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], and Mark R. Tuttle[1]

[1] Amazon Web Services
[2] University College London
[3] University of Oxford
[4] Queen Mary University of London

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.

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,
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 [24,30,39,52] to deductive verification [21,34].

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 Section 5.7). BLITZ [16] or F-Soft [36] could be used in place of CBMC. SATABS [19], Ufo [3], Cascade [55], Blast [9], CPAchecker [10], Corral [33,43,44], and others [18,47] 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 [45] or separation logic as in [8] 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 Section 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 [7] virtual machine and KLEE. S^{2}E has been used on firmware. Parvez et al. [48] use symbolic execution to generate inputs targeting a potentially buggy statement for debugging. Kuznetsov et al. [42] used a prototype of S^{2}E to find bugs in Microsoft device drivers. Zaddach et al. [56] built the tool Avatar on top of S^{2}E 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. [6,28] used S^{2}E to search for security vulnerabilities in interrupt handlers for System Management Mode on Intel platforms. Experts can use S^{2}E on firmware. One can model device behavior (see Section 4.2) by adding a device model to QEMU or using the signaling mechanism used by S^{2}E during symbolic execution. One can declare an MMIO region (see Section 4.1) by inserting it into the QEMU memory hierarchy. Both require understanding either QEMU or S^{2}E 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. [38] 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 [54], 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) [53] 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 Section 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 [42,51], 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 [6,28]. 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 runtime 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 [35] 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 $[\mathtt{address}, \mathtt{address} + \mathtt{size})$ 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;
}
```

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.

The natural model for memory is as an array of bytes, and CBMC does the same. 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 [17,40]. 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 Section 4.2.

4.5 Linker scripts

Linking is the final stage in the process of transforming source code into an executable program. Compilation transforms source files into object files, which consist of several *sections* of related object code. A typical object file contains sections for executable code, read-only and read-write program data, debugging symbols, and other information. The linker combines several object files into a single executable object file, merging similar sections from each of the input files into single sections in the output executable. The linker combines and arranges the sections according to the directives in a *linker script*. Linker scripts are written in a declarative language [14].

The functionality of most programs is not sensitive to the exact layout of the executable file; therefore, by default, the linker uses a generic linker script[5] the directives of which are suited to laying out high-level programs. On the other hand, low-level code (like boot loaders, kernels, and firmware) must often be hard-coded to address particular memory locations, which necessitates the use of a custom linker script.

One use for a linker script is to place selected code into a specialized memory region like a *tightly-coupled memory* unit [5], which is a fast cache into which developers can place hot code. Another is device access via memory-mapped I/O as discussed in Sections 4.1 and 4.2. Low-level programs address these hard devices by having a variable whose address in memory corresponds to the address that the hardware exposes. However, no programming language offers the ability to set a variable's address from the program; the variable must instead be laid out at the right place in the object file, using linker script directives.

While linker scripts are essential to implement the functionality of low-level code, their use in higher-level programs is uncommon. Thus, we know of no work that considers the role of linker scripts in static program analysis; a recent formal treatment of linkers [37] explicitly skips linker scripts. Ensuring that static analysis results remain correct in the presence of linker scripts is vital to verifying and finding bugs in low-level code; we next describe problems that linker scripts can create for static analyses.

Linker script challenges All variables used in C programs must be *defined* exactly once. Static analyses make use of the values of these variables to decide program correctness, provided that the source code of the program and libraries used is available. However, linker scripts also define symbols that can be accessed

---

[5] On Linux and macOS, running `ld --verbose` displays the default linker script.

as variables from C source code. Since C code never defines these symbols, and linker scripts are not written in C, the values of these symbols are unknown to a static analyzer that is oblivious to linker scripts. If the correctness of code depends on the values of these symbols, it cannot be verified. To make this discussion concrete, consider the code in Figure 1.

