Verifying a Bootloader

Axel Marmet

January 2021

1 Introduction

When building formally verified stacks many layers of abstraction must be taken into account. It is not enough to verify the software if the operating system it will certainly call is not itself verified. In exactly the same way to completely trust the OS we must verify that is correctly setup by the bootloader. This project is part of an effort to reduce the TCB of a formally verified stack for network functions[Zao+19]. The part of the stack that is verified is not the bootloader itself as GRUB is used, but the 86 assembly instructions that stand between the entry point of the executable given to GRUB and the call to the main function of the operating system.

2 How to verify

To verify that a program of a language does what it should we can use Operational Semantics. The book Formal Reasoning about Programs defines operational semantics as "a family of techniques for automatically defining a transition system, or other relational characterisation, from program syntax.". Using operational semantics we can take in an initial configuration and automatically use the transition system to evaluate the problem. This can be seen as an abstract interpreter. To do so we use a tool called the k-framework.

3 What is the K framework

The K framework is a rewrite-based executable semantic framework[RŞ10]. Many languages have been formalised in K. For example Java[BR15], Javascript[PŞR15] or C[Ell12]. We present a simple example that can be found in the tutorial. listing 1 shows the syntax of an extremely simple arithmetic language.

Listing 1: Syntax of simple imperative language

```
1  module IMP-SYNTAX
2    imports DOMAINS-SYNTAX
3    syntax AExp  ::= Int | Id
4                   | "-" Int
5                   | AExp "/" AExp              [left, strict]
6                   > AExp "+" AExp              [left, strict]
7                   | "(" AExp ")"               [bracket]
8    syntax BExp  ::= Bool
9                   | AExp "<=" AExp             [seqstrict]
10                  | "!" BExp                   [strict]
11                  > BExp "&&" BExp             [left, strict(1)]
12                  | "(" BExp ")"               [bracket]
13   syntax Block ::= "{" "}"
14                  | "{" Stmt "}"
15   syntax Stmt  ::= Block
16                  | Id "=" AExp ";"            [strict(2)]
17                  | "if" "(" BExp ")"
18                    Block "else" Block         [strict(1)]
19                  | "while" "(" BExp ")" Block
20                  > Stmt Stmt                  [left]
21   syntax Pgm ::= "int" Ids ";" Stmt
```

```
22    syntax Ids ::= List{Id,","}
23  endmodule
```

The syntax is given in Backus-Naur form (BNF) with some additional informations. For example left specifies to the parser that the term is left associative and strict (i) specifies that the *ith* non terminal term must be evaluated before the expression itself. For example line 17 and 18 show that when evaluating an if statement the condition must be evaluated first.
The semantics shown in listing 2 are implemented by rewriting a configuration. A configuration is simply a set of cells that can be nested. In this simple example there are only two cells:

1. the cell K that holds the program that is being executed
2. the cell *state* that holds a map from variable names to integer

As shown on line 12 there is a unifying algorithm that allows to rewrite cells based on the content of others. Line 12 can be read as "if we are currently evaluating an identifier X and there exists a term I pointed to by X in our environment then we can rewrite X to I in our program".

Listing 2: Semantics of simple imperative language

```
1   module IMP
2     imports IMP-SYNTAX
3     imports DOMAINS
4     syntax KResult ::= Int | Bool
5
6     configuration <T color="yellow">
7                     <k color="green"> $PGM:Pgm </k>
8                     <state color="red"> .Map </state>
9                   </T>
10
11  // AExp
12    rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
13    rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
14    rule I1 + I2 => I1 +Int I2
15    rule - I1 => 0 -Int I1
16  // BExp
17    rule I1 <= I2 => I1 <=Int I2
18    rule ! T => notBool T
19    rule true && B => B
20    rule false && _ => false
21  // Block
22    rule {} => .    [structural]
23    rule {S} => S   [structural]
24  // Stmt
25    rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
26    rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
27    rule if (true)  S else _ => S
28    rule if (false) _ else S => S
29    rule while (B) S => if (B) {S while (B) S} else {}  [structural]
30  // Pgm
31    rule <k> int (X,Xs => Xs); _ </k> <state> Rho:Map (.Map => X|->0) </state>
32      requires notBool (X in keys(Rho))
33    rule int .Ids; S => S  [structural]
34
35  // verification ids
36    syntax Id ::= "n"     [token]
37                | "sum"   [token]
38  endmodule
```

4 What is verified

We prove that after the assembly instructions are executed :

1. the CPU supports various extensions and the instructions to control them.
2. 64-bit mode is enabled and the code segment is a 64-bit read/execute segment with Descriptor Privilege Level = 0
3. Paging is enabled and the first 4 GiB of memory are identity-mapped with RWX permissions

Because GRUB executes first we also have some assumptions that it works correctly. The most important ones are that paging is disabled, the CPU is in protected mode and we are in 32-bit mode.

