embedded conference scandinavia

Fuzzing Hard-to-get-at Embedded Software with Virtual Platforms

Jakob Engblom, Intel, Sweden – jakob.engblom@intel.com
Robert Guenzel, Intel, Germany

intel

Fuzzing? Why and What?

Software testing technique

Sends “random” inputs to a software component

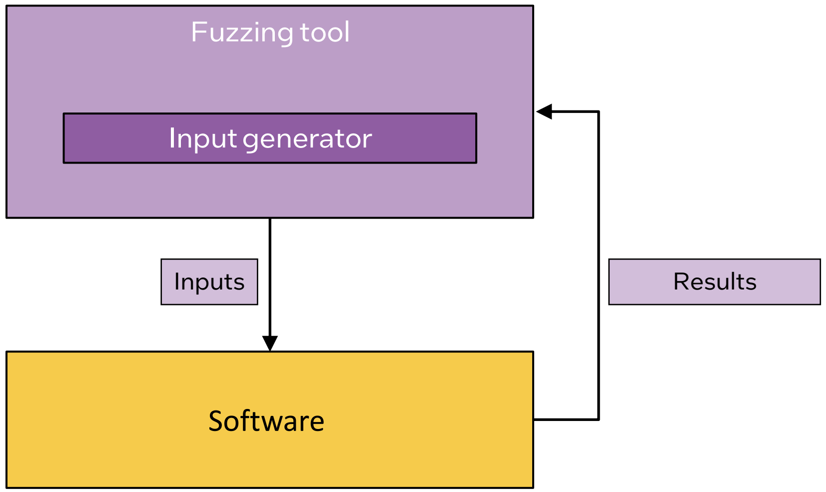
- Inputs should follow the “syntax” of real inputs
- Observe the results, look for errors

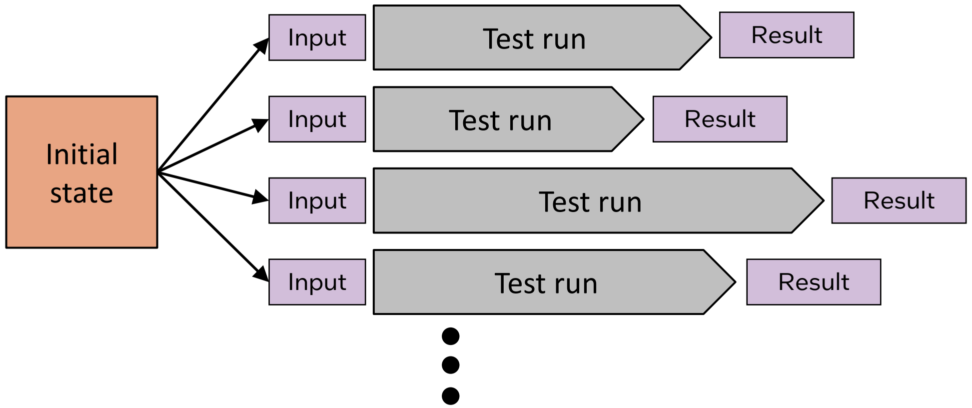
Finds more problems than manually written tests

- Explore corner cases that developers did not think about
- Automation = large volume of variant tests

Iterate tests from the same initial state

Huge area of research today! How can we take advantage of it for hard-to-get-at code?





Coverage-Guided Fuzzing

However... totally random tests are very inefficient – mostly thrashing

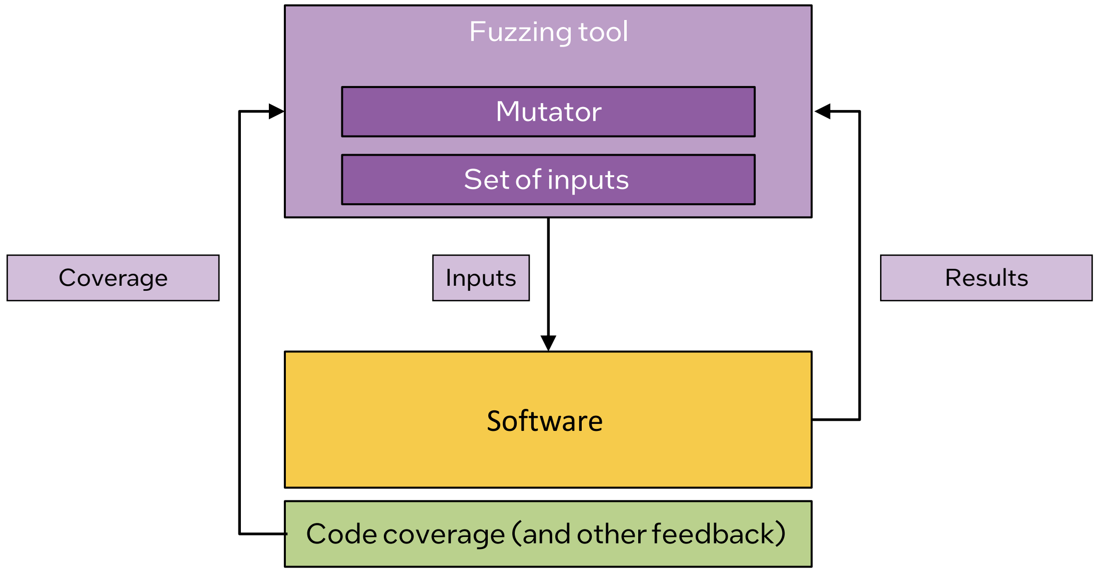
Solution: **coverage-guided fuzzing**

- Discern the code reached by each test
  - I.e., measure the "goodness" of a test case
- Can be done without source code!

