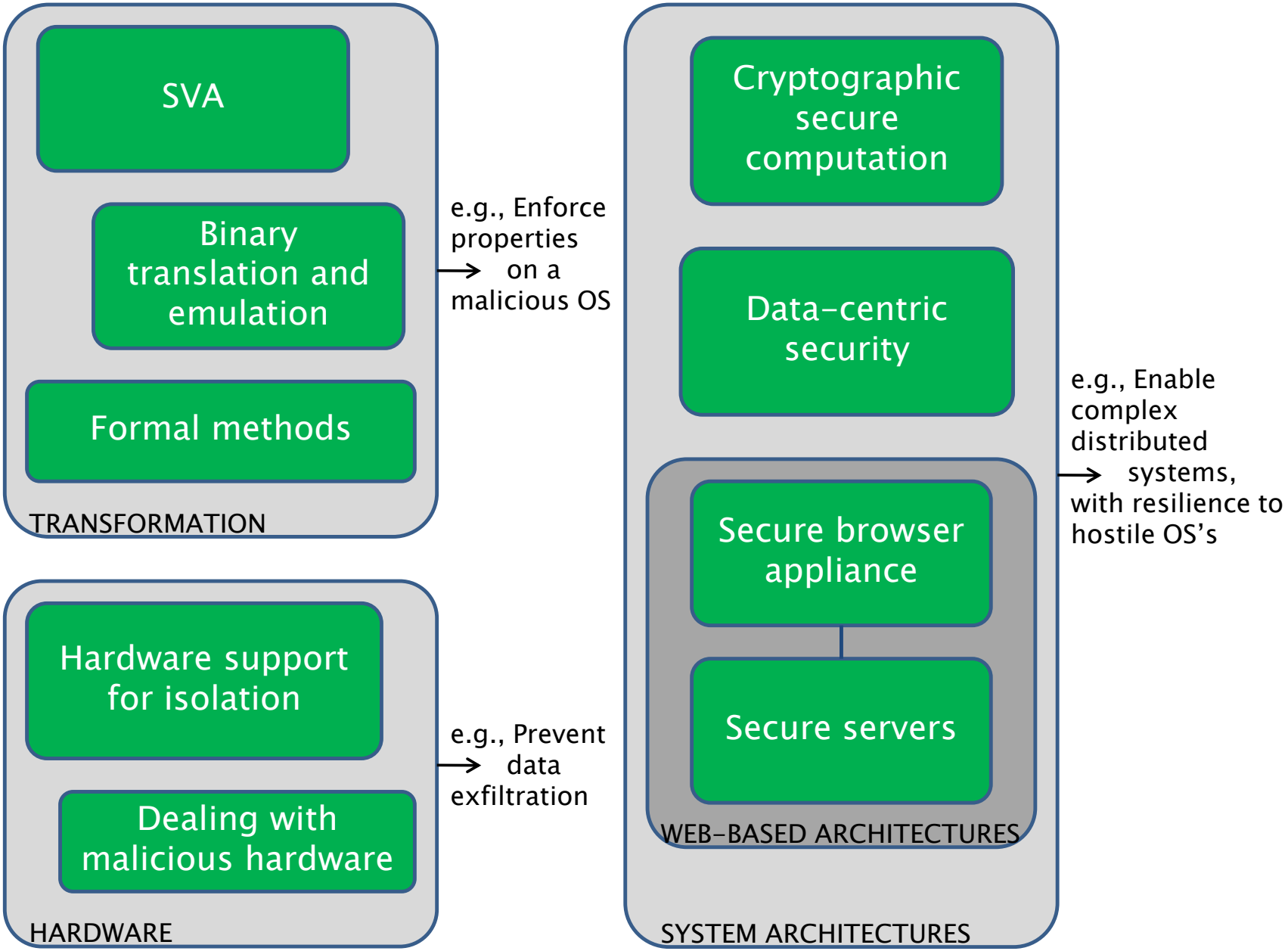

SVA
Binary translation and emulation
Formal methods
TRANSFORMATION
e.g., Enforce properties on a malicious OS
Hardware support for isolation
Dealing with malicious hardware
HARDWARE
e.g., Prevent data exfiltration
Cryptographic secure computation
Data-centric security
Secure browser appliance
Secure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

Exhaustive Testing for Trustworthy CPU Emulators

Lorenzo Martignoni, Stephen McCamant, Dawn Song, and Petros Maniatis*

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

UC Berkeley 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

- Harvard:
  - Design logical machine model, and
  - Use it to build binary-level systems with correctness proofs
- Berkeley:
  - Extract logical machine models from existing implementations, and
  - Use them to compare and find bugs
- Illinois:
  - Use emulator to recover from untrusted hardware