```
/* main.c */                    /* link.ld */
#include <string.h>             SECTIONS {
                                  .text : {
extern char text_start;            text_start=.;
extern char text_size;             *(.text)
extern char scratch_start;        }
                                  text_size=SIZEOF(.text);
int main() {                      .scratch : {
  memcpy(&text_start,               scratch_start=.;
         &scratch_start,            .=.+0x1000;
         (size_t)&text_size);       scratch_end=.;
}                                 }
                                }
```

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

This example, adapted from the GNU linker manual [14], shows the common pattern of copying an entire region of program code from one part of memory to another. The linker writes an executable file in accordance with the linker script on the right; the expression "." (period) indicates the current byte offset into the executable file. The script directs the linker to generate a code section called `.text` and write the contents of the `.text` sections from each input file into that section; and to create an empty 4 KiB long section called `.scratch`. The symbols `text_start` and `scratch_start` are created at the address of the beginning of the associated section. Similarly, the symbol `text_size` is created at the address equal to the code size of the `.text` section. Since these symbols are defined in the linker script, they can be freely used from the C program on the left (which must declare the symbols as `extern`, but not define them). While the data at the symbols' locations is likely garbage, the symbols' *addresses* are meaningful; in the program, the addresses are used to copy data from one section to another.

Contemporary static analysis tools fail to correctly model the behavior of this program because they model symbols defined in C code but not in linker scripts. Tools like SeaHorn [32] and KLEE [12] do support linking of the intermediate representation (IR) compiled from each of the source files with an IR linker. By using build wrappers like wllvm [46], they can even invoke the native system linker, which itself runs the linker script on the machine code sections of the object files. The actions of the native linker, however, are not propagated back to the IR linker, so the linked IR used for static analysis contains only information

derived from C source, and not from linker scripts. As a result, these analyzers lack the required precision to prove that a safe program is safe: they generate false positives because they have no way of knowing (for example) that a `memcpy` is walking over a valid region of memory defined in the linker script.

Information required for precise modeling As we noted earlier in this section, linker scripts provide definitions to variables that may only be declared in C code, and whose addresses may be used in the program. In addition, linker scripts define the layout of code sections; the C program may copy data to and from these sections using variables defined in the linker script to demarcate valid regions inside the sections. Our aim is to allow the static analyzer to decide the memory safety of operations that use linker script definitions (if indeed they are safe, i.e., don't access memory regions outside those defined in the linker script). To do this, the analyzer must know (referencing our example in Figure 1 but without loss of generality):

1. that we are copying `&text_size` bytes starting from `&text_start`;
2. that there exists a code section (i.e., a valid region of memory) whose starting address equals `&text_start` and whose size equals `&text_size`;
3. the concrete values of that code section's size and starting address.

Fact 1 is derived from the source code; Fact 2—from parsing the linker script; and Fact 3—from disassembling the fully-linked executable, which will have had the sections and symbols laid out at their final addresses by the linker.

Extending CBMC CBMC compiles source files with a front-end that emulates the native compiler (`gcc`), but which adds an additional section to the end of the output binary [41]; this section contains the program encoded in CBMC's analysis-friendly intermediate representation (IR). In particular, CBMC's front-end takes the linker script as a command-line argument, just like `gcc`, and delegates the final link to the system's native linker. CBMC thus has access to the linker script and the final binary, which contains both native executable code and CBMC IR. We send linker script information to CBMC as follows:

1. use CBMC's front end to compile the code, producing a fully-linked binary,
2. parse the linker script and disassemble the binary to get the required data,
3. augment the IR with the definitions from the linker script and binary, and
4. analyze the augmented intermediate representation.

Our extensions are Steps 2 and 3, which we describe in more detail below. They are applicable to tools (like SeaHorn and KLEE) that use an IR linker (like `llvm-link`) before analyzing the IR.

Extracting linker script symbols Our extension to CBMC reads a linker script and extracts the information that we need. For each code section, it extracts the symbols whose addresses mark the start and end of the section, if any; and the symbol whose address indicates the section size, if any. The `sections` key of Figure 2 shows the information extracted from the linker script in Figure 1.

Extracting linker script symbol addresses To remain architecture independent, our extension uses the `objdump` program (part of the GNU Binutils [31]) to extract the addresses of all symbols in an object file (shown in the `addresses` key of Figure 2). In this way, it obtains the concrete addresses of symbols defined in the linker script.