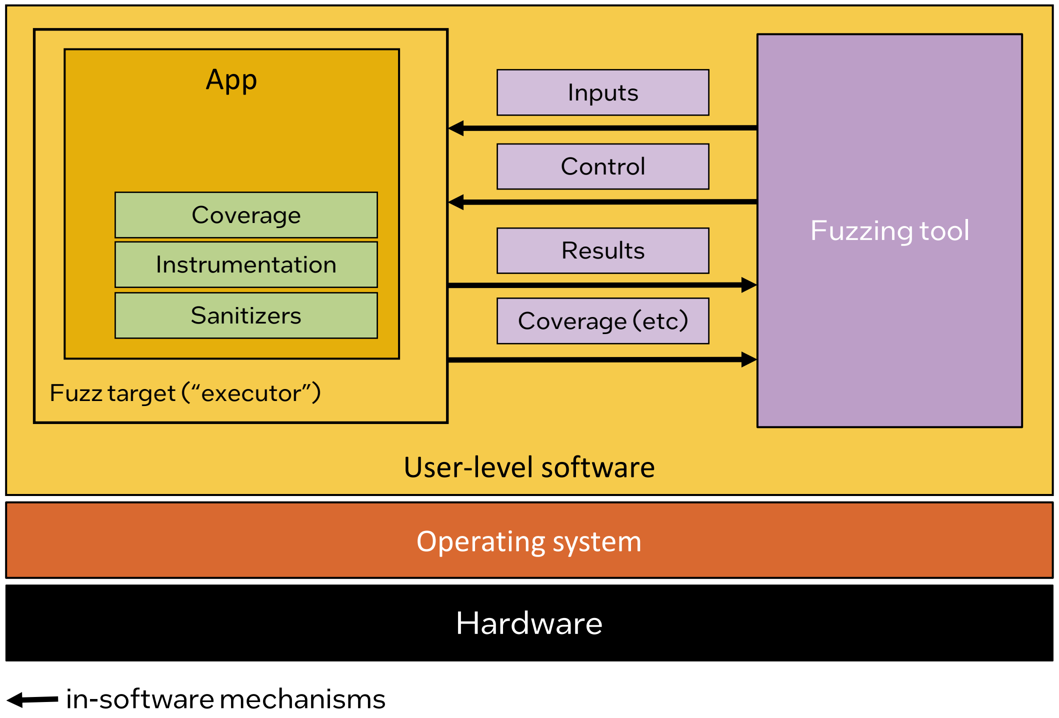
Mutate the set of test cases

- Evolve towards effective tests that are as different from each other as feasible
- Discard tests that do not add coverage

