Testing CPU emulators

Lorenzo Martignoni\textsuperscript{1}  Roberto Paleari\textsuperscript{2}
Giampaolo Fresi Roglia\textsuperscript{2}  Danilo Bruschi\textsuperscript{2}

\textsuperscript{1}Università degli Studi di Udine
\textsuperscript{2}Università degli Studi di Milano

International Symposium on Software Testing and Analysis – 2009
class BOCHSAPI BX_CPU_C {
    public:
    
    // General register set
    bx_gen_reg_t gen_reg[8];
    Bit32u eflags;

    // user segment register set
    bx_segment_reg_t sregs[6];

    // system segment registers
    bx_global_segment_reg_t gdtr;
    bx_global_segment_reg_t idtr;
    bx_segment_reg_t ldtr;
    bx_segment_reg_t tr;

    // Control registers
    bx_cr0_t cr0;
    bx_address cr2;
    bx_address cr3;
    bx_cr4_t cr4;

All that via software

Lorenzo Martignoni, Roberto Paleari et al. Testing CPU emulators
Developing CPU emulators

- Any hardware system can be emulated via software

However...

- only superheroes can develop fully-featured CPU emulators for CISC architectures (e.g., IA-32):
  - CPU emulators are very complex pieces of software
  - IA-32 specification is 4 books long!
  - The specifications sometimes lack details or contain ambiguities
Developing CPU emulators

- Any hardware system can be emulated via software

  However...  

- only superheroes can develop fully-featured CPU emulators for CISC architectures (e.g., IA-32):
  - CPU emulators are very complex pieces of software
  - IA-32 specification is 4 books long!
  - The specifications sometimes lack details or contain ambiguities

CPU emulators are unlikely to be bug-free
The importance of testing CPU emulators

- CPU emulators are used for multiple purposes:
  - OSes and drivers development
  - Compilers development
  - Debugging
  - Profiling
  - Security analysis

- It is easy to imagine the cascade effects of bugs in emulators

- Bugs could be prevented with testing, but no specific testing methodology for CPU emulators exists
Our contribution at a glance

A methodology for testing CPU emulators to detect if they properly emulate a given CPU

- Completely automatic
- Based on fuzz-testing and differential-testing
- Prototype implementation for IA-32 emulators
- Used to test four state-of-the-art CPU emulators
1. Introduction
2. (Un)faithful CPU emulation
3. Our testing methodology
4. Experimental results
5. Conclusions
Modeling the CPU

To compare the behavior of the physical CPU with the behavior of the emulator we need to model the CPU and the way it operates.
A simple abstraction of the CPU

▶ We can imagine the CPU as an abstract machine
▶ A state \( s = (pc, R, M, E) \) of the abstract machine consists of:

- \( pc \) – state of the program counter
- \( R \) – state of the CPU registers
- \( M \) – state of the (virtual) memory
- \( E \) – exception state (e.g., \( \perp \), PF, GPF, UD)

▶ State information not directly observable by a program (e.g., content of the caches) are ignored
Abstraction of instructions execution

To execute an instruction means to transition into a new state:

\[ s = (pc, R, M, E) \quad \rightarrow \quad s' = (pc', R', M', E') \]
Abstraction of instructions execution

To execute an instruction means to transition into a new state:

\[ s = (pc, R, M, E) \quad \rightarrow \quad s' = (pc', R', M', E') \]

The execution can result in different output states:

\[ s = (pc, R, M, E) \]

- An exception occurs:
  \[ s' = (pc, R, M, E') \]

- The program is terminated:
  \[ s' = (pc', R', M', \bot) \]

\[ s' = (pc, R, M, E) \]
(Un)faithful CPU emulation

\[ s = (pc, R, M, \perp) \]
(Un)faithful CPU emulation

\[ s = (pc, R, M, \bot) \]
(Un)faithful CPU emulation

\[ s = (pc, R, M, \bot) \]

\[ s' = (pc', R', M', E') \]

\[ s'' = (pc'', R'', M'', E'') \]
(Un)faithful CPU emulation

The emulator **faithfully emulates** the CPU, for the state $s$, if the state resulting from the execution in the two environments match
(Un)faithful CPU emulation

The emulator **unfaithfully emulates** the CPU, for the state $s$, if the state resulting from the execution in the two environments differ.

