus establish sound and, if needed, precise analysis about an aspect of implementation-defined behavior.

4.3 Byte-level memory access

It is common for boot code to access memory a byte at a time, and to access a byte that is not part of any variable or data structure declared in the program text. Accessing a byte in an MMIO region is the most common example. Boot code typically accesses this byte in memory by computing the address of the byte as an integer value, coercing this integer to a pointer, and dereferencing this pointer to access that byte. Boot code references memory by this kind of explicit address far more frequently than it references memory via some explicitly allocated variable or data structure. Any tool analyzing boot code must have a method for reasoning efficiently about accessing an arbitrary byte of memory.

A natural model for memory is as an array of bytes [55]. This enables CBMC to precisely reason about memory accesses in presence of pointer arithmetic, at the expense of not using abstractions like those provided by Separation Logic [52]. Any decision procedure that has a well-engineered implementation of a theory of arrays is likely to do a good job of modeling byte-level memory access. We improved CBMC's decision procedure for arrays to follow the state-of-the-art algorithm [17,37]. The key data structure is a weak equivalence graph whose vertices correspond to array terms. Given an equality $a = b$ between two array terms a and b, add an unlabeled edge between a and b. Given an update $a\{i \leftarrow v\}$ of an array term a, add an edge labeled i between a and $a\{i \leftarrow v\}$. Two array terms a and b are weakly equivalent if there is a path from a to b in the graph, and they are equal at all indices except those updated along the path. This graph is used to encode constraints on array terms for the solver. For simplicity, our implementation generates these constraints eagerly.

4.4 Memory copying

One of the main jobs of any stage of the boot flow is to copy the next stage into memory, usually using some variant of `memcpy`. Any tool analyzing boot code must have an efficient model of `memcpy`. Modeling `memcpy` as a loop iterating through a thousand bytes of memory leads to performance problems during program analysis. We added to CBMC an improved model of the `memset` and `memcpy` library functions.

Boot code has no access to a C library. In our case, the boot code shipped an iterative implementation of `memset` and `memcpy`. CBMC's model of the C library previously also used an iterative model. We replaced this iterative model of `memset` and `memcpy` with a single array operation that can be handled efficiently by the decision procedure at the back end. We instructed CBMC to replace the boot code implementations with the CBMC model using the `-remove-function-body` command-line option described in Sect. 4.2.

4.5 Linker scripts

Boot code and other low-level programs control and access their memory layout using *custom linker scripts*, which is a mechanism external to the source code. This means that current static analysis tools, which only read and analyze the source code, do not correctly model the memory layout of such programs. This is a hindrance to verifying the memory safety of boot code, in which buffers are commonly defined in linker scripts. Without understanding what the runtime memory layout will be, static analysis tools falsely report that boot code is unsafe. In this section, we present informal description of linker scripts, how programmers use them to control memory layout, and why this poses problems for static analysis. We describe how we overcame these problems in Sect. 5.5.

The left-hand side of Fig. 1 depicts a short program written in C. This code is memory safe when linked according to the linker script on the right-hand side of the figure, but current static analyzers—that consider only the C code—cannot show that it is memory safe.

This program contains two unusual features:

- The program declares several variables as `extern`. Normally, this means that the variables must be *defined* (not just declared) in some other source file. However, this program contains no other source files, so those variables will apparently not be allocated. Without the information in the linker script, attempts to analyze or link this program will fail.
- The number of bytes to copy is the *address*, not the value, of `rodata_size`. This variable has not been allocated because it was declared `extern`, so it seemingly does not even have an address.

```
 1  /* main.c */
 2
 3  #include <string.h>
 4
 5  extern char[] rodata_start;
 6  extern char[] rodata_size;
 7  extern char[] data_start;
 8
 9  int main()
10  {
11    memcpy(
12      &data_start,
13      &rodata_start,
14      (size_t)&rodata_size);
15  }
```

```
 1  /* link.ld */
 2
 3  SECTIONS {
 4    .rodata : {
 5      rodata_start = .;
 6      *(.rodata)
 7    }
 8    rodata_size =
 9      SIZEOF(.rodata);
10
11    .data : {
12      data_start = .;
13      *(.data)
14      data_end = .;
15    }
16  }
```

Fig. 1 A program using variables whose addresses are defined in a linker script

To explain these apparent contradictions, we present an overview of how linking works and how the linker script in Fig. 1 makes the C program valid.

Compilers, linkers, and linker scripts A compiler transforms a single source file into an object file. An object file contains machine code organized into several *sections*; each section has a name and contains machine code for a specific purpose. Conventionally, object files emitted by a compiler contain the following sections, among many others:

– the `.text` section, containing executable machine code;
– the `.data` contains static program data (i.e. static and global variables in a C program) that the executable code can read and write at runtime;
– `.rodata` is similar to `.data`, but is loaded into a read-only memory segment at runtime.