Improves fuzzing effectiveness massively



Standard Coverage-Guided Fuzzing

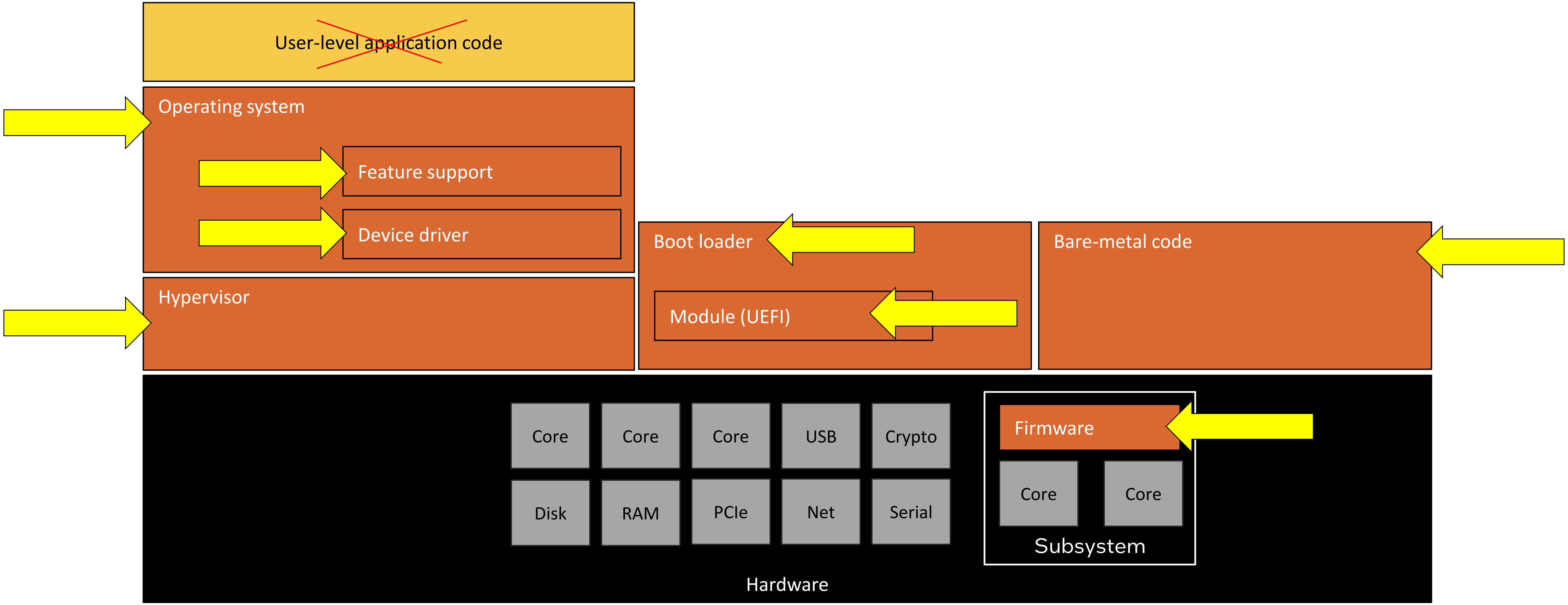


Fuzzer and target run side-by-side on an operating system

- Fuzzer uses host operating-system mechanisms to control and track the target
- Application compiled with instrumentation, coverage, and sanitizers to provide feedback
- On Linux, use "`fork`" to save an initial state to return to for each test

Works well for user-level software

Definition: “Hard-To-Get-At Software”



Hard-To-Get-At Software and Fuzzing

No help from an operating system

- Where do you run the fuzzing tool?
- How do you get inputs into the target from the fuzzing tool?
- How do you detect a failure when there is no OS underneath to help?

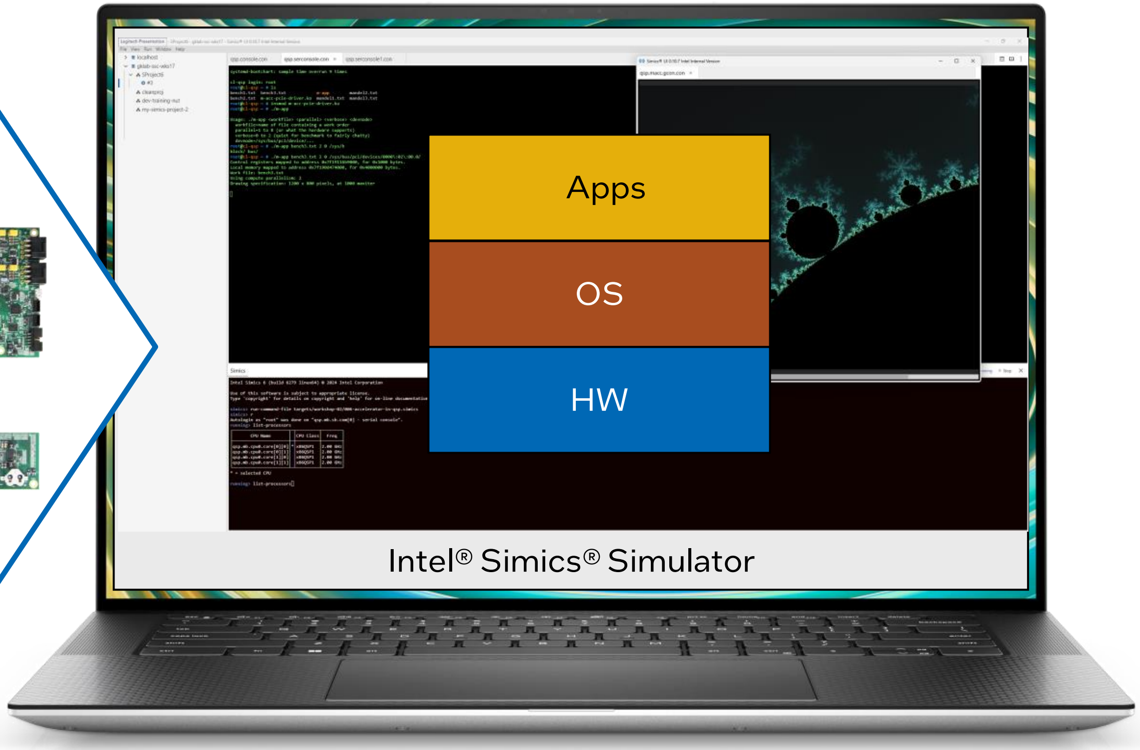
Code insight (coverage) is more difficult to get

- Compilation with instrumentation might be hard (code size, data size, I/O)
- Code runs in protected memory, …

