Sem86: A Full-System Emulator Without Hard-Coded Semantics

Jos Craaijo[1][0009−0001−9799−3517], Freek Verbeek[1,2][0000−0002−6625−1123], and Binoy Ravindran[2][0000−0002−8663−739X]

[1] Open Universiteit, Valkenburgerweg 177, 6419 AT Heerlen
{jos.craaijo,freek.verbeek}@ou.nl
[2] Virginia Tech, Blacksburg, VA 24061, United States
{freek,binoy}@vt.edu

Abstract. Emulation can be used to run legacy software, analyze malware in a sandboxed environment, or run software compiled for different architectures. An emulator is based on instruction semantics. There is, however, not a single instruction semantics to be followed, as x86 allows undefined behavior. In order to make accurate CPU-specific emulators, we argue the need for an emulator that can easily switch between different semantics. However, all existing emulators contain hard-coded semantics. We present Sem86, an x86 emulator that loads semantics at runtime from data in an input file. We implement all necessary hardware needed to emulate typical x86 operating systems such as Windows 98, Windows XP and Windows 7. Additionally, we implement a sound card, network card and support for high-resolution display output. Sem86 can automatically *bisect* instruction execution to determine at which point different semantics would diverge. We demonstrate this by constructing a toy malware example that exploits undefined instruction behavior to detect whether it is running in an emulator, and show that Sem86 can pinpoint the exact instruction that is used.

Keywords: x86 semantics · Full-system emulators · Malware analysis

1 Introduction

Malware often attempts to frustrate analysis by disabling itself when it detects that it is running inside an emulator [7, 8, 21–23]. Some emulator detection techniques make use of *undefined* instruction behavior. These outputs are unspecified in CPU reference manuals, and are often implemented differently on different CPU architectures and emulators.

Existing emulators [4, 5, 15] do not accurately implement undefined behavior [17,18]. Most existing formal semantics [1,9,11,14] simply mark such outputs as "undefined", and do not provide specific implementations. This leads to differences in behavior, which malware exploits to detect whether it is running inside an emulator. Such differences are actively used by real-world malware. For example, malware was encountered that uses the output of undefined flags from the `IMUL` instruction to detect emulation [23].

Since implementations of undefined behavior can differ between CPUs, there exists no single "correct" semantics. Ideally, an emulator would be able to easily switch between different semantics, such that different CPU-specific semantics can be used. In other words, we argue the need for an emulator that can accurately—thus including undefined behavior—emulate many different real-world CPUs. Such an emulator should, e.g., run the Intel Celeron differently from the AMD Athlon even though they are both x86 architectures, as they implement undefined behavior differently.

Emulators usually hard-code semantics, which means that the emulator needs to be reprogrammed and then recompiled from source to change the semantics. This is unfeasible if one wants to switch between many different semantics.

We present Sem86, an x86 emulator without hard-coded semantics. Instead, semantics are provided to the emulator as data in an input file, which makes switching between different semantics trivial. Instruction semantics are encoded in a simple format, that allows for conversion from and to other formats, such as SMT-LIB [3]. All source code is available under the AGPLv3 open source license on `https://liblisa.nl/sem86`.

Despite not having hard-coded semantics, Sem86 is 1.91× as fast as Bochs and achieves 42.4% of QEMU's performance. It implements all necessary hardware to boot real x86 operating systems that support Pentium 5-era hardware, such as Windows 98, Windows XP and Windows 7. It also implements several optional hardware components for audio, high-resolution display output and internet access.

We construct a toy example that uses the IMUL instruction to detect emulation, and show that Sem86 emulates this program differently when provided with different semantics. Additionally, Sem86 is able to *bisect* execution of the program to identify at exactly which instruction emulator detection takes place.