A linker joins several object files into a single object. By default, it joins identically-named sections in each of the input files into a single contiguous section in the output object, as depicted in Fig. 2.

The functionality of most programs does not depend on the exact layout of the program's own machine code and data. The default behavior of a linker therefore suffices to correctly link the majority of programs. However, the functionality of low-level software (like boot loaders, kernels, and firmware) often does depend on the exact layout of structures in the object file. For example, [10] discusses the custom sections that are present in the Linux kernel's object file, and [44,45,47,48] describe various link-level optimizations that are applied to the kernel. In such cases, programmers manually control the object file's layout by running the linker according to a hand-written linker script. Linker scripts are imperative programs that can direct the linker to place sections at particular addresses; control the access permissions of those sections; and place *symbols* (named addresses that demarcate code or data) at arbitrary points in the object file. Figure 3 depicts an object file a linker might emit when linking the program in Fig. 1, following the script in that figure.

Lines 4 and 11 of the linker script instruct the linker to create sections called `.rodata` and `.data` in the object file. These sections are populated with machine code from the object files that the linker gets as input: line 6 says to place the machine code from all input `.rodata` sections into the output `.rodata` section, and line 13 does the same for the `.data` section. Figure 3 depicts those two sections as gray regions in the object file. In addition, the linker script adds *symbols* (named addresses) to the object file. The statement on line

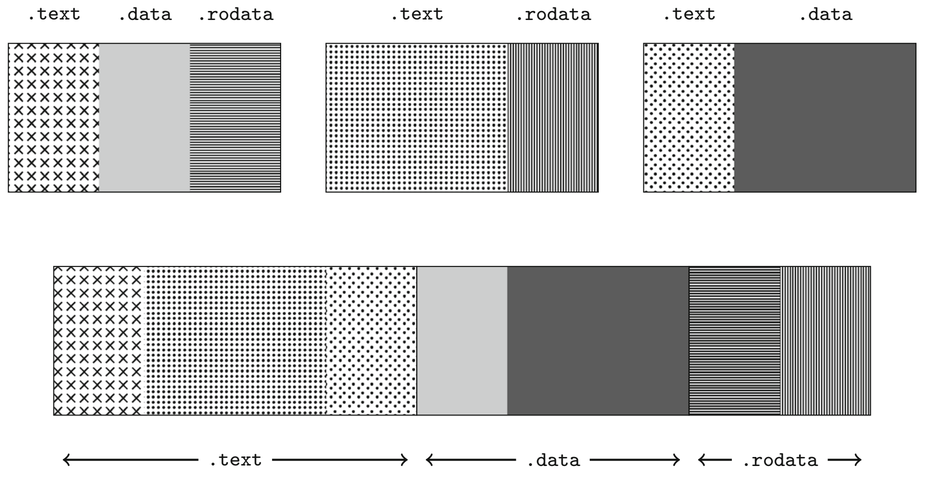

Fig. 2 Default linkage of several object files. The three object files in the top row, given as input to the linker, are linked into the output object at the bottom. The `.text` input sections (patterned) are joined into a single contiguous section, as are the `.data` input sections (plain) and `.rodata` input sections (striped)

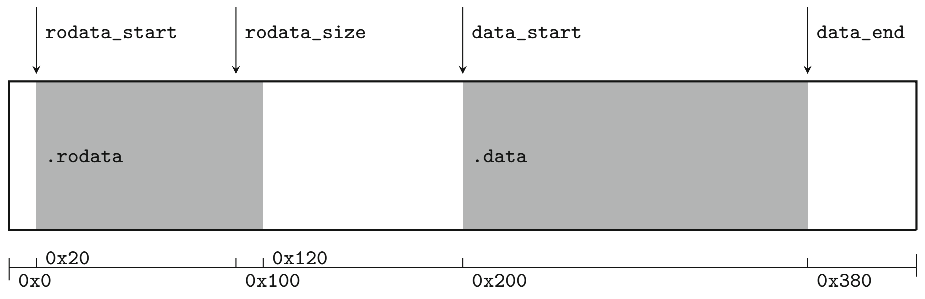

Fig. 3 Object file emitted by a linker following the script in Fig. 1

9 sets the address of the `rodata_size` symbol to be the number of bytes in the `.rodata` section. The period ("`.`") expression in a linker script evaluates to the current address in the object file, and increases as the linker populates the file with machine code. Thus, the statement "`rodata_start = .;`" on line 5 means "place a symbol called `rodata_start` in the object file at the current address." If that statement immediately follows the opening of a new section—as it does in Fig. 1—then it serves to demarcate the start of that section. Similarly, the statement on line 14 comes just after the linker wrote the machine code of the `.data` section and just before the curly brace that terminates the section. This means that the `data_end` symbol demarcates the end of the `.data` section.