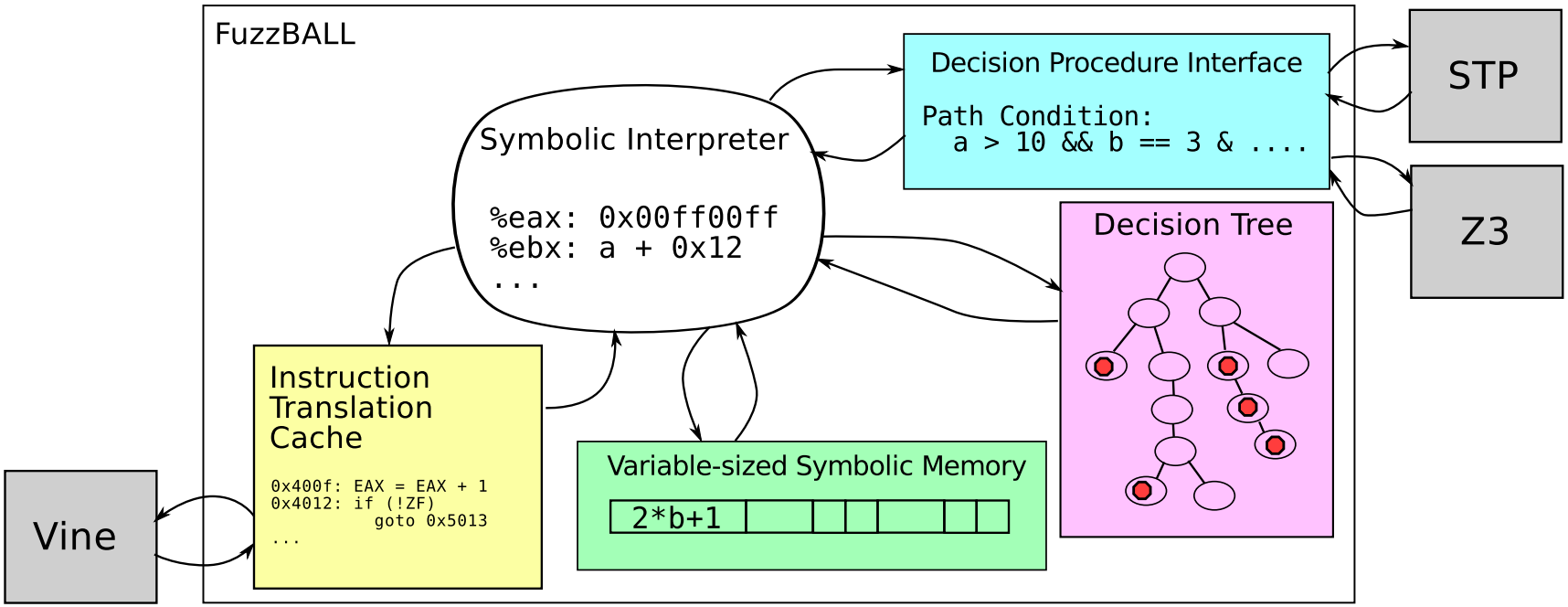
Test generation from a hi-fi emulator

- 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

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



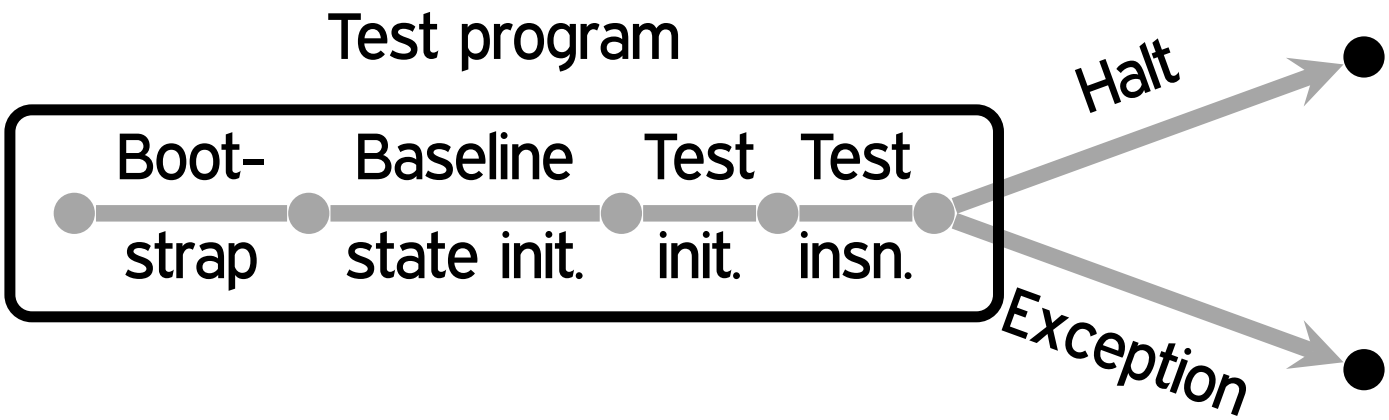
Online symbolic execution, tailored to the challenges of the binary level

Explore instruction and CPU states

Two applications of symbolic execution:

1. Explore instruction decoder with symbolic instruction bytes $\longrightarrow$ determine space of possible instructions
2. Explore instruction emulator with symbolic machine state $\longrightarrow$ determine CPU states that affect an instruction's execution

Generate and execute concrete tests



- For each path, generate a concrete CPU state that triggers the path
- Dependency-aware initialization instructions
- Combine with bootloader and baseline initializer to create stand-alone disk image
- Run tests on hi-fi, lo-fi emulators and a hardware-based VM

Behavioral differences found

- Compared latest versions of QEMU and Bochs to an Intel Core i5 CPU
- Generated 84395 tests for 919 instructions, 3575 tests showed differences:
  - 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

Thanks!