```
"sections" : {                          }
  ".text": {                          },
    "start": "text_start",            "addresses" : {
    "size": "text_size"                 "text_start": "0x0200",
  },                                    "text_size": "0x0600",
  ".scratch" : {                        "scratch_start": "0x1000",
    "start": "scratch_start",           "scratch_end": "0x2000",
    "end": "scratch_end"              }
```

Figure 2. Output from our linker script parser when run on the linker script in Figure 1, on a binary with a 1 KiB `.text` section and 4 KiB `.scratch` section.

Augmenting the intermediate representation CBMC maintains a symbol table of all the variables used in the program. Variables that are declared `extern` in C code and never defined have no initial value in the symbol table. CBMC can still analyze code that contains undefined symbols, but as noted earlier in this section, this can lead to incorrect verification results. Our extension to CBMC extracts information described in the previous section and integrates it into the target program's IR. For example, given the source code in Figure 1, CBMC will replace it with the code given in Figure 3.

In more detail, CBMC

1. converts the types of linker symbols in the IR and symbol table to `char *`,
2. updates all expressions involving linker script symbols to be consistent with this type change,
3. creates the IR representation of C-language definitions of the linker script symbols, initializing them before the entry point of `main()`, and
4. uses the `__CPROVER_allocated_memory` API described in Section 4.1 to mark code sections demarcated by linker script symbols as allocated.

The first two steps are necessary because C will not let us set the address of a variable, but will let us store the address in a variable. CBMC thus changes the IR type of `text_start` to `char *`; sets the value of `text_start` to the address of `text_start` in the binary; and rewrites all occurrences of "`&text_start`" to "`text_start`". This preserves the original semantics while allowing CBMC to model the program. The semantics of Step 4 is impossible to express in C, justifying the use of CBMC rather than a simple source-to-source transformation.

```
#include <string.h>              #include <string.h>

extern char text_start;          char *text_start = 0x0200;
extern char text_size;           char *text_size = 0x0600;
extern char scratch_start;       char *scratch_start = 0x1000;

int main() {                     int main() {
                                   __CPROVER_allocated_memory(
                                     0x0200, 0x0600);
                                   __CPROVER_allocated_memory(
                                     0x1000, 0x1000);
  memcpy(&text_start,              memcpy(text_start,
         &scratch_start,                  scratch_start,
         (size_t)&text_size);             (size_t)text_size);
}                                }
```

Figure 3. Transformation performed by CBMC for linker-script-defined symbols.

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 Section 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 Figure 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,
  – 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.
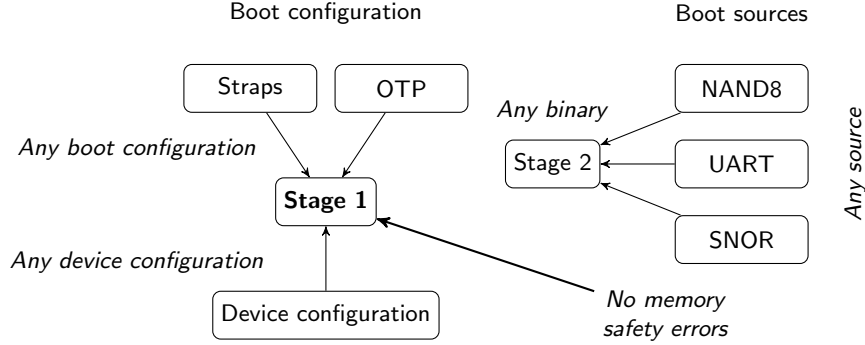
Figure 4. Boot code is free of memory safety errors.

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 (Section 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 (Section 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 (Section 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 (Section 4.4).

5.5 Linker scripts

Linker scripts allocate regions of memory and pass the addresses of these regions and other constants to the code through the symbol table. For example, the linker script 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 Section 4.5.

5.6 Test harness

The `main` procedure for the boot code begins by clearing the BSS section, copying a small amount of data from a ROM, printing some debugging information, and invoking three functions