5 Previous work

The semantics of user space x86 were already available[Das+19]. Such program as shown in listing 3 could be compiled into listing 4 and the generated asm file could be given to the K framework to be semantically executed.

Listing 3: Simple sum to N program in C

```
1 int sumToN(int N) {
2   int s = 0;
3   int n = N;
4   while (n > 0) {
5     s = s + n;
6     n = n - 1;
7   }
8   return s;
9 }
```

Listing 4: Compiled version of listing 3

```
1  sumToN:
2    pushq %rbp
3    movq %rsp, %rbp
4    movl %edi, -20(%rbp)
5    movl $0, -4(%rbp)
6    movl -20(%rbp), %eax
7    movl %eax, -8(%rbp)
8  L3:
9    cmpl $0, -8(%rbp)
10   jle L2
11   movl -8(%rbp), %eax
12   addl %eax, -4(%rbp)
13   decl -8(%rbp)
14   jmp L3
15 L2:
16   movl -4(%rbp), %eax
17   popq %rbp
18   ret
```

The semantics were in part automatically generated, which, as is explained later, can be problematic when instructions have to be modified.

6 Correctly handling instruction in memory

In the initial semantics the code did not exist in memory but in an abstract data structure where all the instructions were of size 1 and the RIP referenced indices in this data structure. This abstraction makes things easier without simplifying things too much when assuming user space x86. This is due to the fact that when an OS loads an executable all the different segments (text, data, readonly data, etc) are relocated and protected from one another by paging and so on. However when we are setting up paging we cannot afford this luxury.
As such one of the first steps was to insert the code section in the memory. To do so correctly it was necessary to know the size of each opcodes, which is not easy to do in a CISC ISA like x86.
The solution to this is not to write the entire semantics of an x86 assembler but simply to give this information in the original configuration given to the semantics. To do so we compile the our executable and then disassemble it using *objdump*.

6.1 Mixed mode assembly

Because we are switching from 32 bits mode to 64 bits mode in the code it is a necessity to have 32 bits instructions and then later 64 bits instructions. This is problematic for both objdump and gdb as they use the information in

the ELF file header, which says that the file format is *elf64-x86-64* which leads the 32 bits section to be disassembled incorrectly.
For example the bytes "a3 00 30 1a 00 b8 00 30 1a 00 0f 22 d8" get incorrectly translated to

```
1 movabs %eax,0x1a3000b8001a3000;
2 add    %cl,(%rdi);
3 and    %al,%bl;
```

It should be disassembled as

```
1 mov    %eax,0x1a3000
2 mov    $0x1a3000,%eax
3 mov    %eax,%cr3
```

The solution found is to use *objcopy* to extract the 32 bits section of text and translate it to *elf32-i386*. Then it can be disassembled correctly.

6.2 Custom Loader Script

Because we do not want to disassemble the entire executable given to GRUB but only the startup code, we use a custom loader script to create multiple text, rodata and data sections depending if the sections come from the asm file or others. The relevant part of the linker are as follows

```
1       .text.start.32 ALIGN(4K) :
2       {
3           *boot.o( .text.32 )
4       }
5       .text.start.64 ALIGN(4K) :
6       {
7           *boot.o( .text.64 )
8       }
9       ...
10      .data.start ALIGN(4K) :
11      {
12          *boot.o(.data)
13      }
14      .rodata.start ALIGN(4K) :
15      {
16          *boot.o(.rodata)
17      }
```

This allows to isolate the sections coming from the assembly program.