The previous paragraph illustrates how the program in Fig. 1 successfully links when the linker runs according to the script, despite the apparent problems that we highlighted. The variables in the program are `extern`-declared because they are defined in the linker script, rather than in a source file, and the linker sets their addresses. Reading those variable's addresses in the program is thus safe. It is also safe to copy the memory region starting at the address of `rodata_start`, because that memory region is a section of machine code that the linker allocated, and we do not stray beyond the boundary of that region because the number of bytes we copy is exactly the section's size. The program is therefore safe, but only when linked in accordance with the script. It is therefore necessary to analyze the script, as well as the program itself, to decide the memory safety of this program.

Linker scripts pose challenges for static analysis. Current static analysis tools parse and analyze only the source code, neglecting to read the linker script if one exists. This means that they lack the following information, referring to Fig. 1 but without loss of generality:

1. The numerical addresses of `rodata_start`, `rodata_size`, and `data_start`, which will only be known at link time, i.e., after the compiler has generated object files as shown in Fig. 3;
2. The fact that the symbol `rodata_start` demarcates a valid region of memory that is `rodata_size` bytes long, and that `data_start` similarly demarcates the start of another valid memory region—which is information contained in the linker script.

The numerical addresses of symbols can be read directly from the object file, if one exists, using a standard utility like `readelf(1)` or `objdump(1)`. However, the relationship between sections and symbols are not written to the object file. There may in fact be a symbol called `rodata_start` that has the same address as the beginning of the `rodata` section in the object file. However, the linker may have placed the symbol there coincidentally; we cannot assume that the program author explicitly demarcated the section with a symbol unless the linker script says so. To discover the symbols that demarcate object file sections, static analysis tools must therefore parse the linker script and incorporate that information into their analyses.

Without this information, static analyzers assume that the addresses of `rodata_start`, `rodata_size`, and `data_start` are set to an arbitrary value. This admits the possibility of a memory safety violation or other undefined behavior, for example

– if either of the memory regions at those addresses are not actually allocated;
– if the memory regions are allocated but overlap (since calling `memcpy(3)` on overlapping memory regions is undefined behavior);
– if `rodata_size` is larger than the size of the region allocated at `data_start`.

This issue prevented us from verifying the safety of our boot code, which uses a linker script, using existing tools. We describe how we extended CBMC to overcome this issue in Sect. 5.5.

5 Industrial boot code verification

In this section, we describe our experience proving memory safety of boot code running in an AWS data center. We give an exact statement of what we proved, we point out examples of the verification challenges mentioned in Sect. 4 and our solutions, and we go over the test harness and the results of running CBMC.

We use CBMC to prove that 783 lines of AWS boot code are memory safe. Soundness of this proof by bounded model checking is achieved by having CBMC check its loop unwinding assertions (that loops have been sufficiently unwound). This boot code proceeds in two stages, as illustrated in Fig. 4. The first stage prepares the machine, loads the second stage from a boot source, and launches the second stage. The behavior of the first stage is controlled by configuration information in hardware straps and one-time-programmable memory (OTP), and by device configuration. We show that no configuration will induce a memory safety error in the stage 1 boot code. More precisely, we prove:

Assuming

– a buffer for stage 2 code and a temporary buffer are both 1024 bytes,

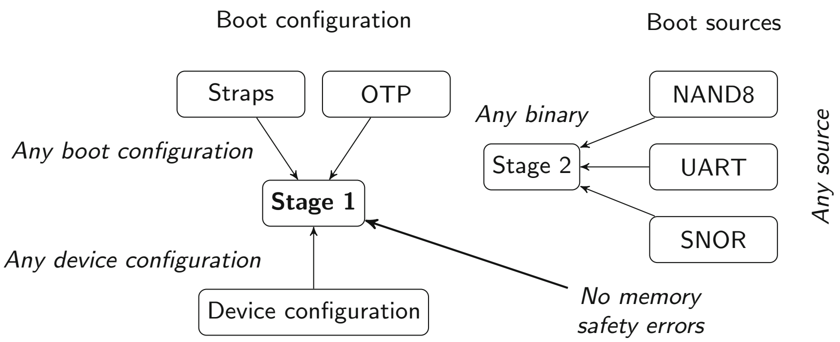

Fig. 4 Boot code is free of memory safety errors

- the cryptographic, CRC computation, and printf methods have no side effects and can return unconstrained values,
- the CBMC model of `memcpy` and `memset`, and
- ignoring a loop that flashes the console lights when boot fails

then

- for every boot configuration,
- for every device configuration,
- for each of the three boot sources, and
- for every stage 2 binary,

the stage 1 boot code will not exhibit any memory safety errors.

Due to the second and third assumptions, we may be missing memory safety errors in these simple procedures. Memory safety of these procedures can be established in isolation. We find all memory safety errors in the remainder of the code, however, because making buffers smaller increases the chances they will overflow, and allowing methods to return unconstrained values increases the set of program behaviors considered.

The code we present in this section is representative of the code we analyzed, but the actual code is proprietary and not public. The open-source project rBoot [11] is 700 lines of boot code available to the public that exhibits most of the challenges we now discuss.

