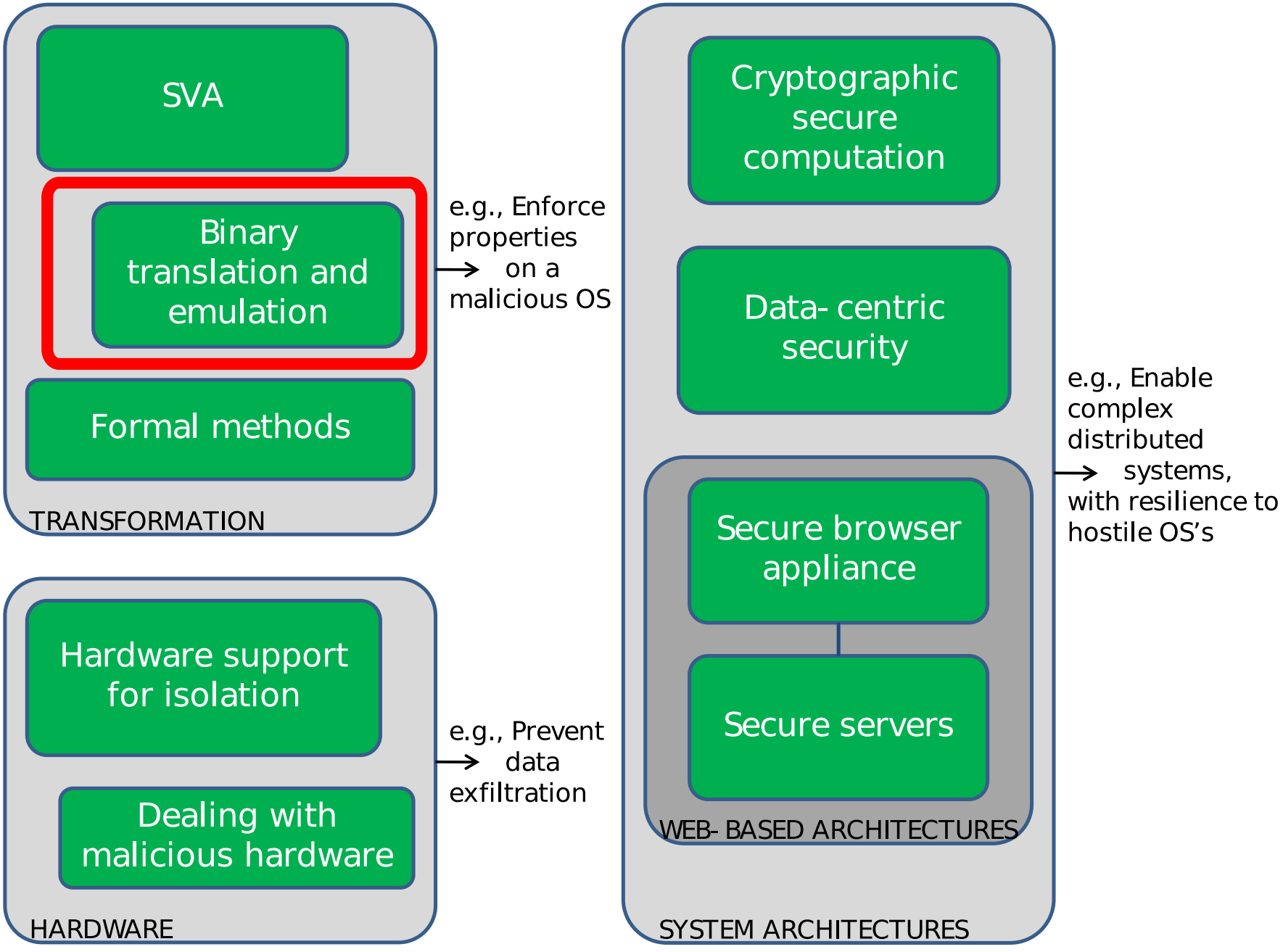

SVA
Binary translation and emulation
Formal methods
TRANSFORMATION
e.g., Enforce properties on a malicious OS
Hardware support for isolation
Dealing with malicious hardware
HARDWARE
Cryptographic secure computation
Data- centric security
Secure browser appliance
Secure servers
WEB- BASED ARCHITECTURES
SYSTEM ARCHITECTURES
e.g., Enable complex distributed systems, with resilience to hostile OS's

Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators

Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam[†], Dawn Song, and Petros Maniatis[*]

{martigno,smcc,dawnsong}@cs.berkeley.edu, ppoosank@cmu.edu, petros.maniatis@intel.com

UC Berkeley, [†]CMU, and [*]Intel Labs

Goal: compare emulators and real CPU

- Processor emulators are used e.g., in VMs to isolate one OS from another, or for taint tracking
- But processor spec. is complex, hard to implement correctly
- Perform systematic evaluation to ensure emulators match behavior of real hardware

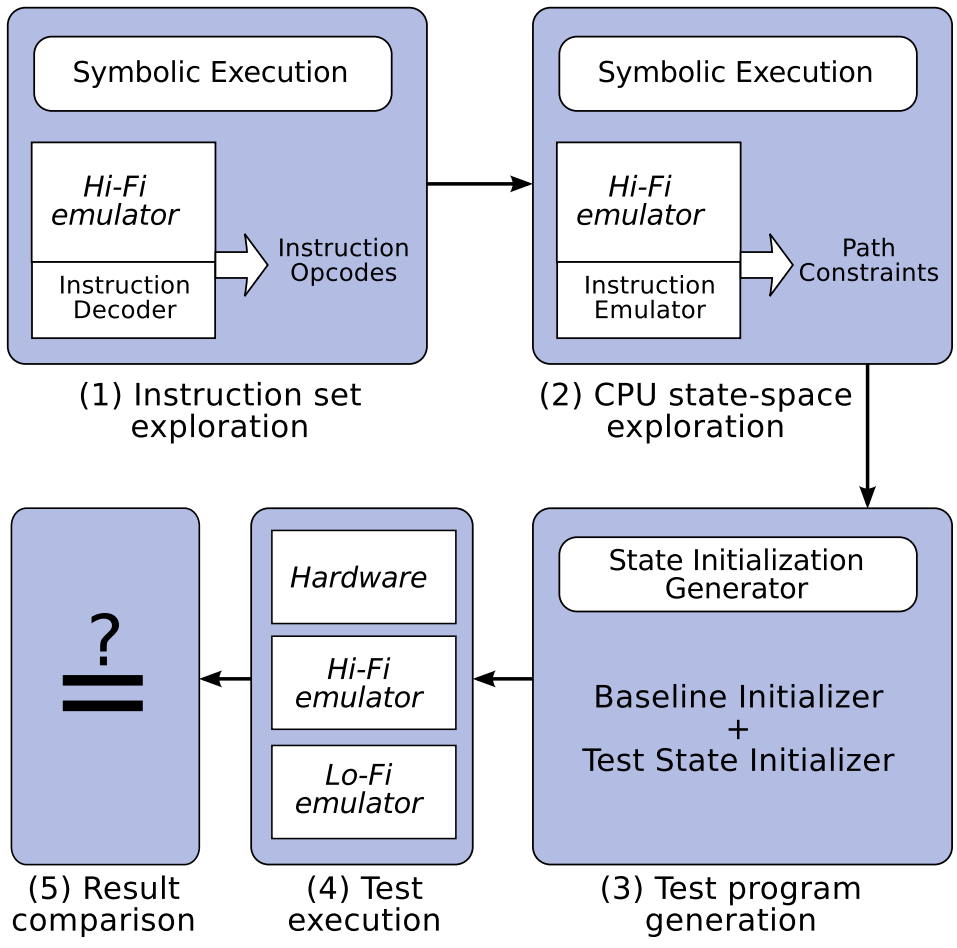
Connection with other DHOSA work

- Berkeley:
  - Extract machine models from existing emulator implementations, and
  - Use them to compare and find bugs
- Harvard:
  - Design a logical machine model, and
  - Use it to build binary-level systems with correctness proofs
- Stony Brook:
  - Extract machine models from an existing compiler implementation, and
  - Use them for ahead-of-time rewriting

Test generation from a hi-fi emulator

- Assume we have a *high fidelity* emulator which is close to correct
  - But may have other disadvantages, e.g. speed
- Perform *path-exploration lifting*: explore behavior of hi-fi emulator, generate test cases
- Run tests on other emulators and on real hardware

