A system for testing specifications of CPU semantics

or,
What I did on my summer vacation

Lindsey Kuper
Static analysis of executables: what, why, and how?
Static analysis of executables: what, why, and how?

- Abstract interpretation: a sound overapproximation of the behavior of a concrete system.
Static analysis of executables: what, why, and how?

- Abstract interpretation: a *sound overapproximation* of the behavior of a concrete system.
Static analysis of executables: what, why, and how?

- Abstract interpretation: a sound overapproximation of the behavior of a concrete system.
- What kinds of static analysis can one do on executables?
Static analysis of executables: what, why, and how?

- Abstract interpretation: a *sound overapproximation* of the behavior of a concrete system.


- What kinds of static analysis can one do on executables?

- Value-set analysis (VSA)
Static analysis of executables: what, why, and how?

- Abstract interpretation: a *sound overapproximation* of the behavior of a concrete system.


- What kinds of static analysis can one do on executables?
  - Value-set analysis (VSA)
  - Quantifier-free bit-vector (QFBV) analysis
Static analysis of executables: what, why, and how?

- Abstract interpretation: a *sound overapproximation* of the behavior of a concrete system.
- What kinds of static analysis can one do on executables?
  - Value-set analysis (VSA)
  - Quantifier-free bit-vector (QFBV) analysis
  - ...

Tuesday, October 19, 2010
The TSL testing problem
The TSL testing problem

Lim, J, and Reps, T., "A System for Generating Static Analyzers from Machine Instructions", CC '08
The TSL testing problem

- TSL (Transformer Specification Language) lets us generate static analyzers from specifications. Great!

Lim, J, and Reps, T., “A System for Generating Static Analyzers from Machine Instructions”, CC ’08
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!
The TSL testing problem

- **TSL** (Transformer Specification Language) lets us generate static analyzers from specifications. Great!

- But how do we know if the generated analysis engines (*multiplicatively many!*!) are correct?
Our approach
Our approach

- ISA spec (IA32, PPC, ...) written in TSL language
- Analysis spec aka "interpretation"

TSL system

Analysis engine
Our approach

- Narrow the focus to testing just the ISA specs.
Our approach

- Narrow the focus to testing just the ISA specs.
Our approach

- Narrow the focus to testing just the ISA specs.
- Can we really *isolate* an ISA spec? We can come close by using EMUL, the “simplest” interpretation.
Our approach

- Narrow the focus to testing just the ISA specs.
- Can we really *isolate* an ISA spec? We can come close by using EMUL, the “simplest” interpretation.
Our approach

- Narrow the focus to testing just the ISA specs.
- Can we really *isolate* an ISA spec? We can come close by using EMUL, the “simplest” interpretation.
Our approach

- Narrow the focus to testing just the ISA specs.
- Can we really isolate an ISA spec? We can come close by using EMUL, the “simplest” interpretation.
- And for now, start with IA32.
Goal for the summer
Goal for the summer

- Find out how complete and precise our IA32 TSL specification is...
Goal for the summer

• Find out how complete and precise our IA32 TSL specification is...

• ...by generating an IA32 emulator, then comparing the emulator to the real processor.
Goal for the summer

- Find out how complete and precise our IA32 TSL specification is...
- ...by generating an IA32 emulator, then comparing the emulator to the real processor.
- If resulting states differ on the same inputs, the spec was (probably) buggy.
Goal for the summer

• Find out how complete and precise our IA32 TSL specification is...
  • ...by generating an IA32 emulator, then comparing the emulator to the real processor.
  • If resulting states differ on the same inputs, the spec was (probably) buggy.
  • We already have all the pieces: IA32 spec, EMUL, and a third-party tool for testing CPU emulators. This will be easy, right?!
A brief digression: abstract interpretation, $\gamma$, and $\alpha$
A brief digression:
abstract interpretation, \( \gamma \), and \( \alpha \)
How to test a CPU emulator
How to test a CPU emulator

EmuFuzzer’s design

Emulated environment

Randomly chosen state

\[ E_0 \]

Interpret instruction

\[ E_1 \]

\[ E_1' \]

\[ E_1 \equiv E_1' \]

Physical environment

Add eax, 4

\[ P_0 \]

Execute instruction

\[ P_1 \]

Martignoni, L. et al., “Testing CPU Emulators”, ISSTA ’09
How to test a CPU emulator

EmuFuzzer’s design

Emulated environment

Randomly chosen state

\( E_0 \)

"Concretize"

Interpret instruction

\( E_i \)

\( E_i' \)

\( \equiv \)

Physical environment

\( P_0 \)

add eax, 4

Execute instruction

\( P_1 \)

\( \alpha \)

"Abstract"

Martignoni, L. et al., “Testing CPU Emulators”, ISSTA ’09

Tuesday, October 19, 2010
How to test a CPU emulator

EmuFuzzer’s design

But wait!

Emulated environment

Randomly chosen state

“Concretize”

Interpret instruction

Physical environment

add eax, 4

Execute instruction

We don’t only want to test ordinary emulators.

Martignoni, L. et al., “Testing CPU Emulators”, ISSTA ’09

Tuesday, October 19, 2010
How to test any analysis engine (someday)
How to test any analysis engine

Our design

Abstract environment

- $E_0$
- $E_1$
- $E_1'$

Physical environment

- $P_0$
- $P_1$

Interpret instruction

Randomly (or deliberately) chosen state

Add eax, 4

Execute instruction

α

"Abstract"
Look-thru memory
Look-thru memory

"Hey, what did you say eax was again?"

E₀

α
does nothing
at first!