5.1 Memory-mapped I/O

MMIO regions are not explicitly allocated in the code, but the addresses of these regions appear in the header files. For example, an MMIO region for the hardware straps is given with

```
#define REG_BASE        (0x1000)
#define REG_BOOT_STRAP  (REG_BASE + 0x110)
#define REG_BOOT_CONF   (REG_BASE + 0x124)
```

Each of the last two macros denotes the start of a different MMIO region, leaving 0x14 bytes for the region named `REG_BOOT_STRAP`. Using the builtin function added to CBMC (Sect. 4.1), we declare this region in the test harness with

```
__CPROVER_allocated_memory(REG_BOOT_STRAP, 0x14);
```

5.2 Device behavior

All of the devices accessed by the boot code are accessed via an API. For example, the API for the UART is given by

```
int UartInit(UART_PORT port, unsigned int baudRate);
void UartWriteByte(UART_PORT port, uint8_t byte);
uint8_t UartReadByte(UART_PORT port);
```

In this work, we havoc all of the devices to make our result as strong as possible. In other words, our device model allows a device read to return any value of the appropriate type, and still we can prove that (even in the context of a misbehaving device) the boot code does not exhibit a memory safety error. Because all devices have an API, we can havoc the devices using the command line option added to CBMC (Sect. 4.2), and invoke CBMC with

```
--remove-function-body UartInit
--remove-function-body UartReadByte
--remove-function-body UartWriteByte
```

5.3 Byte-level memory access

All devices are accessed at the byte level by computing an integer-valued address and coercing it to a pointer. For example, the following code snippets from `BootOptionsParse` show how reading the hardware straps from the MMIO region discussed above translates into a byte-level memory access.

```
#define REG_READ(addr) (*(volatile uint32_t*)(addr))

regVal = REG_READ(REG_BOOT_STRAP);
```

In CBMC, this translates into an access into an array modeling memory at location 0x1000 + 0x110. Our optimized encoding of the theory of arrays (Sect. 4.3) enables CBMC to reason more efficiently about this kind of construct.

5.4 Memory copying

The `memset` and `memcpy` procedures are heavily used in boot code. For example, the function used to copy the stage 2 boot code from flash memory amounts to a single, large `memcpy`.

```
int SNOR_Read(unsigned int address,
              uint8_t* buff,
              unsigned int numBytes) {
  ...
  memcpy(buff,
         (void*)(address + REG_SNOR_BASE_ADDRESS),
         numBytes);
  ...
}
```

CBMC reasons more efficiently about this kind of code due to our loop-free model of `memset` and `memcpy` procedures as array operations (Sect. 4.4).

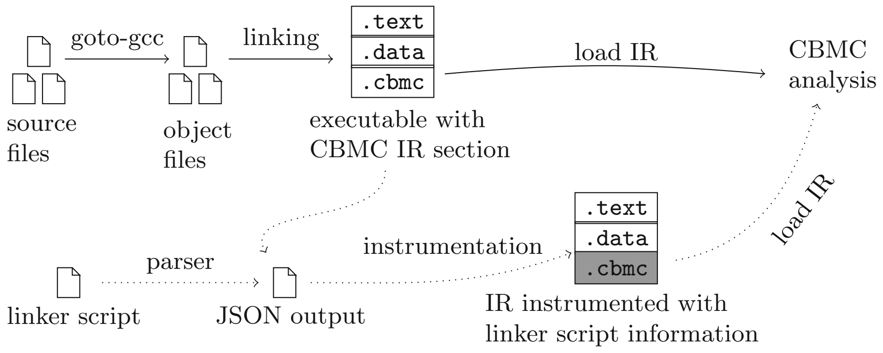

Fig. 5 How CBMC analyzes programs. Dotted lines represent the changes we made that allow CBMC to precisely reason about programs that use linker scripts

5.5 Improving analysis of programs that use linker scripts

We aim to verify the memory safety of boot code that uses a custom linker script to control its runtime memory layout. Due to the issues we described in Sect. 4.5, current static analyzers cannot correctly decide the memory safety of programs that use linker scripts. We therefore extended CBMC with a linker script parser, together with code that augments CBMC's understanding of the runtime memory layout with the information gotten from the linker script. This work can be applied to other static analyzers so that they too can more precisely reason about linker script using programs. In this section, we describe how CBMC uses the information that our parser returns.

High-level overview Figure 5 depicts how CBMC analyzes programs, together with our extensions.

To analyze a program, users first compile each of the source files with a compiler called `goto-gcc`. This compiler is a drop-in replacement for `gcc(`**1**`)`; in fact, it runs `gcc` in the background and writes the output to an object file. In addition, `goto-gcc` writes an extra section at the end of the object file that contains the source file compiled into CBMC's internal representation (IR). This means that the object files can be linked and executed as usual, but they can also be read and analyzed by CBMC. CBMC users then link the object files into an executable (which will contain the IR for the whole program); CBMC can then load the IR from the executable file and analyze it.