### Mathematical Representation

$$s = (pc, R, M, \bot)$$

$$s' = (pc', R', M', E')$$

$$s'' = (pc'', R'', M'', E'')$$

The emulator unfaithfully emulates the CPU, for the state $s$, if the state resulting from the execution in the two environments differ.
An example of **faithful** CPU emulation

CPU state \((R)\)
- **eax**: 0x00000000
- **esp**: 0xbfe7d4e4
- **fs**: 0x007b

Memory state \((M)\)
- \(0x08090000\): \texttt{mov} \$0x1, %eax
- \(0x08090005\): \texttt{push} %fs
- \(0xbfe7d4e0\): aa bb cc dd

Exception state \((E)\)
\(\perp\)

---

CPU state \((R)\)
- **eax**: 0x00000001
- **esp**: 0xbfe7d4e4
- **fs**: 0x007b

Memory state \((M)\)
- \(0x08090000\): \texttt{mov} \$0x1, %eax
- \(0x08090005\): \texttt{push} %fs
- \(0xbfe7d4e0\): aa bb cc dd

Exception state \((E)\)
\(\perp\)
An example of **faithful** CPU emulation

CPU state \((R)\)
- **eax**: 0x00000000
- **esp**: 0xbfe7d4e4
- **fs**: 0x007b

Memory state \((M)\)
- 0x08090000: `mov $0x1, %eax`
- 0x08090005: `push %fs`
- ... 
- 0xbfe7d4e0: aa bb cc dd

Exception state \((E)\)
- ⊥

execute current instruction

CPU state \((R)\)
- **eax**: 0x00000001
- **esp**: 0xbfe7d4e4
- **fs**: 0x007b

Memory state \((M)\)
- 0x08090000: `mov $0x1, %eax`
- 0x08090005: `push %fs`
- ... 
- 0xbfe7d4e0: aa bb cc dd

Exception state \((E)\)
- ⊥

execute current instruction

CPU state \((R)\)
- **eax**: 0x00000000
- **esp**: 0xbfe7d4e4
- **fs**: 0x007b

Memory state \((M)\)
- 0x08090000: `mov $0x1, %eax`
- 0x08090005: `push %fs`
- ... 
- 0xbfe7d4e0: aa bb cc dd

Exception state \((E)\)
- ⊥

execute current instruction

Lorenzo Martignoni, Roberto Paleari et al. Testing CPU emulators
An example of **faithful** CPU emulation

<table>
<thead>
<tr>
<th>CPU state (R)</th>
<th>CPU state (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax: 0x00000000</td>
<td>eax: 0x00000001</td>
</tr>
<tr>
<td>esp: 0xbfe7d4e4</td>
<td>esp: 0xbfe7d4e4</td>
</tr>
<tr>
<td>fs: 0x007b</td>
<td>fs: 0x007b</td>
</tr>
</tbody>
</table>

**Memory state (M)**
- 0x08090000 mov $0x1, %eax
- 0x08090005 push %fs
- ...
- 0xbfe7d4e0 aa bb cc dd

**Exception state (E)**

**Output states match → Emulator is faithful**

execute current instruction

execute current instruction
An example of **unfaithful** CPU emulation

CPU state \((R)\)
\[
\begin{align*}
eax & \quad 0x00000001 \\
esp & \quad 0xbfe7d4e4 \\
fs & \quad 0x007b \\
\end{align*}
\]

Memory state \((M)\)
\[
\begin{align*}
0x08090000 & \quad \text{mov } \$0x1, \ %eax \\
0x08090005 & \quad \text{push } \%fs \\
\ldots & \quad \ldots \\
0xbfe7d4e0 & \quad \text{aa bb cc dd} \\
\end{align*}
\]

Exception state \((E)\)
\[
\bot
\]

CPU state \((R)\)
\[
\begin{align*}
eax & \quad 0x00000001 \\
esp & \quad 0xbfe7d4e0 \\
fs & \quad 0x007b \\
\end{align*}
\]