P₀

"eax = 2"
Look-thru memory

- The ability to *lazily* instantiate the emulator's state (memory and registers) from that of the process as each instruction is being emulated.
TSL validator main loop
TSL validator main loop

1. Halt test program $P$ at instruction $i$
2. Clear emulator's memory
3. Emulate $i$ using look-thru memory
4. Write-protect all of $P$'s memory pages
5. Try to single-step $P$ on instruction $i$
6. Write fault? Unprotect and try again
7. Repeat for instruction $i + 1$
TSL validator main loop

- Halt test program $P$ at instruction $i$
- Repeat for instruction $i + 1$
- Clear emulator's memory
- Try to single-step $P$ on instruction $i$
- Write fault? Unprotect and try again
- Emulate $i$ using look-thru memory
- Write-protect all of $P$'s memory pages
TSL validator main loop

- Halt test program $P$ at instruction $i$
- Repeat for instruction $i + 1$
- Clear emulator's memory
- Try to single-step $P$ on instruction $i$
- Write fault? Unprotect and try again
- Emulate $i$ using look-thru memory
- Write-protect all of $P$'s memory pages

Diagram:
- $E_0$ and $P_0$ connected with a dashed line.
TSL validator main loop

1. Halt test program $P$ at instruction $i$
2. Repeat for instruction $i + 1$
3. Clear emulator's memory
4. Try to single-step $P$ on instruction $i$
5. Write-protect all of $P$'s memory pages
6. Write fault? Unprotect and try again
7. Emulate $i$ using look-thru memory

Diagram:

- $E_0$ to $P_0$ with arrow $\alpha$
TSL validator main loop

1. Halt test program $P$ at instruction $i$
2. Repeat for instruction $i + 1$
3. Clear emulator's memory
4. Try to single-step $P$ on instruction $i$
5. Write-fault? Unprotect and try again
6. Emulate $i$ using look-thru memory
7. Write-protect all of $P$'s memory pages

Diagram:
- $E_0$ to $P_0$ with a dashed line labeled $\alpha$
TSL validator main loop

1. Halt test program $P$ at instruction $i$
2. Repeat for instruction $i + 1$
3. Clear emulator's memory
4. Try to single-step $P$ on instruction $i$
5. Write fault? Unprotect and try again
6. Emulate $i$ using look-thru memory
7. Write-protect all of $P$'s memory pages

Diagram:
- $E_0$ connected to $P_0$
TSL validator main loop

- Halt test program $P$ at instruction $i$
- Repeat for instruction $i + 1$
- Clear emulator's memory
- Emulate $i$ using look-thru memory
- Write-protect all of $P$'s memory pages
- Try to single-step $P$ on instruction $i$
- Write fault? Unprotect and try again

State comparison is tricky
TSL validator main loop

State comparison is tricky

Halt test program $P$ at instruction $i$

Repeat for instruction $i + 1$

Clear emulator's memory

Emulate $i$ using look-thru memory

Write-protect all of $P$'s memory pages

Try to single-step $P$ on instruction $i$

Write fault? Unprotect and try again
TSL validator main loop

- Halt test program $P$ at instruction $i$
- Clear emulator's memory
- Repeat for instruction $i + 1$
- Emulate $i$ using look-thru memory
- Write-protect all of $P$'s memory pages
- Try to single-step $P$ on instruction $i$
- Write fault? Unprotect and try again
- State comparison is tricky
Future work:
short-term stuff
Future work: short-term stuff

- **The hard part of state comparison**: identify changed locations on the real process side, and compare them with corresponding locations on the emulator side.
Future work: short-term stuff

- **The hard part of state comparison**: identify changed locations on the real process side, and compare them with corresponding locations on the emulator side.

- Better logging and reporting: eventually, we’d like to have a “dashboard” that will tell us roughly how complete and correct the existing TSL specifications are.
Future work: short-term stuff

- **The hard part of state comparison**: identify changed locations on the real process side, and compare them with corresponding locations on the emulator side.

- Better logging and reporting: eventually, we’d like to have a “dashboard” that will tell us roughly how complete and correct the existing TSL specifications are.

- How to deal with test programs that “misbehave”? (Programs that mess with their own memory protection, install their own seg fault handlers, …)
Future work:
long-term stuff
Future work:
long-term stuff

ISA spec
(IA32, PPC, ...)
written in TSL language

Analysis spec
aka "interpretation"

TSL system

Analysis engine
Future work: long-term stuff

- Support for more ISAs. (x64, at least!)
Future work: long-term stuff

- Support for more ISAs. (x64, at least!)
- **Support for true abstract interpretations**, not just EMUL.
Future work: long-term stuff

- Support for more ISAs. (x64, at least!)
- **Support for true abstract interpretations**, not just EMUL.
- Find ways to choose which inputs to test that will be most likely to turn up bugs in a specification.
What I learned
What I learned

- Emulators, debuggers, and static analyzers are not made of magic
What I learned

- Emulators, debuggers, and static analyzers are not made of magic
- First real systems programming experience: didn’t quite cross the kernel space boundary, but came right up next to it
What I learned

- Emulators, debuggers, and static analyzers are not made of magic
- First real systems programming experience: didn’t quite cross the kernel space boundary, but came right up next to it
- A metric for how much I can accomplish in 13 weeks
What I learned

• Emulators, debuggers, and static analyzers are not made of magic

• First real systems programming experience: didn’t quite cross the kernel space boundary, but came right up next to it

• A metric for how much I can accomplish in 13 weeks

• Finally convinced that OOP is good for something
Thank you!

Questions?