Towards Formalizing the x86-64 Instruction Decoder in $\mathbb{K}$

Andrew H. Miranti **Sandeep Dasgupta** **Grigore Roşu**
University of Illinois at Urbana Champaign, USA
{miranti2, sdasgup3, grosu}@illinois.edu

Abstract

The x86-64 instruction set architecture being one of the most complex and widely used ISAs, ensuring the correctness of the x86-64 binary code is important. One of the strongest ways to ensure that is to define a formal semantics of the x86-64 language. We present ongoing work in augmenting the capability of an existing x86-64 formal semantics, with the most complete user-level instruction support, by formalizing the instruction decoder and thereby allowing direct analysis of binary code.

2012 ACM Subject Classification Software and its engineering → Formal language definitions

Keywords and phrases x86-64, ISA specification, Formal Semantics

1 Introduction

x86-64 being one of the most complex ISAs and widely used in servers and desktop, ensuring the correctness of the program written in binary is extremely important. A formal semantics of x86-64 is imperative for formal reasoning about binary code, which is one of the strongest ways to ensure correctness. A formal semantics of the ISA allows direct reasoning about the binary code, which is desirable, not only because it allows to analyze the binary even when the source code is not available (e.g., legacy code or malware), but also because it avoids the need to trust the correctness of compilers.

Completely formalizing the semantics of the x86-64 ISA, however, is extremely challenging especially due to the complexity and the large number of instructions that are informally specified in the reference manual [4]. Despite the challenges, we witnessed the heroic effort [8] in formalizing the semantics of most of the user-level instructions using the language semantics engineering framework $\mathbb{K}$. However, the developed semantics is for the assembly language notation of the binary program and hence limited in the sense that any binary analysis using that formalism requires, as a prerequisite, converting the binary to the supported mnemonic notation using an off-the-shelf disassembler. Such a requirement includes the disassembler in the trusted computing base. Through the present work, we eliminate the limitation by formally specifying an instruction decoder and thereby allowing [8] to work directly on binary code. Also, our work is publicly available [7].

2 Instruction Decoding

While a semantics of x86 mnemonics is valuable on its own, its direct uses are limited alone. Very few real world programs are most conveniently available in assembler form, and not source or binary forms. Formalizations for many source code languages already exist, and thus would be a natural choice in the event that source code is available. However, many programs must make use of precompiled binaries for which source is not available – either due to IP concerns, the circumstances of the researcher (as an example, malware researchers) or due to the age of the program. In order to reason about these programs, one must work from what one has: an executable binary file. Thus, in order to apply the existing semantics of x86 to more common real world programs, the semantics must incorporate a disassembler of some form to translate from binary back to the modeled assembly language.

2.1 Approach

Two basic approaches to applying binaries to the existing x86 semantics present themselves. First, one could disassemble the entire binary into a set of x86 assembler files, and then pass those directly to run on the semantics unmodified. Alternatively, one could formalize the semantics of instruction decoding, and perform the decoding at runtime with the semantics. The first option would seem relatively simpler, as it would allow the use of existing tools to perform the decoding. However, disassembly of x86 programs is no trivial task. While there exist many tools capable of disassembling a binary, they are forced to make certain assumptions about the programs under disassembly. The basic algorithms are as follows:

1. Linear sweep (as implemented in objdump [5]): Starting from the beginning of the text segment, disassemble instructions sequentially over the entire program. This approach is the simplest, but breaks if it starts at an incorrect offset, or unexpected data appears in the text segment of the program.
2. Recursive descent (as implemented in IDA [2]): Starting at a known address (e.g. the program entry), decode sequentially until a branch is detected, in which case spawn another thread which decodes the target.
3. Probabilistic (Millet *et al.* [9]): Use semantic features (for example, register use-def chains and control transfer targets), that only a real code body would likely demonstrate, as hints to discover correlations between code bytes (or true instructions) and use those hints to determine whether a given byte is likely part of an instruction and, if so, at what offset that instruction is most likely to begin.

However, each of these algorithms share a common flaw: they cannot offer any formal certainty that the decoded program is actually the one that would run if the binary were executed, and thus any property proved from that program must be met with some skepticism. As an example of why, consider the target of an indirect jump (of the form `jmp %rax`) – where the target is computed by an arbitrary function. Determining the value of that function's return can be arbitrarily complex, up to undecidable. Thus, the location of program flow is, in the general case, undecidable. Even if no deliberate example of an indirect jump to a computed location exists, one can still occur through a programming error such as a buffer overflow. Few, if any, real world x86 programs lack any indirect jumps, as `ret` performs one. Thus, any attempt to prove a property over a disassembled binary must solve this problem. Hence, the need for a formalized decoder.

In order to sidestep these issues, we took the alternative approach of decoding individual instructions as the semantics execute them. At any given step of execution, the semantics know exactly where the program counter is, and thus exactly where the next instruction should start. After decoding and executing the instruction, the semantics know exactly where the program counter should go next (from the size and semantics of the instruction decoded). This however requires a formalization of instruction decoding in $\mathbb{K}$, in order to execute the decoder in tandem with the semantics. To avoid reinventing the wheel, we chose to base the formalized decoder implementation on an existing implementation - Intel's x86 instruction decoding and disassembly tool, XED [6]. We ported their decoding algorithm to $\mathbb{K}$, and modified it to better fit the application and environment.

An x86 instruction can be broken down into $0-4$ prefixes, the opcode byte(s), the MODRM byte, the SIB byte, the displacement and the immediate(s). Of these sections, all but the opcodes are optional. One cannot know how many of these sections exist, or how to interpret the values contained in them until they have decoded previous sections.