A key question then is where we can obtain CPU-specific semantics that include semantics even for undefined behavior. To the best of our knowledge, the only source for such semantics is libLISA [10]. That tool uses program synthesis to derive instruction semantics from a CPU, by running millions of observations per instruction. They actually synthesize different undefined behavior for different x86-64 architectures. The ideal setup then is to use libLISA to synthesize instruction semantics for a set of CPUs, and use Sem86 to obtain a trustworthy CPU-specific emulator for each of these. Currently, we provide a handwritten set of semantics inspired by libLISA semantics, but in the near future we aim to replace these handwritten semantics with synthesized ones taken ad verbatim from libLISA.

As part of this work we discovered an emulation bug in Bochs. This bug caused wrong instructions to be executed under specific conditions, due to insufficient cache invalidation. We have reported this bug, which has since been fixed by the developers of Bochs.

2 Related Work

2.1 System-Level Emulators

There are many x86 emulators. Emulators such as Bochs [15] and QEMU [4] do not aim to accurately implement undefined behavior, as most software runs correct. Some emulators, such as MartyPC [2], implement fully accurate semantics and cycle-accurate timing. However, this work relies on the reverse-engineered 8086/8088 microcode and has currently not advanced beyond the original 808x CPUs. All these emulators hard-code their semantics, meaning the semantics cannot be easily changed, or extracted and translated to different formats.

Table 1: Comparison of related work.

	QEMU	Bochs	x86isa/SAIL	Gem5	Dasgupta	libLISA	Sem86
Semantics implementation	Code	Code	Data	Data	Data	Data	Data
x86 hardware implemented	✓	✓		~			✓
System-level emulation	✓	✓	✓	✓			✓
Execution performance	++	+	- -	-	- -	+	+
Complete system model			✓	✓			

2.2 x86 Semantics

Models of x86 semantics, such as the x86isa ACL2 model by Goel et al. [9], as well as its translation into SAIL [1] and Gem5 [5] allow for full-system execution. ACL2 and Gem5 can execute these semantics directly, while SAIL can generate emulators as C or OCaml code.

A lack of hardware implementations, as well as performance issues, make it impractical to run real operating systems on these emulators. The emulators do not support most of the hardware required to boot regular operating systems, and are only able to boot Linux kernels. The x86isa project requires compiling Linux from scratch because it does not implement standard x86 timer- and display functionality. Gem5 generally does not execute normal bootloaders, and instead loads a (Linux) kernel image directly. We were unable to verify if the SAIL x86 semantics are able to generate a functional emulator, as only instructions for MIPS emulation are provided.

In full-system emulation mode, the ACL2 model can execute around 320 thousand instructions per second [9]. The SAIL model has similar execution

performance when booting a FreeBSD kernel. At this performance level, it would take two to three hours to boot a typical Windows XP installation, or half a day for Windows 7. This makes it unfeasible to use these emulators for typical x86 operating systems.

Pydrofoil [6] generates a JITing emulator from the RISC-V SAIL specification and achieves a 250× speedup over the default emulator generated by SAIL. However, despite this speedup, which allows it to reach a peak of 22 million instructions executed per second in some benchmarks, its performance still falls short of QEMU and Bochs: it is still 26.7× slower than QEMU. In comparison, according to our measurements, Sem86 is just 2.1 times slower than QEMU.

Captive [24] can automatically generate emulators from a specification. In ARM benchmarks, it achieves a 1.68× speedup over QEMU. The emulator runs partially inside a virtual machine, as a bare metal program. A bare metal program can use hardware features typically only accessible by operating systems, which makes it possible to generate more efficient code, but requires a host with virtualization support (KVM). It currently only supports emulation of x86-64 in userspace mode, and does not support 32-bit x86 at all.

Many other semantics exist, but only focus on userspace [10,11,13,14,16,19]. For example, Morrisett et al. [19] implemented a Coq model for a subset of x86, for use in Google's Native Client (NaCl). The CompCert compiler [16] proves equivalence between C source code and compiled artifacts, using a Coq model for x86 which also only implements a subset of x86. While all of these semantics are executable, and some have been used to implement userspace emulators [10], none are capable of full-system emulation.