```
SecuritySettingsOtp();
BootOptionsParse();
Stage2LoadAndExecute();
```

that read security settings from some one-time programmable memory, read the boot options from some hardware straps, and load and launch the stage 2 code.

The test harness for the boot code is 76 lines of code that looks similar to

```
void environment_model() {
  __CPROVER_allocated_memory(REG_BOOT_STRAP, 0x14);
  __CPROVER_allocated_memory(REG_UART_UART_BASE,
                             UART_REG_OFFSET_LSR +
                             sizeof(uint32_t));
  __CPROVER_allocated_memory(REG_NAND_CONFIG_REG,
                             sizeof(uint32_t));
}

void harness() {
  environment_model();
```

```
    SecuritySettingsOtp ();
    BootOptionsParse ();
    Stage2LoadAndExecute ();
}
```

The `environment_model` procedure defines the environment of the software under test not declared in the boot code itself. This environment includes more than 30 MMIO regions for hardware like some hardware straps, a UART, and some NAND memory. The fragment of the environment model reproduced above uses the `__CPROVER_allocated_memory` built-in function added to CBMC for this work to declare these MMIO regions and assign them unconstrained values (modeling unconstrained configuration information). The `harness` procedure is the test harness itself. It builds the environment model and calls the three procedures invoked by the boot code.

5.7   Running CBMC

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

Running CBMC on the test harness above as a job under AWS Batch, it finished successfully in 10:02 minutes. 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 seconds.

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, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 341–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
2. AFL: American fuzzy lop, http://lcamtuf.coredump.cx/afl
3. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 672–678. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA. pp. 65–71. IEEE Computer Society (1997), https://doi.org/10.1109/SECPRI.1997.601317
5. Arm Holdings: ARM1136JF-S and ARM1136J-S Technical Reference Manual (2006), https://developer.arm.com/docs/ddi0211/latest/
6. Bazhaniuk, O., Loucaides, J., Rosenbaum, L., Tuttle, M.R., Zimmer, V.: Symbolic execution for BIOS security. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15). USENIX Association, Washington, D.C. (2015)
7. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. pp. 41–41. ATEC '05, USENIX Association, Berkeley, CA, USA (2005)
8. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 178–192. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with Blast. In: Cerioli, M. (ed.) Fundamental Approaches to Software Engineering. pp. 2–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 184–190. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Burton, R.A.: rBoot: An open source boot loader for the ESP8266 (2017), https://github.com/raburton/rboot
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
13. C bounded model checker github repository, https://github.com/diffblue/cbmc
14. Chamberlain, S., Taylor, I.L.: The GNU linker. Red Hat, Inc. (2018), https://sourceware.org/binutils/docs/ld/
15. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. 30(1), 2:1–2:49 (Feb 2012)
16. Cho, C.Y., D'Silva, V., Song, D.: Blitz: Compositional bounded model checking for real-world programs. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 136–146 (Nov 2013)

17. Christ, J., Hoenicke, J.: Weakly equivalent arrays. In: Lutz, C., Ranise, S. (eds.) Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9322, pp. 119–134. Springer (2015), https://doi.org/10.1007/978-3-319-24246-0_8

18. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

19. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 570–574. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

20. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004), https://doi.org/10.1007/978-3-540-24730-2_15

21. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

22. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 302–314. POPL '09, ACM, New York, NY, USA (2009)

23. Corin, R., Manzano, F.A.: Taint Analysis of Security Code in the KLEE Symbolic Execution Engine, pp. 264–275. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

24. Synopsys static analysis (Coverity), http://coverity.com

25. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 463–478. USENIX, Washington, D.C. (2013)

26. Dillig, T., Dillig, I., Chaudhuri, S.: Optimal guard synthesis for memory safety. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 491–507. Springer International Publishing (2014)