In a previous section, we noted that some of the information that CBMC requires—namely, the addresses of linker script defined symbols—must be read from a fully-linked executable. We can therefore take advantage of the process depicted in Fig. 5, because that process already yields an executable file. The parser that we wrote thus takes that executable file as input, reads the addresses from the file, and sends that information to CBMC.

Programs that use linker scripts do not compile correctly without that script. We therefore extended CBMC's linker so that it passes the linker script to `gcc`. We also parse the linker script to get the other information that CBMC requires: the mapping from section names to the names of the symbols that demarcate the section. Therefore, when a user now runs CBMC over an executable, it no longer immediately analyzes the loaded IR. Instead, if the codebase contained a linker script, CBMC passes the executable and linker script to the parser, and then instruments the IR with the information that the parser returns. CBMC then analyzes this augmented IR, which now contains enough information to correctly decide memory safety.

```
{
        "sections" : {
                ".data" : {
                        "start" : "data_start",
                        "end"   : "data_end"
                },
                ".rodata" : {
                        "start" : "rodata_start",
                        "size"  : "rodata_size"
                }
        },
        "addresses" : {
                "rodata_start"  : "0x20",
                "rodata_size"   : "0x100",
                "data_start"    : "0x200",
                "data_end"      : "0x380"
        }
}
```

Fig. 6 JSON output from our linker script parser

Parser output Our parser takes a linker script, together with an object file linked according to that script, as input. Given the linker script in Fig. 1 and the object file in Fig. 3, our parser would emit the JSON-formatted map in Fig. 6.

There are two top-level keys in the map. The `addresses` key contains the memory address of every symbol in the object file. If there are sections whose start address, end address, or size was demarcated by a symbol, then that information will be in the `sections` key. For every such section in the object file, the `sections` key maps the section's name to:

– the symbol whose address was set to the section's start address; and either
– the symbol whose address was set to just past the end of the section, or
– the symbol whose address was set to the section's size.

Recall that static analysis in the presence of linker scripts is imprecise because the linker script contains information that does not exist in the source code. When CBMC receives the linker parser output, it therefore transforms the program's IR by adding this missing information. There are three transformations that CBMC makes:

– transforming the type of linker defined symbols such that they can be assigned an address;
– assigning the addresses in the `addresses` map to linker defined symbols;
– declaring that the regions of memory described in the `sections` map are allocated.

We describe these transformations, illustrated in Fig. 7, in the next paragraphs.
Type transformation We transform the intermediate representation of the target program so that it contains the information needed to decide its memory safety. Static analyzers must ensure that linker defined symbols have the address that the linker assigned to them. However, it is not possible to assign a value to an address (i.e.`&data_end = 0x380`) in C, since that is the linker's job. Our solution is to change the program such that we assign `0x380` directly to the variable, rather than to the variable's address. However, since some scalar types (like `char`) cannot contain arbitrary memory addresses, we change the declaration of linker defined variables to make them pointer types, if they were not already. We also remove the `extern` qualifier, so that we can define the variables (rather than just declaring them). Finally, we also change all accesses to linker defined variables in the program, such that they correctly access the pointer value (rather than the original variable address).

```
extern char[] rodata_start;           char *rodata_start;
extern char[] rodata_size;            char *rodata_size;
extern char[] data_start;             char *data_start;
extern char[] data_end;               char *data_end;

__CPROVER_initialize()                __CPROVER_initialize()
{                                     {
                                        rodata_start = 0x20;
                                        rodata_size = 0x100;
                                        data_start = 0x200;
                                        data_end = 0x380;

                                      __CPROVER_allocated_memory(
// ...                                     data_start,
                                           data_end - data_start);

                                      __CPROVER_allocated_memory(
                                           rodata_start,
                                           rodata_size);
}                                     }

int main()                            int main()
{                                     {
  memcpy(                               memcpy(
    &data_start,                          data_start,
    &rodata_start,                        rodata_start,
    (size_t)&rodata_size);                (size_t)(rodata_size));
}                                     }
```

Fig. 7 Transforming the types of linker-defined symbols. We transform the IR of the program on the left into the one on the right. The ampersands have been removed from all accesses to the variables in the program; the types of linker defined symbols are now pointers, and no longer extern-declared; the addresses of those symbols are assigned to the pointer; and the section's extent is noted to be allocated memory

The linker manual [14] sets the convention of declaring linker defined variables to have type `char[]`. However, linker defined variables can be declared as having any type, since only their address may be accessed from the program (so the bit width of the variable's value is immaterial). Many programmers do declare linker defined variables as having type `char`, `char*`, or something else; furthermore, accesses to `foo` of type `char[]` can sometimes be preprocessed into `&foo[0]`. We give the correct pointer type to the declaration no matter what the original type was.