3 Implementation

In this section we describe the implementation of Sem86. We aim to build an emulator that is capable of booting real-world operating systems, such as Windows 98, Windows XP and Windows 7, loading semantics at runtime from an input file. Booting these operating systems requires executing hundreds of millions (Windows 98) up to tens of billions of instructions (Windows 7). Therefore, emulation performance is extremely important.

Sem86 provides implementations of all necessary hardware to run normal operating systems. This includes PIT, PIC, CMOS and PS/2 devices, PCI, a VBE-compatible VGA card, as well as optional hardware such as the ES1370 sound card and the NE2000 network card. We provide support for IDE disks, ATAPI CD-ROM drives, and floppy disks. For more modern operating systems, basic APIC support is also available.

Instruction semantics are loaded from a data file at runtime. Other aspects of the architecture, such as the execution loop, paging, control registers, interrupt handling and system hardware are not specified in the semantics, and are hard coded in the emulator itself. While this reduces the expressivity of the semantics, it makes it possible to implement performance optimizations such as the fast memory technique or caching.

3.1 Semantics

An emulator requires (1) a mapping from bitstrings to semantics, i.e., an *instruction decoder*, and (2) an intermediate language in which semantics can be specified.

Instruction Decoding. x86 instruction decoding typically requires an expressive language: since x86 has variable-length instructions, extra instruction bytes may need to be fetched during decoding. It is not possible to fetch all potentially-needed instruction bytes in advance, as page faults can reveal how far ahead the instruction decoder is fetching bytes.

Additionally, x86 instructions can use *prefixes*: bytes that can be prefixed once or multiple times to the instruction bitstring. Prefixes can, for example, affect register operands, change address- and operand size or override the segment register used by a memory access. There are even corner-cases where some prefixes behave differently on different CPUs [12]. Thus, instruction decoding should not be hard-coded.

Our implementation instead uses *bitpatterns* to match instructions. A bitpattern is a sequence of fixed bits (0, 1) and parts (a, b, c, ...). An instruction bitstring is decoded by matching it against the bitpattern. The result of successful decoding is a mapping of parts to matched values.

Example 1. The bitpattern `00001111 10101111 11bbbaaa` is a possible encoding of the `IMUL` instruction. Here, the parts bbb and aaa represent registers.

To decode an instruction, for example `00001111 10101111 11001010`, we first verify that all fixed bits match the instruction. Then, we compute a mapping of parts to their corresponding values. For this example, part `aaa` is `010` and part `bbb` is `001`.

To decode instruction prefixes, we use finite state automata to compute a *shortest equivalent prefix sequence*. We then add separate bitpatterns and semantics for each possible combination of prefixes.

Example 2. The `IMUL` instruction may optionally take a data-size override prefix (`66`) which reduces operand sizes to 16-bit instead of 32-bit (or vice versa when the CPU is running in 16-bit mode). This means that there will be two entries for this instruction: one with a data-size override prefix (the byte sequence `66`), and one without (the empty byte sequence ϵ). Each will have a state machine to map all possible equivalent prefix sequences:

$$\{\texttt{66}, \texttt{662E}, \texttt{2E66}, \texttt{3E66}, \dots\} \mapsto \texttt{66}$$

$$\{\epsilon, \texttt{2E}, \texttt{3E}, \dots\} \mapsto \epsilon$$

For example, when encountering an instruction prefixed with `2E`, we first reduce this to the empty byte sequence ϵ (according to the mapping provided by the finite state machine), and then match the rest of the instruction bitstring against the bitpattern.

Even though this approach is much less expressive than Turing-complete languages typically used for instruction decoding, it is sufficient to represent all x86 instructions. The downside of this approach is that it is not as compact as hand-crafted solutions: a single instruction form like `IMUL` may be encoded with dozens of different finite state machines and bitpatterns. In practice this is not a problem, as caching and JIT compilation mean that instruction decoding plays no role in emulator performance.