Resetting target state is tricky

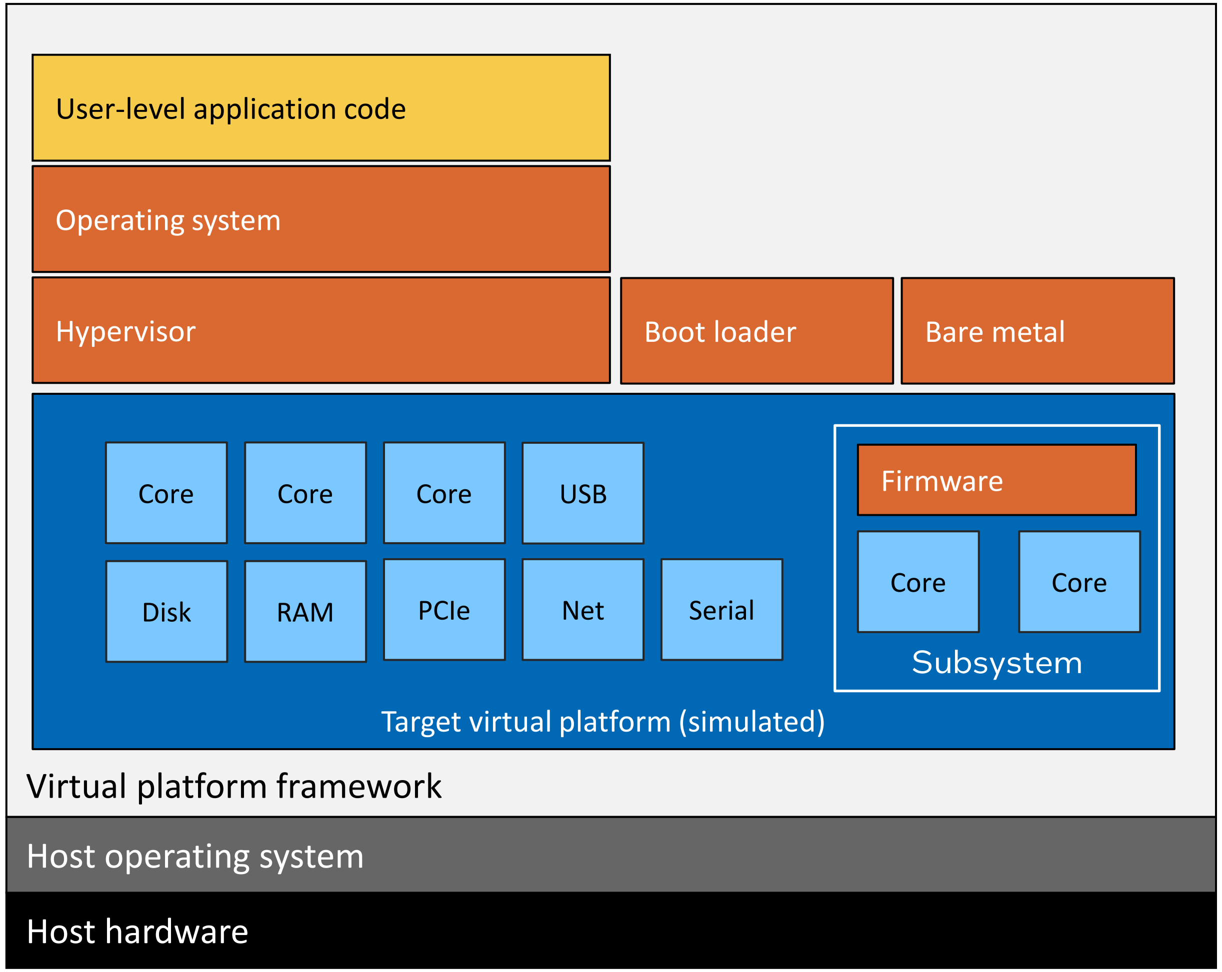
- Code works directly with the hardware – how to rewind hardware registers?
- Operating-system forking is not available

Solution: Virtual Platforms



Run your software without the hardware – on a software model

Virtual Platforms? Why and What?