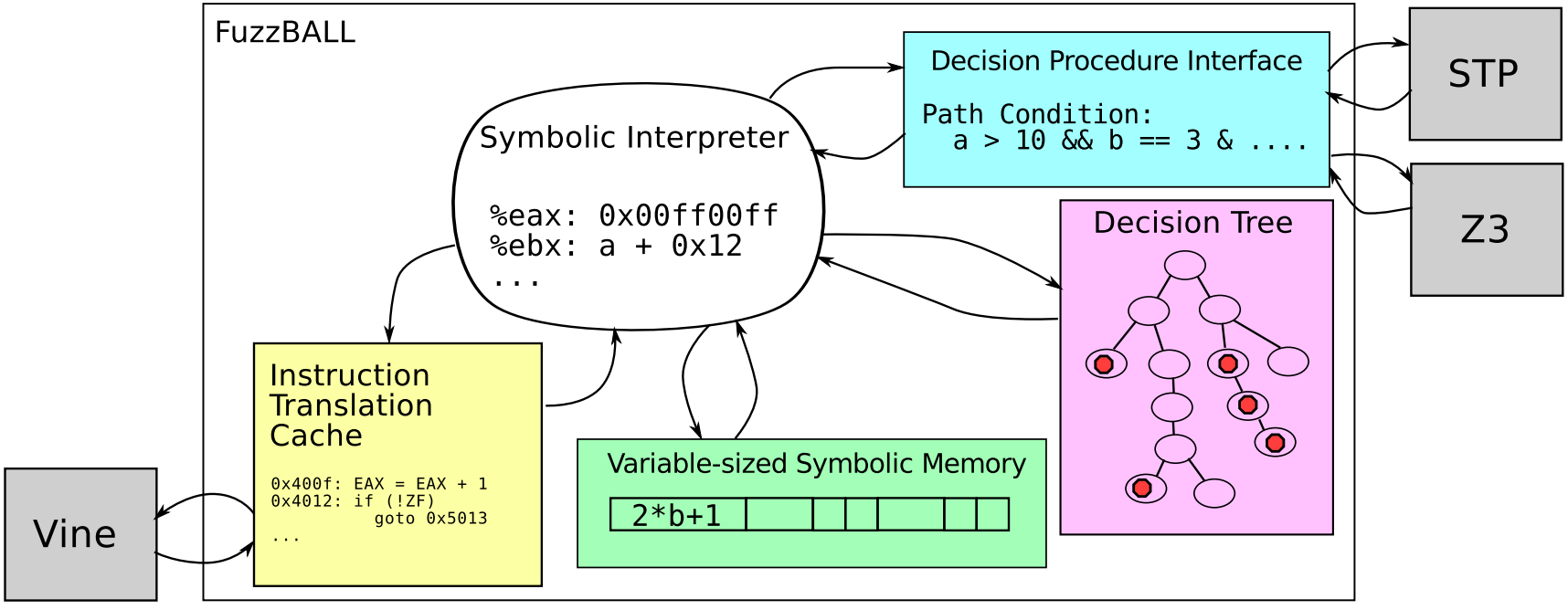
Road map


Symbolic Execution
Hi-Fi emulator
Instruction Decoder
Instruction Opcodes
(1) Instruction set exploration
Symbolic Execution
Hi-Fi emulator
Instruction Emulator
Path Constraints
(2) CPU state-space exploration
State Initialization Generator
Baseline Initializer
+
Test State Initializer
(3) Test program generation
Hardware
Hi-Fi emulator
Lo-Fi emulator
(4) Test execution
?
(5) Result comparison

Overview of symbolic execution

- *Symbolic execution* explores feasible program paths based on a *symbolic input*
- Replace input with variables, computations produce formulas
  - Solve branch conditions with a decision procedure
- A symbolic execution path can represent many concrete executions, but is still precise
  - Complete, but state-space may be large