Address fixup and memory allocation After performing the previous step, it is possible to assign the addresses of linker defined symbols to their associated variables in the IR. We do this assignment in `__CPROVER_initialize()`, which is a function that CBMC symbolically executes just before executing `main()`. We also use that function to tell CBMC that the memory regions corresponding to object file sections are allocated, using the `__CPROVER_allocated_memory` directive. This ensures that CBMC does not report a memory safety violation when the program accesses memory in those regions.

Application to AWS boot code The linker script used in AWS boot code calls `memcpy(`**3**`)` on linker defined memory regions, in a similar manner to Fig. 1. It defines a region to hold the stage 2 binary and passes the address and size of the region as the addresses of the symbols `stage2_start` and `stage2_size`.

```
.stage2 (NOLOAD) : {
  stage2_start = .;
  . = . + STAGE2_SIZE;
```

```
        stage2_end = .;
    } > RAM2
    stage2_size = SIZEOF(.stage2);
```

The code declares the symbols as externally defined, and uses a pair of macros to convert the addresses of the symbols to an address and a constant before use.

```
    extern char stage2_start[];
    extern char stage2_size[];

    #define STAGE2_ADDRESS ((uint8_t*)(&stage2_start))
    #define STAGE2_SIZE    ((unsigned)(&stage2_size)))
```

CBMC's new approach to handling linker scripts modifies the CBMC intermediate representation of this code as described in the previous paragraphs.

Scope and limitations Our linker script parser, distributed with the CBMC source code [13], recognizes a subset of the linker command language. This subset is sufficient to enable CBMC to determine the memory safety of the AWS boot code mentioned above. The full language contains many additional constructs that were not used in the AWS linker script, but which would need to be considered by static analyses for memory safety. One example is access control: linker scripts can designate sections as being readable, writeable, or executable at runtime. It is a segmentation violation to read from or write to sections that do not have the correct permission; future work would allow static analysis tools to detect the possibility of such accesses. Linkers also place sections according to the padding constraints imposed by particular machine architectures and calling conventions, and the linker language includes constructs for fine-tuning or overriding these constraints. If the correctness of the code depends on data being padded according to explicit linker directives, then analysis tools would need to understand these directives to correctly decide memory safety.

Our work targets linkers that emit object files in the Executable and Linkable Format (ELF), specified in [53]. ELF is used on modern BSD and Linux-based operating systems; [42] is a reference work on linkers and loaders that also describes the linkers for other object formats, like COFF and PE (used on Microsoft Windows) and Mach-O (used on macOS).

The implementation of our linker script parser can take any linker script as input; it is designed to ignore any linker script construct that it does not understand. Thus, it currently extracts only the section and symbol layout information described earlier in this section. It should be straightforward to extend the parser to parse and emit information about additional linker language directives for programs whose memory safety depends on the meaning of those directives.

Summary Current static analysis tools cannot decide the memory safety of programs whose memory layout is described by a custom linker script. We extended CBMC so that it reads the memory layout from the linker script and linked executable. After doing so, CBMC adds the values of memory addresses and the extents of safe memory regions to the program's IR. This means that CBMC no longer falsely reports that accesses to linker defined memory regions are unsafe, enabling us to prove the memory safety of AWS boot code.

5.6 Running CBMC

Building the boot code and test harness for CBMC takes 8.2 s compared to building the boot code with `gcc` in 2.2 s.

Running CBMC on the test harness above as a job under AWS Batch, it finished successfully in 10:02 min. It ran on a 16-core server with 122 GiB of memory running Ubuntu 14.04, and consumed one core at 100% using 5 GiB of memory. The new encoding of arrays improved this time by 45 s.

The boot code consists of 783 lines of statically reachable code, meaning the number of lines of code in the functions that are reachable from the test harness in the function call graph. CBMC achieves complete code coverage, in the sense that every line of code CBMC fails to exercise is dead code. An example of dead code found in the boot code is the default case of a switch statement whose cases enumerate all possible values of an expression.

The boot code consists of 98 loops that fall into two classes. First are `for`-loops with constant-valued expressions for the upper and lower bounds. Second are loops of the form `while (num) ...; num--` and code inspection yields a bound on `num`. Thus, it is possible to choose loop bounds that cause all loop-unwinding assertions to hold, making CBMC's results sound for boot code.

6 Conclusion

This paper describes industrial experience with model checking production code. We extended CBMC to address issues that arise in boot code, and we proved that initial boot code running in data centers at Amazon Web Services is memory safe, a significant application of model checking in the industry. Our most significant extension to CBMC was parsing linker scripts to extract the memory layout described there for use in model checking, making CBMC the first static analysis tool to do so. With this and our other extensions to CBMC supporting devices and byte-level access, CBMC can now be used in a continuous validation flow to check for memory safety during code development. All of these extensions are in the public domain and freely available for immediate use.