Technology

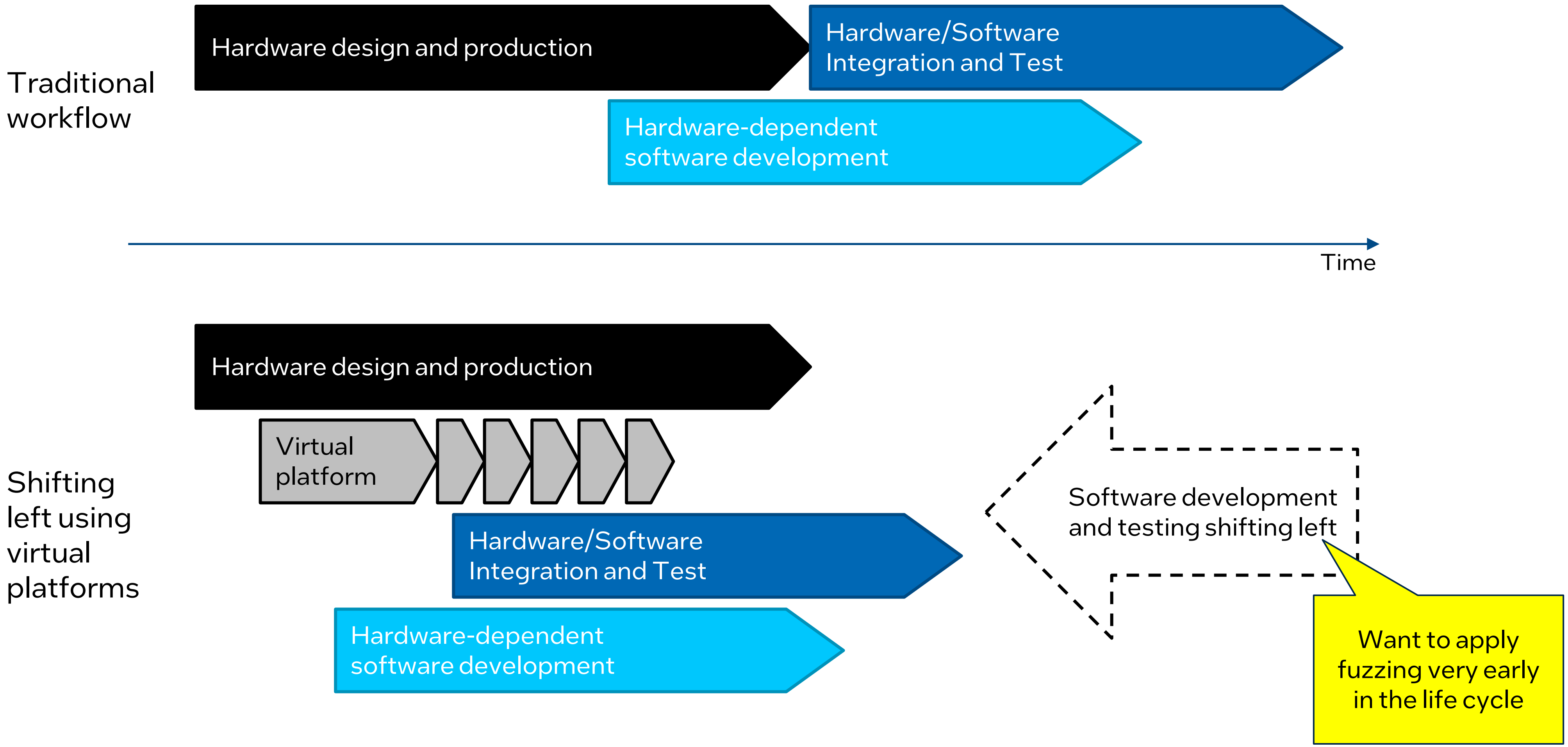
- Software model of hardware
- Run **the same software** as the hardware
  - Same builds, same binaries
- Fast enough to run real workloads

Use case examples

- Explore system architecture
- Develop software early
- Continuous integration of software and hardware
- Debug and test software

Fuzzing is a test technology

Note: Shift-Left = Virtual Platform



Why do Fuzzing on a Virtual Platform?

Fuzz hardware-dependent code

- Fuzz code that interacts closely with hardware
- VP = possible to roll back disk and peripheral device state

Fuzz code with limited interfaces

- Fuzz code that is hard to interface with on real hardware
- VP = access to the platform internals

Shift-left software quality

- Fuzzing increases quality
- Software can run on VP in pre-silicon, why wait for hardware?

Richer fuzzing environment

- VP can observe more types of failures than hardware
- VPs can inject hardware stimuli to provoke software

If you have a VP anyway

- Additional value from existing investment in model
- Avoid constructing complicated setups based on a standard VMs

Alternatives to Virtual Platforms?

Port hard-to-get-at code to run as user-level program

- Requires stubbing/faking/simulating hardware interface
- Not the same code
- Not the same compiler

Use a standard virtual machine (VM)

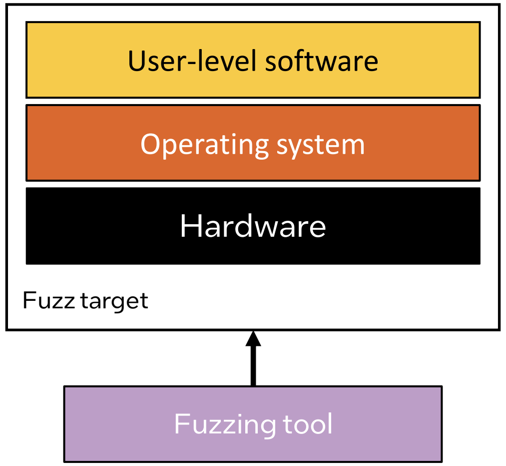
- Modify aspects of the target software to make it work
- Observe execution of code at OS level
- Compile to the VM = compile for the host architecture
- Does not provide the actual target hardware