6.3 Ways to load instructions

The first idea to load instruction was recursive in nature as this translates very well in the K framework. When parsing the given disassembled file we would create a map from the instruction's machine code to the instruction for example:

```
1   24'983504 |-> execinstr ( xgetbv .Operands )
2   24'983505 |-> execinstr ( xsetbv .Operands )
```

The syntax $x'y$ represent a bitvector of width x and signed integer value y in base 10. The fetch stage would start by loading 1 byte at the address specified by the program counter, if this byte was not a key in the decoding map then 2 bytes were loaded. In the general case if i bytes did not appear in the decoding map then $i+1$ bytes were then loaded. This was quite intuitive but also quite slow. The final method is to store a second map along the decoding map. This second map is a partial function from addresses to the size of the instructions. That way the fetch stage can directly ask the memory for the correct number of bytes. As an example if we had in the file

```
1 0x101000:     0xbc00302e00    movl    $0x2e3000,%esp
2 0x101005:     0x9c            pushfl
```

Then after parsing the decoding map would contain

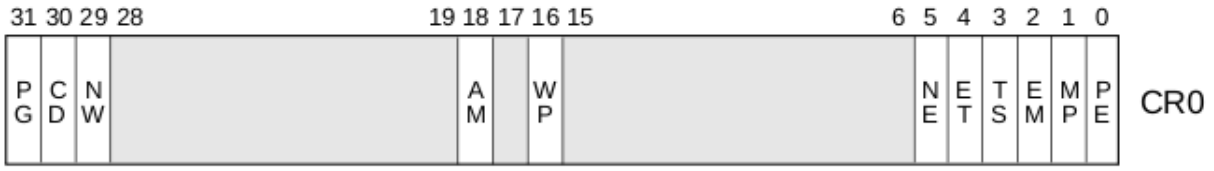

Figure 1: Description of *CR0* from the manual

```
40'807457009152 |-> execinstr ( movl $ 0x2e3000 , %esp , .Operands )
8'156 |-> execinstr ( pushfl .Operands )
```

and the size map

```
52'1052672 |-> 40
52'1052677 |-> 8
```

7 System registers

Because the initial semantics were concerned with the user space there were no control registers, segment registers or model specific registers. As such the control registers *CR[0..4]*, *CR8* and *XCR0*, all the segment registers, the model specific register *IA32_EFER* and the *FLAGS* register were added.
For example the control register CR0 is described in the Intel manual by the fig. 1 Which is reflected in the configuration by the following flags

```
("PG"      |-> mi(1, 0))    // Paging
("CD"      |-> mi(1, 0))    // Cache Disable
("NW"      |-> mi(1, 0))    // Not Write-through
("AM"      |-> mi(1, 0))    // Alignment Mask
("WP"      |-> mi(1, 0))    // Write Protect
("NE"      |-> mi(1, 0))    // Numeric Error
("ET"      |-> mi(1, 1))    // Extension Type
("TS"      |-> mi(1, 0))    // Task switched
("EM"      |-> mi(1, 0))    // Emulation
("MP"      |-> mi(1, 0))    // Monitor Coprocessor
("PE"      |-> mi(1, 1))    // Protection Enable
```

as it is easier to internally represent most of the system registers as set of flags and then reconstitute them in a bitvector if necessary.

8 System instructions

The semantics for the following system instructions were added:

1. *lgdt* : Load Global Descriptor Table, used to load the segment descriptor table. Necessary for 64 bits mode.
2. *ljmp* : Long Jump, used to jump not only to a new rip address but also a new code segment.
3. *mov* : Moving to and from control register, require current privilege level to be 0.
4. *mov* : Moving to segment register, necessary to load null into unused segments.
5. *xgetbv* / *xsetbv* : Get/Set Value of Extended Control Register, necessary to enable extensions like SSE
6. *cpuid* : CPU Identification, used to obtain information about the processor model and its supported features
7. *pushf* : Push EFLAGS Register onto the Stack
8. *popf* : Pop Stack into EFLAGS Register
9. *rdmsr* / *wrmsr*: Read from / write to Model Specific Register, necessary to access *IA32_EFER* to set the 64 bits mode.

For the sake of conciseness we will not explain how all the instructions and registers are linked but for example: To setup the 64 bit mode we must write to the MSR *IA32_EFER*, to check if this MSR is supported we have to check leaf $0x80000001$ with *cpuid* but not all processors support leaf $0x80000001$ so leaf $0x80000000$ must be checked first to verify the maximum leaf supported. But not all processors support *cpuid* and to verify if it is supported the software must verify that the bit 21 in *EFLAGS* is writable by using *pushf* and *popf*.
All those links are present in the semantics and the supported features can easily be turned on or off in the configuration.