References

1. Abdulla PA, Bouajjani A, Cederberg J, Haziza F, Rezine A (2008) Monotonic abstraction for programs with dynamic memory heaps. In: Gupta A, Malik S (eds) Computer aided verification. Springer, Berlin, pp 341–354
2. AFL: American fuzzy lop. http://lcamtuf.coredump.cx/afl. Accessed 6 Apr 2020
3. Albarghouthi A, Li Y, Gurfinkel A, Chechik M (2012) Ufo: a framework for abstraction- and interpolation-based software verification. In: Madhusudan P, Seshia SA (eds) Computer aided verification. Springer, Berlin, pp 672–678
4. Arbaugh WA, Farber DJ, Smith JM (1997) A secure and reliable bootstrap architecture. In: 1997 IEEE symposium on security and privacy. IEEE Computer Society, pp 65–71
5. Bazhaniuk O, Loucaides J, Rosenbaum L, Tuttle MR, Zimmer V (2015) Symbolic execution for BIOS security. In: 9th USENIX workshop on offensive technologies. USENIX Association, Washington, DC
6. Bellard F (2005) QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, pp 41–46

7. Berdine J, Calcagno C, Cook B, Distefano D, O'Hearn PW, Wies T, Yang H (2007) Shape analysis for composite data structures. In: Damm W, Hermanns H (eds) Computer aided verification. Springer, Berlin, pp 178–192
8. Beyer D, Henzinger TA, Jhala R, Majumdar R (2005) Checking memory safety with blast. In: Cerioli M (ed) Fundamental approaches to software engineering. Springer, Berlin, pp 2–18
9. Beyer D, Keremoglu ME (2011) CPAchecker: a tool for configurable software verification. In: Gopalakrishnan G, Qadeer S (eds) Computer aided verification. Springer, Berlin, pp 184–190
10. Bovet DP (2013) Special sections in Linux binaries. https://lwn.net/Articles/531148/. Accessed 6 Apr 2020
11. Burton RA (2017) rBoot: an open source boot loader for the ESP8266. https://github.com/raburton/rboot. Accessed 6 Apr 2020
12. Cadar C, Dunbar D, Engler DR (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating systems design and implementation. USENIX Association, Washington, DC, pp 209–224
13. C bounded model checker github repository. https://github.com/diffblue/cbmc. Accessed 6 Apr 2020
14. Chamberlain S (1994) Using `ld`. https://sourceware.org/binutils/docs-2.27/ld/. Accessed 6 Apr 2020
15. Chipounov V, Kuznetsov V, Candea G (2012) The S2E platform: design, implementation, and applications. ACM Trans Comput Syst 30(1):2:1–2:49
16. Cho CY, D'Silva V, Song D (2013) Blitz: compositional bounded model checking for real-world programs. In: 2013 28th IEEE/ACM international conference on automated software engineering. IEEE, pp 136–146
17. Christ J, Hoenicke J (2015) Weakly equivalent arrays. In: Lutz C, Ranise S (eds) Frontiers of combining systems. Springer, Berlin, pp 119–134
18. Cimatti A, Griggio A (2012) Software model checking via IC3. In: Madhusudan P, Seshia SA (eds) Computer aided verification. Springer, Berlin, pp 277–293
19. Clarke E, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs N, Zuck LD (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 570–574
20. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Jensen K, Podelski A (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 168–176
21. Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S (2009) VCC: a practical system for verifying concurrent C. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) Theorem proving in higher order logics. Springer, Berlin, pp 23–42
22. Condit J, Hackett B, Lahiri SK, Qadeer S (2009) Unifying type checking and property checking for low-level code. In: 36th annual ACM SIGPLANSIGACT symposium on principles of programming languages. ACM, New York, pp 302–314
23. Corin R, Manzano FA (2012) Taint analysis of security code in the KLEE symbolic execution engine. Springer, Berlin, pp 264–275
24. Synopsys static analysis (Coverity). http://coverity.com. Accessed 6 Apr 2020
25. Davidson D, Moench B, Ristenpart T, Jha S (2013) FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution. In: USENIX security symposium. USENIX, Washington, DC, pp 463–478
26. Dillig T, Dillig I, Chaudhuri S (2014) Optimal guard synthesis for memory safety. In: Biere A, Bloem R (eds) Computer aided verification. Springer, Berlin, pp 491–507
27. Dodge C, Irvine C, Nguyen T (2005) A study of initialization in Linux and OpenBSD. SIGOPS Oper Syst Rev 39(2):79–93
28. Engblom J (2016) Finding BIOS vulnerabilities with symbolic execution and virtual platforms. https://software.intel.com/en-us/blogs/2017/06/06/finding-bios-vulnerabilities-with-excite. Accessed 6 Apr 2020
29. Ferreira JF, Gherghina C, He G, Qin S, Chin WN (2014) Automated verification of the FreeRTOS scheduler in hip/sleek. Int J Softw Tools Technol Transf 16(4):381–397
30. Fortify static code analyzer. https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview. Accessed 6 Apr 2020
31. Haran A, Carter M, Emmi M, Lal A, Qadeer S, Rakamarić Z (2015) SMACK+Corral: a modular verifier. In: Baier C, Tinelli C (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 451–454
32. Harrison J, HOL Light theorem prover. http://www.cl.cam.ac.uk/~jrh13/hol-light. Accessed 6 Apr 2020
33. ISO/IEC 9899:2011(E): Information technology—Programming languages—C. Standard, International Organization for Standardization, Geneva, CH (Dec 2011)
34. Ivančić F, Yang Z, Ganai MK, Gupta A, Ashar P (2008) Efficient SAT-based bounded model checking for software verification. Theor Comput Sci 404(3):256–274

35. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2009) seL4: formal verification of an OS kernel. In: 22nd ACM SIGOPS symposium on operating system principles. ACM, New York, pp 207–220
36. Klocwork static code analyzer. https://www.klocwork.com/. Accessed 6 Apr 2020
37. Kroening D, Strichman O (2016) Decision procedures—an algorithmic point of view. Texts in theoretical computer science. An EATCS series, 2nd edn. Springer, Berlin. https://doi.org/10.1007/978-3-662-50497-0
38. Kuznetsov V, Chipounov V, Candea G (2010) Testing closed-source binary device drivers with DDT. In: USENIX annual technical conference. USENIX Association, Berkeley, CA
39. Lal A, Qadeer S (2014) Powering the static driver verifier using Corral. In: 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, pp 202–212
40. Lal A, Qadeer S, Lahiri SK (2012) A solver for reachability modulo theories. In: Madhusudan P, Seshia SA (eds) Computer aided verification. Springer, Berlin, pp 427–443
41. Lev-Ami T, Manevich R, Sagiv M (2004) TVLA: a system for generating abstract interpreters. In: Jacquart R (ed) Building the information society. Springer, Boston, pp 367–375
42. Levine JR (1999) Linkers and loaders. Morgan Kaufmann, Burlington
43. McMillan KL (2006) Lazy abstraction with interpolants. In: Ball T, Jones RB (eds) Computer aided verification. Springer, Berlin, pp 123–136
44. Moser JR (2006) Optimizing linker load times. https://lwn.net/Articles/192624/. Accessed 6 Apr 2020
45. Moser JR (2006) Prelink and address space randomization. https://lwn.net/Articles/190139/. Accessed 6 Apr 2020
46. Parvez R, Ward PAS, Ganesh V (2016) Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In: 26th annual international conference on computer science and software engineering. IBM Corp., Riverton, pp 116–127
47. Pitre N (2017) Shrinking the kernel with link-time garbage collection. https://lwn.net/Articles/741494/. Accessed 6 Apr 2020
48. Pitre N (2017) Shrinking the kernel with link-time optimization. https://lwn.net/Articles/744507/. Accessed 6 Apr 2020
49. Rakamarić Z, Emmi M (2014) Smack: decoupling source language details from verifier implementations. In: Biere A, Bloem R (eds) Computer aided verification. Springer, Berlin, pp 106–113
50. Rakamarić Z, Hu AJ (2009) A scalable memory model for low-level code. In: Jones ND, Müller-Olm M (eds) Verification, model checking, and abstract interpretation. Springer, Berlin, pp 290–304
51. Redini N, Machiry A, Das D, Fratantonio Y, Bianchi A, Gustafson E, Shoshitaishvili Y, Kruegel C, Vigna G (2017) BootStomp: on the security of bootloaders in mobile devices. In: Kirda E, Ristenpart T (eds) 26th USENIX security symposium USENIX Association Berkeley, CA, pp 781–798
52. Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: 17th annual IEEE symposium on logic in computer science. IEEE, pp 55–74
53. Santa Cruz Operation (SCO): System V Application Binary Interface (1997). www.sco.com/developers/devspecs/gabi41.pdf. Accessed 6 Apr 2020
54. Sen K (2015) Automated test generation using concolic testing. In: 8th India software engineering conference. ACM, New York, p 9
55. Sinz C, Falke S, Merz F (2010) A precise memory model for low-level bounded model checking. In: 5th international workshop on systems software verification, USENIX Association, Berkeley, CA
56. Unified extensible firmware interface forum. http://www.uefi.org/. Accessed 6 Apr 2020
57. Wang F, Shoshitaishvili Y (2017) Angr—the next generation of binary analysis. In: 2017 IEEE cybersecurity development. IEEE, pp 8–9
58. Wang W, Barrett C, Wies T (2014) Cascade 2.0. In: McMillan KL, Rival X (eds) Verification, model checking, and abstract interpretation. Springer, Berlin, pp 142–160
59. Zaddach J, Bruno L, Francillon A, Balzarotti D (2014) AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. In: 21st network and distributed system security symposium. The Internet Society