Intermediate Language. We encode instruction semantics in a simple intermediate language, of which the grammar is shown in Figure 1. The grammar consists of assignments, if-statements, memory- and port I/O accesses, segment descriptor loads, and various exceptional early returns. All values are 128-bit integers, which means the semantics do not require typing. Operators consist of typical integer arithmetic, bitwise operations and floating point arithmetic.

We intentionally do not implement constructs such as loops or gotos. Absence of these constructs makes it trivial to translate this intermediate language to equivalent operations in other specification languages, SMT, or provers.

Expressions to compute memory addresses are restricted to a sum of terms, where each term consists of a single register, shifted right by a constant, and then multiplied by a constant. This is sufficient to model all possible memory accesses on x86.

⟨*state-val*⟩ ::= ⟨*register*⟩
  | ⟨*memory*⟩

⟨*val*⟩ ::= ⟨*const*⟩
  | SignExt[N](⟨*state-val*⟩)
  | ⟨*state-val*⟩
  | ⟨*temp-var*⟩

⟨*statement*⟩ ::= ⟨*val*⟩
  | op(⟨*val*⟩, ⟨*val*⟩, . . .)
  | 'IF' ⟨*val*⟩ 'THEN'
    ⟨*statement*⟩*
    ELSE
    ⟨*statement*⟩*
    'FI'
  | Exception(e, ⟨*val*⟩)
  | Handler(id, ⟨*val*⟩)
  | PortIn(size, ⟨*val*⟩, ⟨*val*⟩)
  | PortOut(size, ⟨*val*⟩, ⟨*val*⟩)
  | ReadDescriptor(. . .)

Fig. 1: The grammar of Sem86's semantics. By keeping the grammar simple, it is easier to implement translation to other languages such as LLVM IR or SMT-LIB.

Example 3. We show the semantics for the `IMUL` instruction in Figure 2. There are two placeholders that depend on the output of instruction decoding: `"<a>"` and `"<b>"`. These will be substituted with the register corresponding to the value of the respective parts.

```
tmp0 := Mul(SignExt[32](<a>), SignExt[32](<b>))
<a> := tmp0
tmp1 := And(tmp0, 0xFFFFFFFF80000000)
tmp2 := Xor(tmp1, 0xFFFFFFFF80000000)
tmp3 := ite(tmp1, tmp2, tmp1)
Flag(CF) := ite(tmp3, 0x1, 0x0)
Flag(OF) := Flag(CF)
Flag(AF) := 0x0
Flag(SF) := SelectBit(31)(tmp0)
Flag(PF) := Parity(tmp0)
tmp4 := And(tmp0, 0xFFFFFFFF)
Flag(ZF) := IsZero(tmp4)
```

Fig. 2: The semantics for the IMUL instruction operating on two 32-bit register operands. First, the product is computed and stored in `tmp0` and copied to destination register `a`. Next, all flags are updated. Updates to the AF, SF, PF and ZF are undefined. In this example, the flags are updated to reflect the result of the multiplication.

The semantics for each instruction also store control-flow behavior. Control-flow behavior can be sequential, relative near jumps, and semantics-defined jumps. Sequential means that the program counter is incremented by the instruction length. Relative near jumps add a constant offset to the program counter. Optionally, adding this constant offset can be conditional on a value computed by the semantics. Semantics-defined jumps indicate that the program counter is updated by the semantics in a non-standard way. These are used to implement operations such as jumps to absolute offsets or long jumps.

Having this control-flow information available in a format that does not require analysis of the semantics themselves, makes it easy to determine the control flow of instructions. This is used, for example, to accurately determine the next instruction after a conditional jump. Without explicit control-flow information, we would have to inspect the program counter after each instruction to determine whether the jump was taken or not. However, since the control-flow behavior explicitly defines what the condition for taking the jump is, we can use this condition instead, which allows for more performant code generation.