Use generic instruction-set simulators

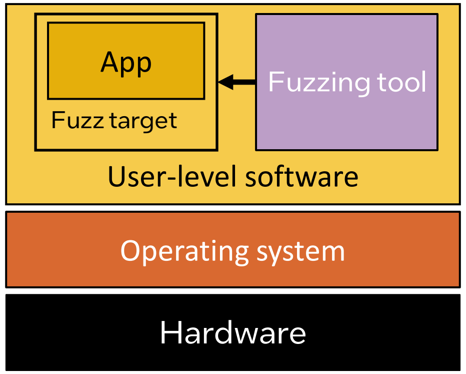
- Fudge the interface to peripherals
- Various generation schemes have been proposed for the peripheral/hardware side

Much easier to just use a virtual platform of the hardware

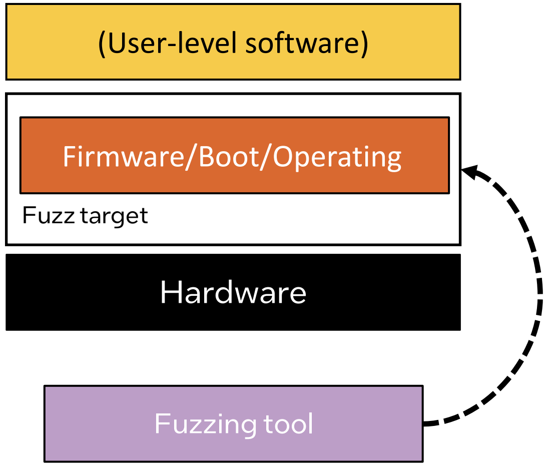
Typical Fuzzing Setups and Virtual Platforms



Black-box fuzzing at target system boundary



Fuzzing individual applications running inside a standard software stack



Fuzzing difficult-to-get-at code that runs directly on the hardware

- VP use: replace the physical hardware with virtual hardware
- Same input/output, standard real-world connections suffice

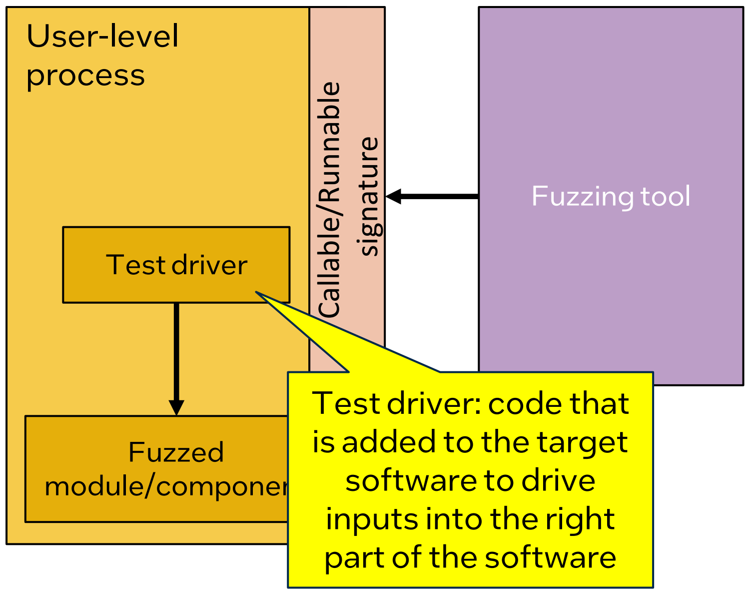
- Easy to do with standard tools
- VP use: when user-level software uses new hardware (instruction sets etc.) – run on VP