Lightweight symb. exec.: FuzzBALL

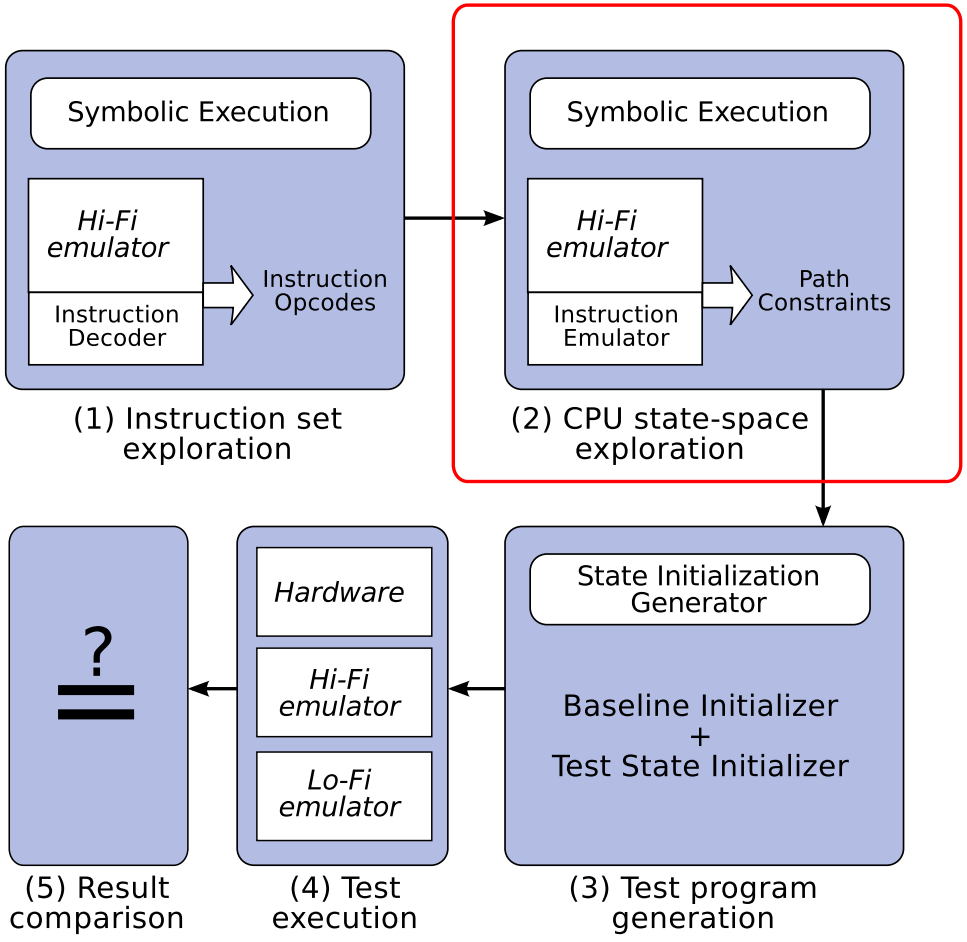


Online symbolic execution, tailored to the challenges of the binary level

Explore space of instructions

- Explore instruction decoder with symbolic instruction bytes
  - 3 bytes $\rightarrow$ 16,777,216 possible sequences
- Choose one byte sequence per decoder execution path
  - 68,977 paths
- Further select one byte sequence per emulator execution routine
  - 880 final byte sequences