3.2 Instruction Execution

The basic execution loop of Sem86 is a cached interpreter. When new instructions are executed, they are stored in a cache data structure as a tuple of a function pointer to the instruction execution function and a decoded representation of the instruction bitstring.

The instruction execution function is compiled by translating the instruction semantics to LLVM IR, and compiling it with LLVM. This generates functions that are similar to the manual implementations found in Bochs [15]. In our current implementation these functions are compiled on-the-fly, but could also be compiled ahead of time.

The cache data structure also contains a pointer to the instruction that will be executed next. This means that typically, no page walk, memory read, or instruction decode is necessary to determine which instruction to execute next. Executing the next instruction typically only requires following a pointer and an indirect call to the function pointer of the instruction execution function.

There are two conditions under which the cache becomes invalid: memory may be overwritten, or page mappings may change. We handle both of these conditions with page granularity.

Unlike many other architectures, x86 does not require the use of an instruction cache invalidation instruction to indicate that the instruction cache should be cleared. Instead, it is expected that any memory writes are immediately reflected in the instruction cache. When a page containing cached instructions is written, we mark it as 'dirty'. For dirty pages, the cache re-verifies memory contents for every executed instruction. This ensures that changed instructions are detected immediately.

When page mappings change, physical memory may have moved to different logical addresses. On x86, operating systems are required to explicitly signal that page mappings have changed by executing specific instructions. When this happens, we clear cached pointers to instructions on any page that has changed. This means that the execution loop will re-walk the page table to redetermine the correct pointer. Pointers between instructions on the same page can remain unchanged, as a page mapping always maps an entire page.

Memory. We implement the *fast memory* technique. This technique uses the host's MMU to accelerate memory accesses. When starting, we pre-allocate an entire 4 GiB memory region and a metadata array. The allocated memory region covers the entire x86 instruction space. This means that if the correct memory is mapped into the allocated memory region, a memory access could be performed by simply adding the emulated address to the start of the memory region. The metadata array contains flags for each page, that indicate whether memory reads and writes can be performed directly on the mapped memory region. When memory needs to be mapped at a certain location, we use `mmap` to place memory at the right location inside the 4 GiB memory region and update the metadata.

For often-executed code pages, we rely on *just-in-time* (JIT) code generation to compile sequences of instructions into native code. We choose page-wide

code generation because this allows for simple cache invalidation. Assuming code pages are not written, they can only be moved in their entirety through page remapping. This means that, although the code on the page may be moved to a different memory location in its entirety, the instructions on a single page, as well as relative jumps, will remain the same.

JIT. For JIT code generation, we use off-the-shelf LLVM. LLVM compilation is too slow to be executed in the main execution loop, so compilation is instead done on dedicated threads. The semantics of all instructions in a sequence are concatenated and transformed to LLVM IR, which is then compiled to a single function by LLVM. For jumps to other instructions on the same page, we generate inline tail calls. This avoids the overhead of the cache lookup in the interpreter loop.

3.3 Bisection and save states

Bisection consists of performing a binary search over some search space, for example the number of instructions executed, to find the point where a change in execution occurs[3]. In Section 4 we use it to analyze a toy malware sample that attempts to detect whether it is running in an emulator using undefined behavior of instructions.

The speed of bisection depends on how quickly one can determine whether a certain point in the search space exhibits the bug or not. If a full operating system needs to boot before being able to determine this, it can take a long time to search through billions of instruction executions.

SEM86 implements *save states*, i.e., capturing and storing the current CPU, memory, and system hardware state. This save state can then be stored to memory or disk, and be used later on to resume execution. The save states can be configured to also include any changes written to disk. This allows save states to be used to rewind back to an earlier point in time.