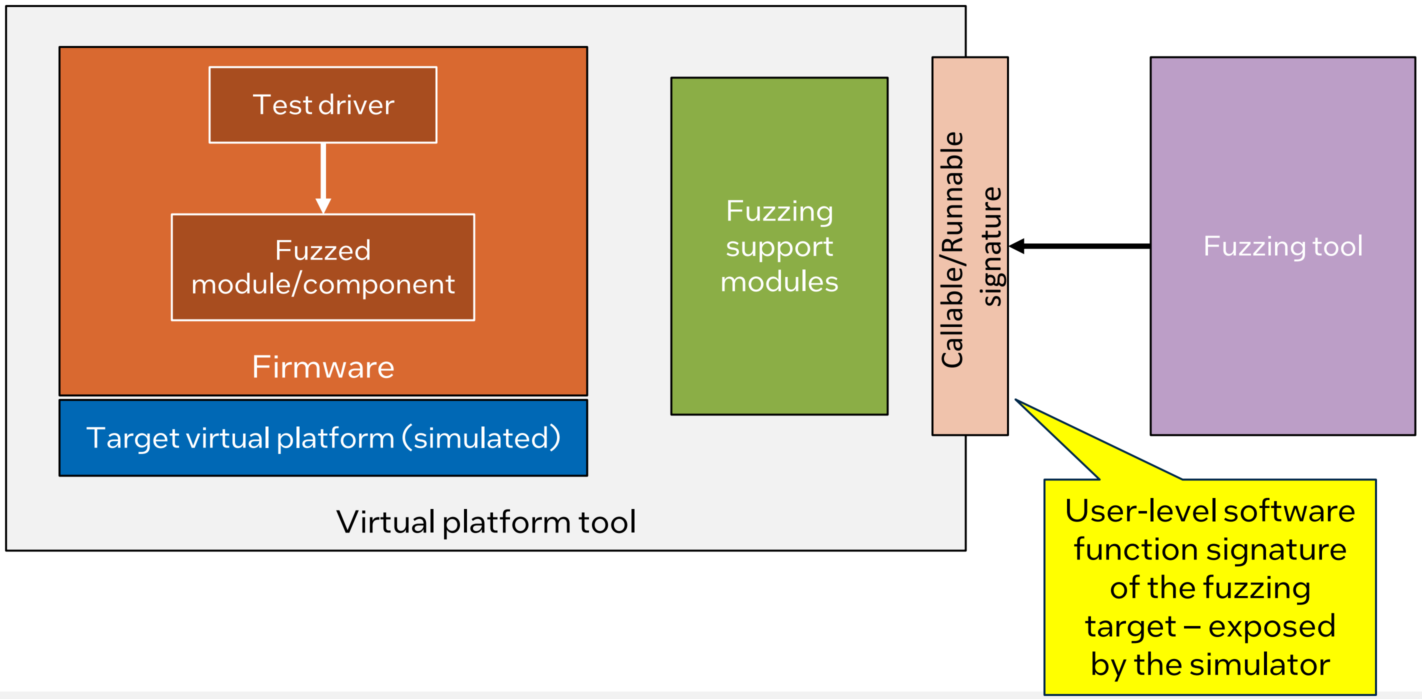
- Not feasible with standard tools
- Requires support in the VP to interface the fuzzer and the software
- Focus of this presentation

Virtual-Platform-Based Guided Fuzzing

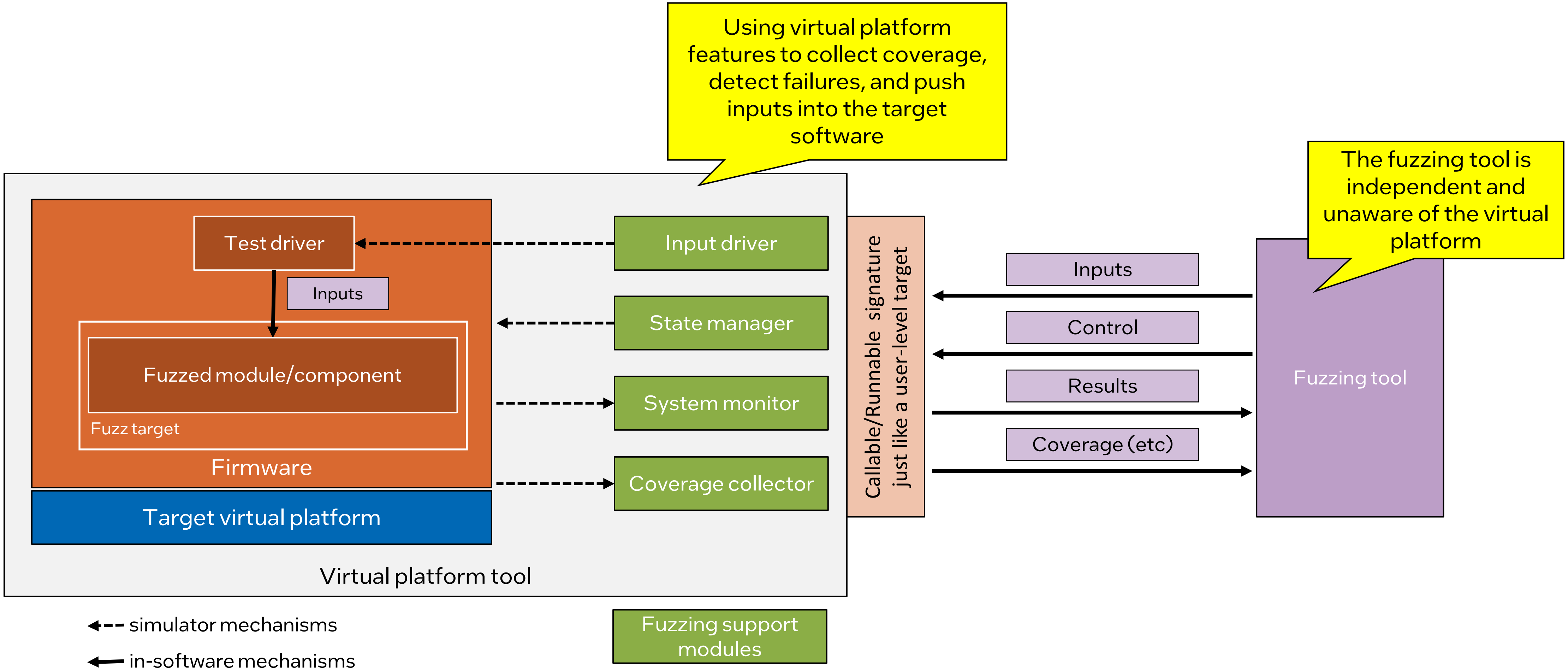
Concept: Make the virtual platform look like a user-level program