9 Segments

As explained earlier a considerable part of the instruction semantics were created automatically. This can be a problem because if it became necessary to modify how the instructions behaved the process could be quite long and error prone. This would have been necessary if all the mechanisms of segmentation had been used. Because there are different segments for data and stack all instructions would have to be checked and the memory access changed depending on if the stack was used or not. Luckily in our assumptions we assume that GRUB configures all the segments to have the same offset of 0 and a limit of $0xffffffff$. And in 64 bits mode segmentation is mostly disabled and only some flags in the code segment are used. As such the current implementation of segment where it is assumed that all segments have the same offset and limit is not a perfect representation of the semantics of segmentation in general but it works perfectly for the program verified in this project.

10 Paging

As opposed to segmentation all virtual addresses are translated in the same manner. That is to say that where the address comes from does not matter. This means that the paging semantics could be entirely implemented. Which is important as one of the guarantees offered by the program is that the first 4GiB of memory are identity mapped. In the interest of time only the paging mode used in the program, IA32-e paging with 1GiB pages, was implemented. But other modes of paging are not more complex and could be quickly implement if it were necessary.

11 Results

The program can be semantically executed and the final configuration can be checked to fulfil all guarantees. Furthermore it can be shown that the set of assumptions is minimal as if any assumptions is missing the main function of NFOS is not called. Semantic unit tests are written where a simple configuration meant to test a single instruction is ran and the resulting configuration is checked automatically to contain the expected result.

12 Future Work

The K framework provides a language-parametric, reachability logic theorem prover[Ste+16]. This hasn't been investigated yet.
Furthermore verification of the semantics by using a widely adopted emulator like *QEMU* has been realised by hand by the author, but a lack of time and the added complexity of mixing 32 and 64 bits instructions combined to the necessity of running GRUB beforehand to adequately setup the processor meant that the checks were not automated.
Finally the output of *objdump* has to be slightly adjusted to prefix hexadecimal numbers with "0x" as otherwise parsing the input file becomes context dependent which is not supported by the K framework. For example "add" can sometimes be the opcode add and sometimes the number 2781 in base 10. However a parser written in another language can be hooked to the input so that a context dependent syntax can be supported.

13 How to use

The project is hosted in a Github repository[1]. It is composed of two submodules, a fork of Vigor and a fork of the x86 semantics. The README lying in the Vigor submodule is still valid and explains how to install it. The instructions on how to install the K framework in the semantics README are also valid. There is no need to compile the semantics directly, scripts have abstracted this away. The script *extract_dis.sh* takes in as input an executable and outputs a disassembled file where the 32 and 64 bits section have been correctly extracted and concatenated. The executable

[1] `https://github.com/axelmarmet/FV_boot`

should be obtained by running make nfos—all in one of the network function directories of Vigor. The README explains how to manually adjust the output file due to the lack of a context dependant parser. Then the *run.sh* takes in as an input one of the configuration files held in the directory *configurations* and the disassembled file to execute. The whole workflow is explained in the main README.

14 Conclusion

In this project two areas of computer science were united. On one hand the computer engineering side of understanding a bootloader and the x86 ISA and on the other hand the concept of operational semantics from the field of formal verification. The goal of the project has been reached. Future work is however still possible.

References

[RȘ10] Grigore Roșu and Traian Florin Șerbănuță. "An overview of the K semantic framework". In: *The Journal of Logic and Algebraic Programming* 79.6 (2010). Membrane computing and programming, pp. 397–434. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2010.03.012. URL: http://www.sciencedirect.com/science/article/pii/S1567832610000160.

[Ell12] Chucky Ellison. "A Formal Semantics of C with Applications". PhD thesis. University of Illinois, July 2012. DOI: http://hdl.handle.net/2142/34297.

[BR15] Denis Bogdănaş and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, Jan. 2015, pp. 445–456. DOI: http://dx.doi.org/10.1145/2676726.2676982.

[PŞR15] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. "KJS: A Complete Formal Semantics of JavaScript". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, June 2015, pp. 346–356. DOI: http://dx.doi.org/10.1145/2737924.2737991.

[Ste+16] Andrei Stefănescu et al. "Semantics-Based Program Verifiers for All Languages". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 74–91. ISSN: 0362-1340. DOI: 10.1145/3022671.2984027. URL: https://doi.org/10.1145/3022671.2984027.

[Das+19] Sandeep Dasgupta et al. "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, June 2019, pp. 1133–1148. DOI: http://dx.doi.org/10.1145/3314221.3314601.

[Zao+19] Arseniy Zaostrovnykh et al. "Verifying Software Network Functions with No Verification Expertise". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 275–290. ISBN: 9781450368735. DOI: 10.1145/3341301.3359647. URL: https://doi.org/10.1145/3341301.3359647.