So the first step of the algorithm is to look for prefixes, especially VEX and EVEX prefixes, each of which will cause future opcode values to be interpreted differently. The presence of other prefixes is also recorded, and using this information the opcode is found. Using the information about opcode and prefixes, the presence or absence of the MODRM and SIB bytes are determined, and these bytes are picked apart for their constituent values.

After all this data has been extracted, the precise instruction and operand variant can be determined. This is where the $\mathbb{K}$ Framework in particular shines – matching arbitrary subsets of properties to particular patterns is the foundation of both the $\mathbb{K}$ rule, and this pivotal instruction selection step. This match gives this step an extremely intuitive representation in $\mathbb{K}$ – one rule per instruction variant. Moreover, the input data fed to the automatically generated sections of the original Intel decoder can be re-purposed to automatically generate these $\mathbb{K}$ rules. The reference decoder requires complex generated code to perform what amounts to pattern matching on various decode properties that can be known without the instruction variant (as an example, branching between two possible sets of variants on whether the MOD bits of MODRM were 0b11). $\mathbb{K}$ can do all of these pattern matches in one step, and in a far more legible format by simply incorporating the desired constraint into a rule.

After instruction selection, the sizes and positions of the remaining sections of the instructions are known to the decoder, and it can extract them trivially. After this point, the next challenge is using the decoded instruction to generate the desired semantics operation. Unfortunately AT&T syntax (which the most complete x86-64 semantics [8], modeled in $\mathbb{K}$, were based on) lacks clear standardization over all of x86, and its implementation by GAS [1] differs from its implementation by XED (for example, in how it produces suffixes for certain instructions), along with changes in instruction mnemonics between input assembler code and output disassembled code makes mapping decoded instruction mnemonics (decoder output) to assembly instruction mnemonics (semantics input) nontrivial. This problem was eventually solved by building a lookup table of assembled instruction variants to original assembler mnemonics using GAS. Once this step was completed, translating the instruction operands was relatively simple.

2.2 Evaluation

At first, the decoder was tested on its ability to convert any binary sequence, specifying to a valid instruction in x86-64, to the corresponding mnemonic. The binary sequences are obtained either from XED test-suite or created manually using GAS [1] assembler. A successful completion of this experiment ensured that the decoding logic is correctly implemented and works for all the valid x86-64 instructions. To gain more confidence, the decoder was also tested by combining the instruction disassembler with a simple linear sweep algorithm and then comparing it with the output of XED. Once an acceptable accuracy was reached, the decoder was combined with the x86 semantics and run on a selection of the gcc-c torture tests [3]. The tests were modified slightly from their original implementations by replacing certain standard library implementations with simpler versions. This was done for the sake of feasibility, as the semantics run much more slowly than native code. As a ground truth, a similarly modifed version of the gcc torture tests were executed normally, and any that did not pass (mostly, due to segmentation faults) running natively due to the simplified stdlib implementations were removed from the selection. This is a limitation of the semantics, due to the lack of a simulated OS for a reasonable implementation, and a need to optimize memory performance, we have omitted a concept of memory page ownership, and thus simulated programs at present will not fault. The gcc torture tests seem generally

to be structured such that the test case will either crash or call abort in the event of an error. This permitted us to avoid needing to compare process memory images to semantics memory images (though such a comparison would doubtlessly be useful - we leave it to future work). A test was considered to run successfully if the semantics produced an $exit_0$ symbol on completion. The initial selection contained 498 tests. Of the 482 of the selected torture tests that executed successfully natively, 468 (97.1%) executed successfully under the semantics. Note that the tests were run with a timeout, and many of the failures were the test running out of time (and this number of failures was very slightly nondeterministic due to background tasks on the machine). The failures were a mix of programs that used unsupported instructions, programs which took too long and timed out, programs that called abort and programs that crashed or aborted during execution. Though not all passed, the vast majority of the tests did (and, doubtlessly a longer timeout would have allowed several of the failure cases to pass), and continuing work will see this number increase.

2.3 Limitation

The new decode + execution semantics is much more practically useful than the original semantics, but still has some limitations. Dynamic linking is not supported, so the binary must be statically linked (or the dynamic sections made unreachable). Presently the semantics do not support system calls so, although the decoder can decode them, they cannot be executed properly. This leads to a need for some debugging symbols, as the semantics must skip to the *<main>* symbol rather than starting at the program entry point, since most libc implementations' initialization code will always make system calls. Future work will be addressing these shortcomings, and addressing technical incompatibilities that currently block symbolic execution over the semantics.

References

1 GNU Assembler. `https://en.wikipedia.org/wiki/GNU_Assembler`. Last accessed: 9th July 2019.

2 Hex-Rays, The Interactive Disassembler.

3 C Language Testsuites: C-torture version 8.1.0. `https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`, 2018. Last accessed: 9th July 2019.

4 Intel 64 and IA-32 Architectures Software Developer Manuals. `https://software.intel.com/en-us/articles/intel-sdm`, 2018. Published on October 12, 2016, updated May 18, 2018.

5 objdump(1) - Linux man page. `https://linux.die.net/man/1/objdump`, July 2018. Last accessed: 9th July 2019.

6 Mark Charney. Intel® X86 Encoder Decoder Software Library. `https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library`, 2015. Last accessed: 9th July 2019.

7 Sandeep Dasgupta. Semantics of x86-64 in K. `https://github.com/kframework/X86-64-semantics/tree/new_memory_model`, 2018. Last accessed: 9th July 2019.

8 Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 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 2019, pages 1133–1148, New York, NY, USA, 2019. ACM. URL: `http://doi.acm.org/10.1145/3314221.3314601`, `doi:10.1145/3314221.3314601`.

9 Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1187–1198, Piscataway, NJ, USA, 2019. IEEE Press. URL: `https://doi.org/10.1109/ICSE.2019.00121`, `doi:10.1109/ICSE.2019.00121`.