Since save states can rewind execution to earlier points in time, save states can be used to speed up bisection. Rather than restarting the emulator from scratch, bisection can resume from a save state.

4 Evaluation

In this Section, we evaluate three aspects of SEM86. First, we construct a toy malware sample and confirm SEM86 is able to identify where this sample uses undefined behavior to detect whether it is running on an emulated CPU. Second, we boot several real-world operating systems to evaluate hardware support. Finally, we evaluate the performance of SEM86 by comparing it with QEMU and Bochs.

[3] We borrow this term from software engineering [20], where bisection is typically used to identify which change introduced a bug, for example with `git bisect`.

4.1 CPU-specific semantics

CPU-specific behavior has not been well-researched. To the best of our knowledge, there exists no comprehensive overview of differences in undefined behavior, nor analysis of real-world usage of such differences for emulator detection. Because of this, it is difficult to find real malware samples (ab)using CPU-specific behavior.

```
int APIENTRY WinMain(...) {
  int result = 0;
  int a = 15;
  int b = 0;

  __asm {
    xor result, 1
    mov eax, a
    mov ecx, b
    imul eax, ecx
    jnz done
    mov result, 0
done:
  };

  if (result) {
    MessageBox(NULL, "Deleting C drive", "", 0);
  } else {
    MessageBox(NULL, "Analysis attempt detected.", "", 0);
  }

  return 0;
}
```

Fig. 3: Our toy malware sample. It forces ZF=0 by performing an `XOR` that produces a non-zero result, and then executes an `IMUL` that will produce a zero result. Finally, it checks whether the `IMUL` instruction has modified the ZF. This is then used to show one of two message boxes: one message box represents malicious behavior, while the other message box represents silently exiting. The sample was compiled under Windows 98 using Visual Studio 6.0.

Instead, we evaluate Sem86's ability to use different semantics by constructing a toy malware sample that relies on CPU-specific behavior. Our toy malware sample determines whether it is running inside an emulator by executing the `IMUL` instruction, and checking the value that is stored in the zero flag (ZF) afterwards. The source code of our toy malware sample is shown in Figure 3.

The ZF is undefined for the `IMUL` instruction, and is often implemented differently both in actual CPUs and emulators. Bochs currently implements the ZF

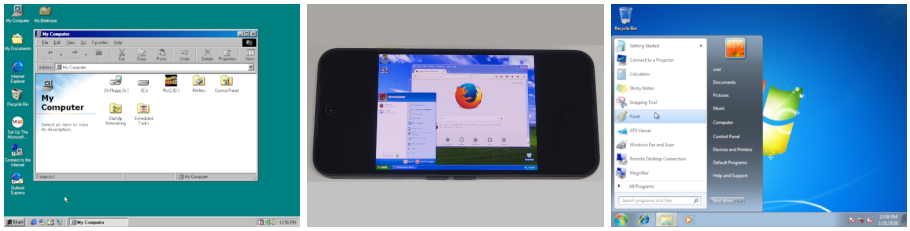
by setting it to 1 if the result was zero, and setting it to 0 otherwise. Although modern CPUs implement various different behaviors, for the purposes of this experiment we assume that a real CPU would not modify ZF, which appears to have been common at the time when this technique was detected [23].

We generated two different semantics: one where `IMUL` sets the ZF to the normal result, which behaves like Bochs, and one where `IMUL` does not modify the ZF. Starting SEM86 with the semantics where `IMUL` behaves like Bochs, shows the "Analysis attempt detected" message, while starting it with the semantics where `IMUL` does not behave like Bochs correctly shows the "Deleting C drive" message.

In order to evaluate bisection, we set the toy malware sample to automatically start upon boot, and then ran a bisection on the display output of the emulator after 400 million executed instructions. For real malware, other characteristics such as file system contents or network access attempts could be used instead of the display output. The bisection was able to automatically identify the `IMUL` instruction as being the instruction that determines which message box is shown.