Memory state \((M)\)
\[
\begin{align*}
0x08090000 & \quad \text{mov } \$0x1, \ %eax \\
0x08090005 & \quad \text{push } \%fs \\
\ldots & \quad \ldots \\
0xbfe7d4e0 & \quad \text{aa bb cc dd} \\
\end{align*}
\]

Exception state \((E)\)
\[
\bot
\]

Output states differ

The emulator is unfaithful
An example of **unfaithful** CPU emulation

---

**CPU state (R)**
- eax: 0x00000001
- esp: 0xbfe7d4e4
- fs: 0x007b

**Memory state (M)**
- ban: 0x08090000
- mov $0x1, %eax
- push %fs
- ...
- 0xbfe7d4e0 aa bb cc dd

**Exception state (E)**

---

**CPU state (R)**
- eax: 0x00000001
- esp: 0xbfe7d4e0
- fs: 0x007b

**Memory state (M)**
- 0x08090000
- mov $0x1, %eax
- push %fs
- ...
- 0xbfe7d4e0 aa bb cc dd

**Exception state (E)**

---

**CPU state (R)**
- eax: 0x00000001
- esp: 0xbfe7d4e0
- fs: 0x007b

**Memory state (M)**
- 0x08090000
- mov $0x1, %eax
- push %fs
- ...
- 0xbfe7d4e0 7b 00 cc dd

**Exception state (E)**

---

**CPU state (R)**
- eax: 0x00000001
- esp: 0xbfe7d4e0
- fs: 0x007b

**Memory state (M)**
- 0x08090000
- mov $0x1, %eax
- push %fs
- ...
- 0xbfe7d4e0 7b 00 00

**Exception state (E)**
An example of **unfaithful** CPU emulation

CPU state \((R)\)
- eax: 0x00000001
- esp: 0xbfe7d4e4
- fs: 0x007b

Memory state \((M)\)
- 0x08090000: mov $0x1, %eax
- 0x08090005: push %fs
- ... ...
- 0xbfe7d4e0: aa bb cc dd

Exception state \((E)\)

CPU state \((R)\)
- eax: 0x00000001
- esp: 0xbfe7d4e4
- fs: 0x007b

Memory state \((M)\)
- 0x08090000: mov $0x1, %eax
- 0x08090005: push %fs
- ... ...
- 0xbfe7d4e0: aa bb cc dd

Exception state \((E)\)

Output states differ \(\Rightarrow\) Emulator is **unfaithful**

execute current instruction

CPU state \((R)\)
- eax: 0x00000001
- esp: 0xbfe7d4e4
- fs: 0x007b

Memory state \((M)\)
- 0x08090000: mov $0x1, %eax
- 0x08090005: push %fs
- ... ...
- 0xbfe7d4e0: 7b 00 cc dd

Exception state \((E)\)

execute current instruction

CPU state \((R)\)
- eax: 0x00000001
- esp: 0xbfe7d4e0
- fs: 0x007b

Memory state \((M)\)
- 0x08090000: mov $0x1, %eax
- 0x08090005: push %fs
- ... ...
- 0xbfe7d4e0: 7b 00 00 00

Exception state \((E)\)
For each CPU state $s$ verify whether the CPU emulator faithfully emulates the physical CPU
Our testing methodology

For each CPU state $s$ verify whether the CPU emulator faithfully emulates the physical CPU

**EmuFuzzer**

1. Generate a test-case, i.e., a CPU state $s = (pc, R, M, \bot)$
2. Generate a program that:
   2.1 Initializes the CPU to $s$
   2.2 Executes the instruction at $pc$
   2.3 Dumps the state $s'$ resulting from the execution of the instruction
3. Run the program in the physical CPU and in the emulator
4. Compare the resulting states
Our testing methodology

For each CPU state $s$ verify whether the CPU emulator faithfully emulates the physical CPU

**EmuFuzzer**

1. Generate a test-case, i.e., a CPU state $s = (pc, R, M, \bot)$
2. Generate a program that:
   2.1 Initializes the CPU to $s$
   2.2 Executes the instruction at $pc$
   2.3 Dumps the state $s'$ resulting from the execution of the instruction
3. Run the program in the physical CPU and in the emulator
4. Compare the resulting states
Test-case generation

The size of the state space is unmanageable!