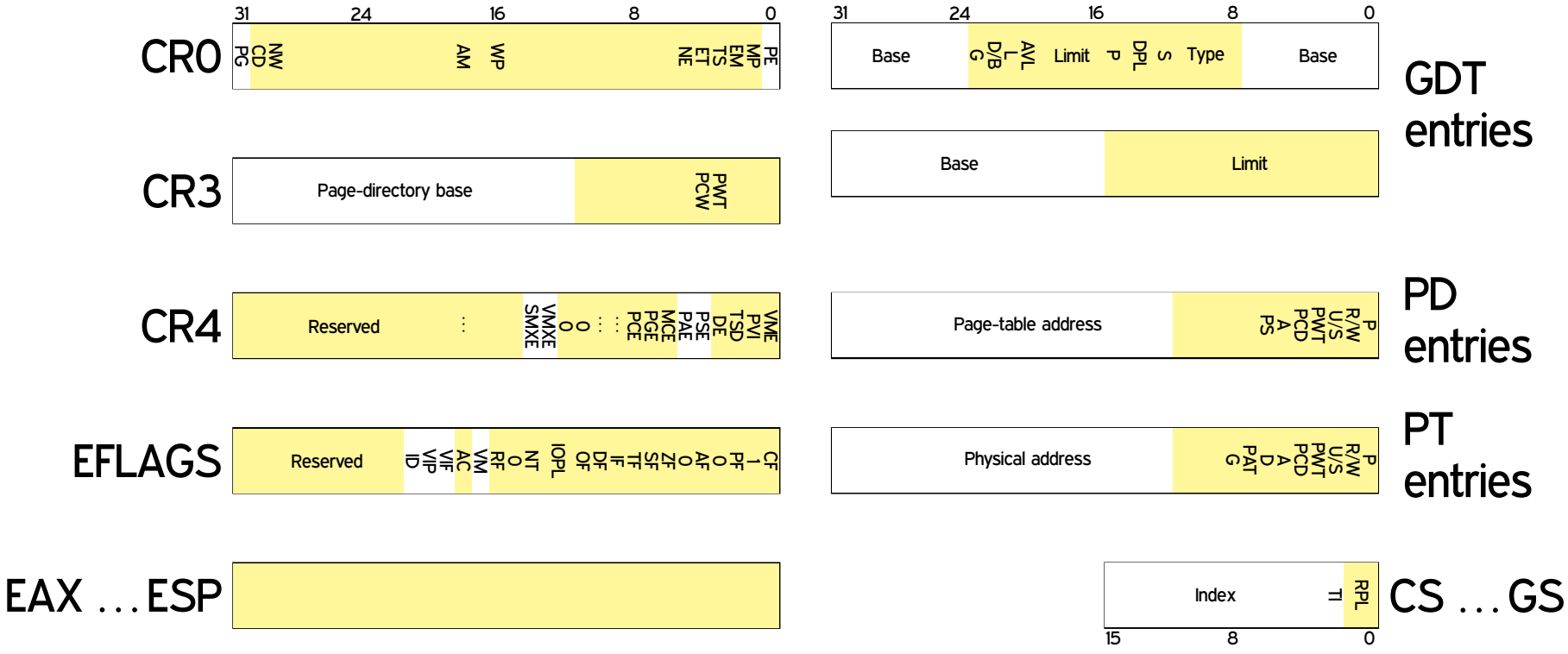
Road map


Symbolic Execution
Hi-Fi emulator
Instruction Decoder
Instruction Opcodes
(1) Instruction set exploration
Symbolic Execution
Hi-Fi emulator
Instruction Emulator
Path Constraints
(2) CPU state-space exploration
State Initialization Generator
Baseline Initializer
+
Test State Initializer
(3) Test program generation
Hardware
Hi-Fi emulator
Lo-Fi emulator
(4) Test execution
?
(5) Result comparison

Explore space of machine states

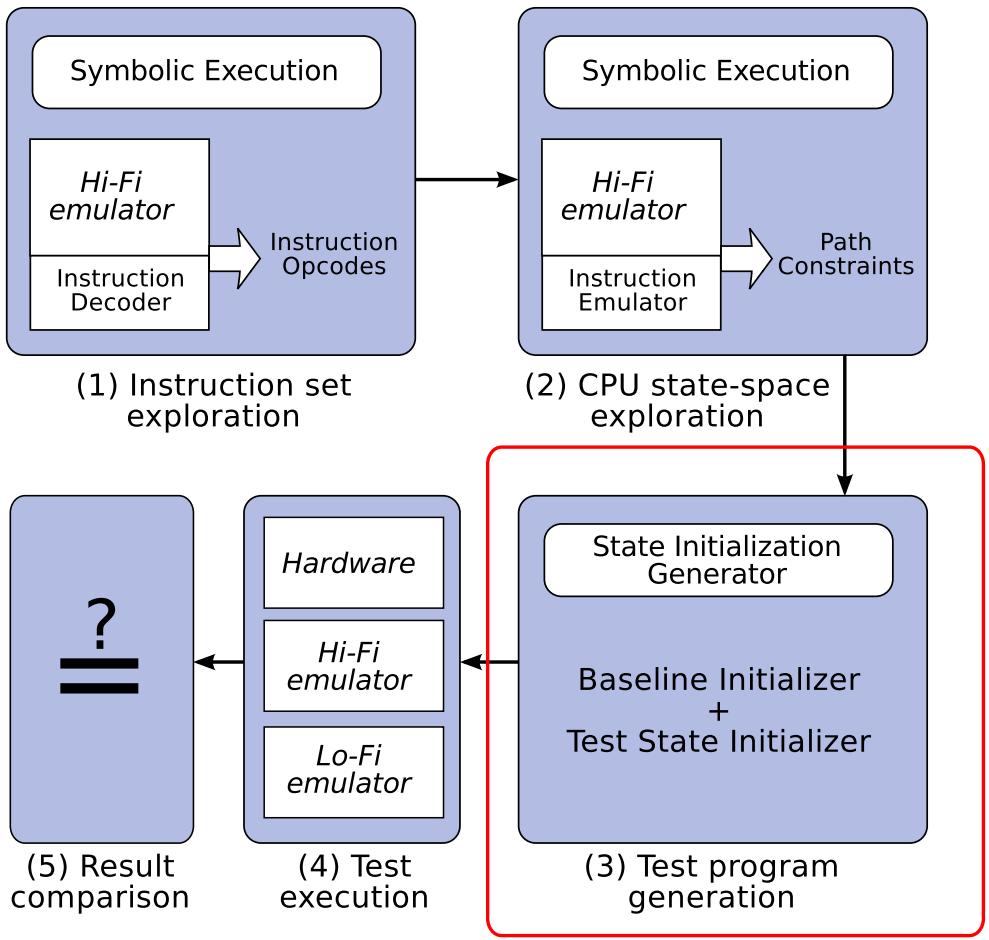
- Explore instruction emulator with symbolic machine state
  - Discover which parts of the CPU and memory state affect an instruction's execution
- Optimization: generate symbolic summaries of repeated computations
  - Saves a factor of 23^6 in segment cache