4.2 Booting real-world operating systems

SEM86 implements all hardware required to boot x86 operating systems. We evaluated this by booting Windows 98, Windows XP and Windows 7. We were able to boot all of these operating systems, as depicted in Figure 4.



(a) Windows 98 SE. (b) Windows XP running in SEM86 on an Android phone. (c) Windows 7.

Fig. 4: Various operating systems running in SEM86.

The ES1370 sound card and the NE2000 networking card cannot be used under Windows 7. While these work on Windows 98 and XP, no drivers are available for Windows 7. In order to enable sound and networking for Windows 7 and newer, more modern hardware needs to be implemented.

To demonstrate that SEM86's implementation is portable, we have ported it to Android. We tested this on an Android phone that has 8 GiB of RAM. While

it is fully functional, it can sometimes, crash due to the memory usage of LLVM, which can reach 4-5 GiB during JIT compilation of some instruction sequences.

4.3 Performance

We have evaluated the performance of Sem86 using CPUMark'99 running on Windows XP. This is a single-threaded benchmark that evaluates integer performance of CPUs, which was commonly used to evaluate CPU performance in the 90s. After completing the benchmark, it displays a single score that summarizes CPU performance. The results are presented in Table 2. We also list how many million instructions were executed per second (MIPS) during the benchmark.

Table 2: The CPUMark'99 score of various emulators, as measured on an AMD 3900X host CPU. The score is the best of 5 runs. MIPS listed is the highest reached during all runs. Since QEMU does not report MIPS, this column is left blank.

Emulator	Score	MIPS
QEMU (10.1.0)	89.8	-
Bochs (3.0)	19.9	74.31
Sem86	38.1	192.7

We compare against QEMU and Bochs, because they are at opposite ends of the optimization spectrum: Bochs is purely an interpreter, and does not use any JIT optimization techniques for portability reasons. On the other hand, QEMU uses bespoke JIT code generation to translate guest instruction traces into native code, and has been heavily optimized for performance.

We believe the difference in performance between Sem86 and QEMU is mostly due to JIT code generation quality. QEMU uses a bespoke JIT, while we rely on off-the-shelf LLVM. LLVM is a general-purpose compiler library, which is not geared specifically to emulators. The code generated by LLVM is likely of lower quality than that generated by QEMU. For example, because we do not have full control over code generation, LLVM sometimes generates larger function preludes that preserve more registers on the stack. While this would likely benefit normal programs, it degrades performance when functions are called hundreds of millions of times per second.

Additionally, QEMU performs basic block linking, where jumps to the next basic block are inlined directly into the block. This avoids the overhead of returning to the execution loop, and looking up the next basic block in the cache data structures. While Sem86 also does this for basic blocks within the same page, we have not implemented this for jumps between different pages.

5 Discussion and Conclusion

We introduced Sem86, an emulator for x86 architectures that treats the instruction semantics to be emulated as configurable input data. As such, it can be configured to emulate, e.g., CPU-specific semantics for accurate and precise emulation. It could also be used to emulate semantics written in the context of academic research efforts, e.g., ACL2 [9] or SAIL [1]. Sem86 runs typical x86 operating systems, such as Windows 98, Windows XP and Windows 7. Our evaluation shows that its performance lies between Bochs and QEMU.

One particular combination that we are looking into is using libLISA's automatically inferred semantics [10]. This would allow "CPU cloning", i.e., analyzing a CPU, extracting its semantics, and then starting an emulator that emulates that exact CPU accurately. Semantics generated by libLISA would not cover all available instructions. For example, libLISA is unable to analyze privileged instructions. These would still need to be manually specified. However, such instructions are a tiny subset of all x86 instructions, reducing the manual effort to a minimum. The main obstacle for using libLISA's semantics is that libLISA currently does not support 16-bit and 32-bit x86 modes. Solving this is mostly an engineering effort. Most importantly, a CPU observer for 16-bit and 32-bit modes would need to be built.