Lorenzo Martignoni, Roberto Paleari et al. Testing CPU emulators
Test-case generation

The size of the state space is unmanageable!
Test-case generation

Random generation

CPU-assisted generation
Test-case generation

Random generation
Random data and code

CPU-assisted generation
Test-case generation

Random generation
Random data and code

CPU-assisted generation

Test-cases including valid and invalid instructions
Test-case generation

Random generation
Random data and code

CPU-assisted generation
Random data and algorithmically generated code

▶ Use the CPU to explore the instruction set:
  ▶ Detect strings representing valid instructions
  ▶ Infer the format of instructions
  ▶ Generate few instruction instances for each opcode found

Lorenzo Martignoni, Roberto Paleari et al. Testing CPU emulators
Test-case generation

Random generation
Random data and code

CPU-assisted generation
Random data and algorithmically generated code

Test-cases including only valid instructions, covering the entire instruction set, and minimizing redundancy
Execution of the test-case

Just mixed C & assembly code

We hook the emulator to intercept the execution of instructions and exceptions
Experimental setup

Tested emulators

- QEMU (0.9.1)
- Valgrind (3.3.1)
- Pin (2.5-23100) © Intel
- BOCHS (2.3.7)

Baseline physical CPU

- Intel Pentium 4 (3 GHz)
Experimental setup

Tested emulators

- QEMU (0.9.1)
- Valgrind (3.3.1)
- Pin (2.5-23100) \(\text{© Intel}\)
- BOCHS (2.3.7)

Baseline physical CPU

- Intel Pentium 4 (3 GHz)

Test-cases

- About 3 millions (70% generated randomly and the remaining 30% using CPU assisted generation)
- Randomly selected 10% of them
Overview of the results

Testing took about 24 hours
(15 test-cases per second on average)
Overview of the results

Distinct mnemonic opcodes unfaithfully emulated

State differences:  
- **R**  
- **M**  
- **E**

**BOCHS**

**Pin**

**Valgrind**

**QEMU**
Overview of the results

Distinct mnemonic opcodes unfaithfully emulated

State differences: $R$, $M$, $E$

Differences in the status register, in the general purposes registers, and FPU

BOCHS

Pin

Valgrind

QEMU
Overview of the results

Distinct mnemonic opcodes unfaithfully emulated

State differences: $R$, $M$, $E$

Differences in the content of the memory

BOCHS
Pin
Valgrind
QEMU
Overview of the results

Distinct mnemonic opcodes unfaithfully emulated

BOCHS
Pin
Valgrind
QEMU

State differences: R M E

Not supported instructions, over supported instructions, and other exceptions
Overview of the results

Distinct mnemonic opcodes unfaithfully emulated

For each deviation discovered it is possible to write a program that executes correctly in the physical CPU, but crashes in the emulated CPU (or vice versa)
Some of the defects found

QEMU
- Certain inputs freeze the emulator
- Floating point instructions accept unaligned operands

BOCHS
- Exceptions might corrupt the stack
- Some floating point instructions operate incorrectly

Valgrind
- Many privileged instructions do not rise exceptions if executed without privileges
- Instructions not executed atomically

Pin
- Exceptions are not always risen properly
- Pushes and pops of segment registers corrupt the stack
Conclusions

- Fully automated testing methodology for CPU emulators
- Optimized algorithm for test-cases generation: systematic exploration of the instruction set with minimized redundancy
- EmuFuzzer: a prototype implementation of the proposed methodology for IA-32 emulators
- Extensive testing of four IA-32 emulators that resulted in the discovery of several defects in each of them
Future work

Extend the prototype to:

- Work independently of the emulator
- Test privileged code
- Test VT emulation in QEMU and BOCHS
Testing CPU emulators

\[ s = (pc, R, M, \bot) \]

\[ s' = (pc', R', M', E') \]

\[ ? \]

\[ s'' = (pc'', R'', M'', E'') \]

Thank you!
Any questions?

Lorenzo Martignoni
lorenzo.martignoni@uniud.it