Controlling exploration

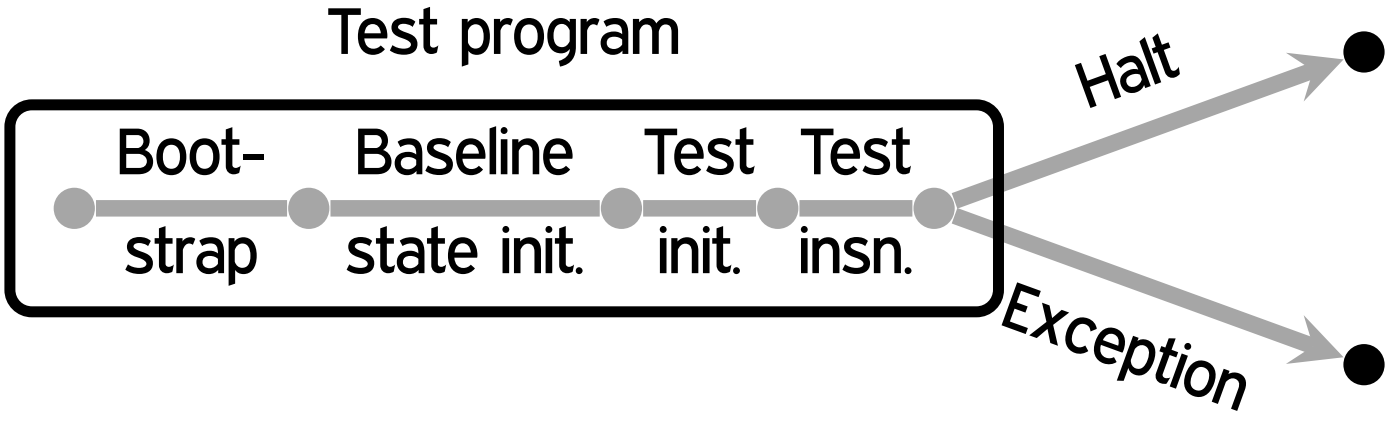


- Symbolic state is about 400 bytes in registers and tables, plus all remaining memory bytes
- Pointer-like values are not symbolic

Road map


Symbolic Execution
Hi-Fi emulator
Instruction Decoder
Instruction Opcodes
(1) Instruction set exploration
Symbolic Execution
Hi-Fi emulator
Instruction Emulator
Path Constraints
(2) CPU state-space exploration
State Initialization Generator
Baseline Initializer
+
Test State Initializer
(3) Test program generation
Hardware
Hi-Fi emulator
Lo-Fi emulator
(4) Test execution
?
(5) Result comparison

Generate concrete tests



- For each path, we have a concrete CPU state that triggers the path
- Generate instructions to initialize state
- Combine with bootloader and baseline initializer to create stand-alone disk image
- Save machine state after end of execution or an exception