For CPUs that support virtualization, building a CPU observer requires building a small VM kernel that can execute instructions and observe results. We are currently attempting to build such a CPU observer, and hope to use this to analyze a CPU and export its semantics for use in Sem86.

Data-Availability Statement

All source code is available on `https://liblisa.nl/sem86` under the AGPLv3 open source license.

Acknowledgments. This paper is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Prime Contract No. N66001-21-C-4028, and by DARPA under Prime Contract No. HR001124C0492. Any views, opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints of publications for Government purposes notwithstanding any copyright notation hereon.

References

1. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami,

N., Sewell, P.: Isa semantics for armv8-a, risc-v, and cheri-mips. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290384, https://doi.org/10.1145/3290384
2. Balsom, D.: Martypc (2023), https://github.com/dbalsom/martypc
3. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
4. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track. vol. 41, pp. 10–55. California, USA (2005)
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH computer architecture news **39**(2), 1–7 (2011)
6. Bolz-Tereick, C., Panayi, L., McKeogh, F., Spink, T., Berger, M.: Pydrofoil: Accelerating Sail-Based Instruction Set Simulators. In: Aldrich, J., Silva, A. (eds.) 39th European Conference on Object-Oriented Programming (ECOOP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 333, pp. 3:1–3:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ECOOP.2025.3, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.3
7. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: 2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN). pp. 177–186. IEEE (2008)
8. Choi, Y., Jeong, Y., Jang, D., Kang, B.B., Lee, H.: Emuid: Detecting presence of emulation through microarchitectural characteristic on arm. Computers & Security **113**, 102569 (2022)
9. Coglio, A., Goel, S.: Adding 32-bit mode to the acl2 model of the x86 isa. arXiv preprint arXiv:1810.04313 (2018)
10. Craaijo, J., Verbeek, F., Ravindran, B.: liblisa: instruction discovery and analysis on x86-64. Proceedings of the ACM on Programming Languages **8**(OOPSLA2), 333–361 (2024)
11. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: 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. pp. 1133–1148 (2019)
12. Domas, C.: Breaking the x86 isa (2017), https://github.com/xoreaxeaxeax/sandsifter, accessed on 02/05/2024
13. Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from i/o samples. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 441–452. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254116, https://doi.org/10.1145/2254064.2254116
14. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86-64 instruction set. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 237–250 (2016)
15. Lawton, K.P.: Bochs: A portable pc emulator for unix/x. Linux Journal **1996**(29es), 7–es (1996)
16. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814

17. Ma, W., Liu, J.C., Forin, A.: Design and testing of a cpu emulator. Texas A&M University, Microsoft Research Technical Report, United States of America **1**, 12 (2009)
18. Martignoni, L., Paleari, R., Roglia, G.F., Bruschi, D.: Testing cpu emulators. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 261–272 (2009)
19. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger sfi for the x86. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 395–404. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254111, https://doi.org/10.1145/2254064.2254111
20. Ness, B., Ngo, V.: Regression containment through source change isolation. In: Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97). pp. 616–621 (1997). https://doi.org/10.1109/CMPSAC.1997.625082
21. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: International Conference on Information Security. pp. 1–18. Springer (2007)
22. Sahin, O., Coskun, A.K., Egele, M.: Proteus: Detecting android emulators from instruction-level profiles. In: International Symposium on Research in Attacks, Intrusions, and Defenses. pp. 3–24. Springer (2018)
23. SonicWall: New anti-vm technique observed in the wild (2017), http://web.archive.org/web/20201026062235/https://www.mysonicwall.com/sonicalert/searchresults.aspx?ev=article&id=1085, accessed on 16/01/2026
24. Spink, T., Wagstaff, H., Franke, B.: A retargetable system-level dbt hypervisor. ACM Transactions on Computer Systems (TOCS) **36**(4), 1–24 (2020)