27. Dodge, C., Irvine, C., Nguyen, T.: A study of initialization in Linux and OpenBSD. SIGOPS Oper. Syst. Rev. 39(2), 79–93 (Apr 2005), http://doi.acm.org/10.1145/1055218.1055226

28. Engblom, J.: Finding BIOS vulnerabilities with symbolic execution and virtual platforms (Jun 2016), https://software.intel.com/en-us/blogs/2017/06/06/finding-bios-vulnerabilities-with-excite

29. Ferreira, J.F., Gherghina, C., He, G., Qin, S., Chin, W.N.: Automated verification of the FreeRTOS scheduler in Hip/Sleek. International Journal on Software Tools for Technology Transfer 16(4), 381–397 (Aug 2014)

30. Fortify static code analyzer, https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview

31. Free Software Foundation: Documentation for binutils 2.29 (2017), https://sourceware.org/binutils/docs-2.29/

32. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015). Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015)

33. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: A modular verifier. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 451–454. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

34. Harrison, J.: HOL Light theorem prover, http://www.cl.cam.ac.uk/~jrh13/hol-light

35. ISO/IEC 9899:2011(E): Information technology - Programming languages - C. Standard, International Organization for Standardization, Geneva, CH (Dec 2011)

36. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404(3), 256–274 (Sep 2008)

37. Kell, S., Mulligan, D.P., Sewell, P.: The missing link: explaining ELF static linking, semantically. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 607–623. ACM (2016)

38. 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.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, ACM, New York, NY, USA (2009)

39. Klocwork static code analyzer, https://www.klocwork.com/

40. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016), https://doi.org/10.1007/978-3-662-50497-0

41. Kroening, D., Tautschnig, M.: Automating software analysis at large scale. In: Hlinený, P., Dvorak, Z., Jaros, J., Kofron, J., Korenek, J., Matula, P., Pala, K. (eds.) Mathematical and Engineering Methods in Computer Science - 9th International Doctoral Workshop, MEMICS 2014, Telč, Czech Republic, October 17-19, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8934, pp. 30–39. Springer (2014), https://doi.org/10.1007/978-3-319-14896-0_3

42. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with DDT. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. pp. 12–12. USENIXATC'10, USENIX Association, Berkeley, CA, USA (2010)

43. Lal, A., Qadeer, S.: Powering the Static Driver Verifier using Corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 202–212. FSE 2014, ACM, New York, NY, USA (2014)

44. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 427–443. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012)

45. Lev-Ami, T., Manevich, R., Sagiv, M.: TVLA: A system for generating abstract interpreters. In: Jacquart, R. (ed.) Building the Information Society. pp. 367–375. Springer US, Boston, MA (2004)

46. Mason, I.A.: Whole program LLVM (2017), https://github.com/SRI-CSL/whole-program-llvm/tree/master/wllvm

47. McMillan, K.L.: Lazy abstraction with interpolants. In: Proceedings of the 18th International Conference on Computer Aided Verification. pp. 123–136. CAV'06, Springer-Verlag, Berlin, Heidelberg (2006)

48. Parvez, R., Ward, P.A.S., Ganesh, V.: Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering. pp. 116–127. CASCON '16, IBM Corp., Riverton, NJ, USA (2016)

49. Rakamarić, Z., Emmi, M.: Smack: Decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 106–113. Springer International Publishing (2014)

50. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: Jones, N.D., Müller-Olm, M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 290–304. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

51. Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: BootStomp: On the security of boot-loaders in mobile devices. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 781–798. USENIX Association (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini

52. Sen, K.: Automated test generation using concolic testing. In: Proceedings of the 8th India Software Engineering Conference. pp. 9–9. ISEC '15, ACM, New York, NY, USA (2015)

53. Unified extensible firmware interface forum, http://www.uefi.org/

54. Wang, F., Shoshitaishvili, Y.: Angr – the next generation of binary analysis. In: 2017 IEEE Cybersecurity Development (SecDev). pp. 8–9 (Sept 2017)

55. Wang, W., Barrett, C., Wies, T.: Cascade 2.0. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 142–160. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

56. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In: 21st Network and Distributed System Security Symposium (NDSS) (Feb 2014)