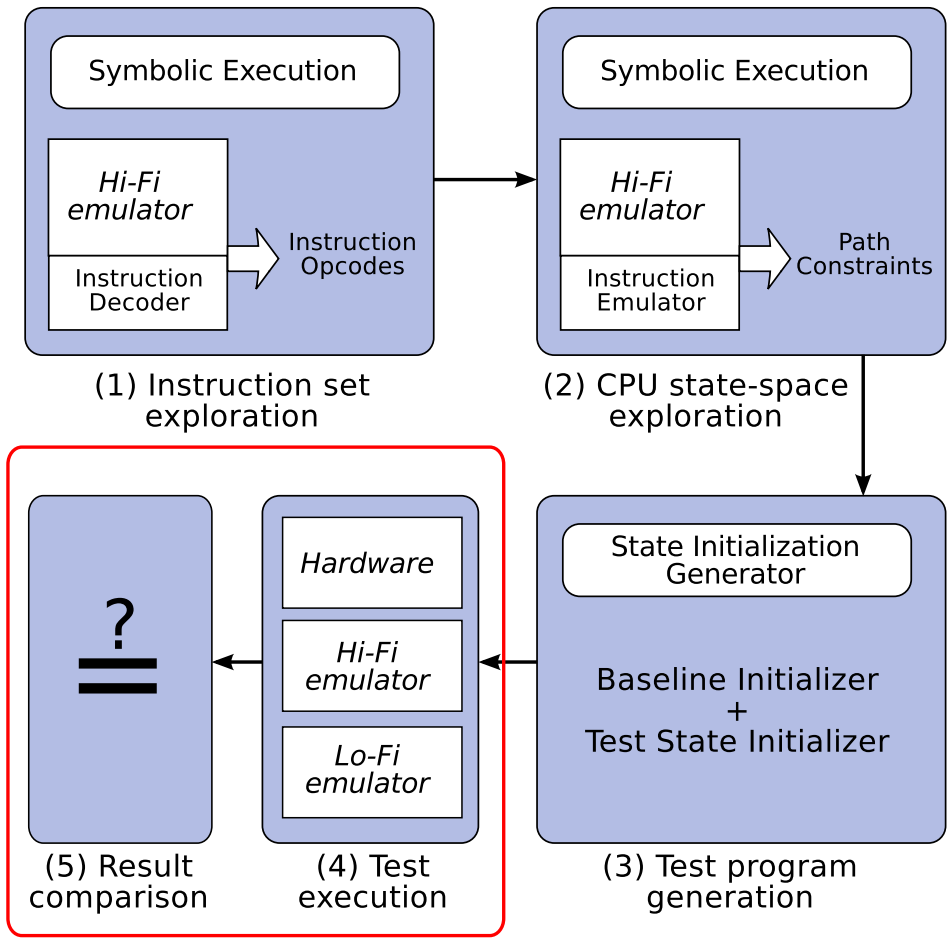
Test generation details

- Prefer test states to be as similar as possible to the baseline state
  - Post-process symbolic execution results with greedy bit flipping
- Initializer generation must be dependency-aware
  - E.g., instruction sequence to set `%ss` modifies `%eax`
  - Compute a topological ordering for each test

Test generation statistics

- Hi-fi emulator: Bochs 2.4.5
- 880 instruction representatives
- 610,516 paths (and test cases)
  - Exhaustive for 95% of instructions
- 545 CPU-core-hours, highly parallelizable
  - About $55 on Amazon EC2

Road map


Symbolic Execution
Hi-Fi emulator
Instruction Decoder
Instruction Opcodes
(1) Instruction set exploration
Symbolic Execution
Hi-Fi emulator
Instruction Emulator
Path Constraints
(2) CPU state-space exploration
State Initialization Generator
Baseline Initializer
+
Test State Initializer
(3) Test program generation
Hardware
Hi-Fi emulator
Lo-Fi emulator
(4) Test execution
?
(5) Result comparison

Behavioral differences found

- Compared latest versions of QEMU and Bochs to an Intel Core i5 CPU
- Test execution takes another 815 core hours (or $80)
- Out of 610,516 tests, 64,692 distinguishable in QEMU and 15,404 distinguishable in Bochs
  - Atomicity violations in QEMU's `leave` and `cmpxchgw`
  - Missing segment limit and permissions checks in QEMU
  - Differing orders of memory accesses
  - Differing values of "undefined" status flags

Next steps

- Extend symbolic execution to more emulators
  - Apply binary level tool to just-in-time instruction translation (e.g., QEMU), closed source VMs (e.g., VMware)
- Perform complete equivalence checking over explored instructions
  - Combine formulas for multiple paths
  - Can prove behavior is identical for all symbolic inputs

Summary

- Need better testing to make emulators trustworthy
- Today: exhaustive tests with path exploration lifting from a hi-fi emulator
- Future: more emulators, more complete equivalence checking

`http://bitblaze.cs.berkeley.edu/`

Thanks!