- Reuse existing fuzzers and their fuzzing logic as-is…
- … while facilitating access to the software using virtual-platform techniques





Virtual-Platform-Based Guided Fuzzing: Details



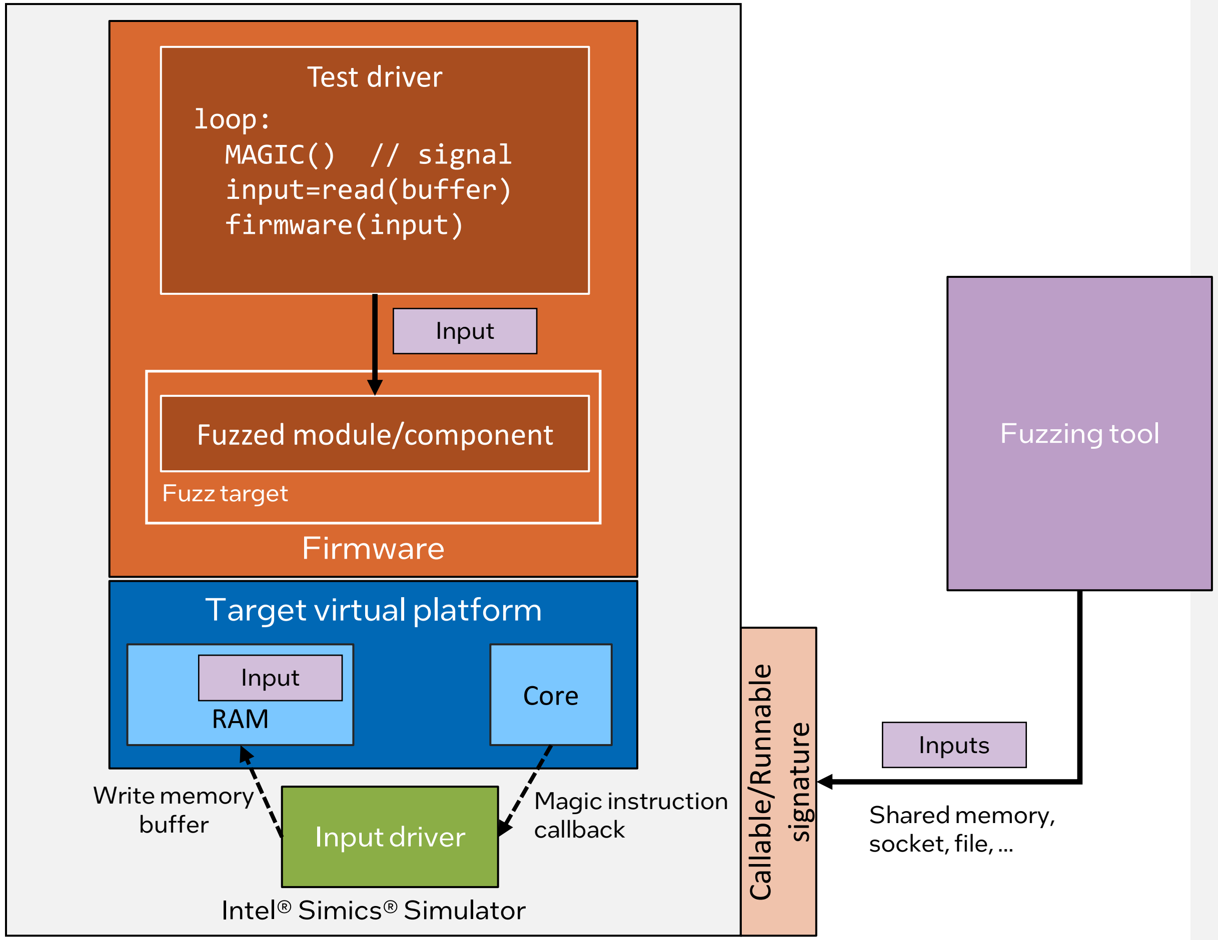
Test Driver and Input Driver

Test driver (target software)

- Depends on target and fuzzing setup
  - Knows how to call into/activate the target
  - Knows how to apply inputs from fuzzer
- **Magic instructions:** key VP trick
  - Test driver issues magic instruction when ready to receive data
  - Input driver catches the magic callback and fills in next test case
- *(Adding to target software stack is the only robust solution; calling into software from VP directly is difficult and brittle)*

Input driver (simulator module)

- Implements the interface towards the fuzzing tool – depends on how the simulator and fuzzer communicate
- Passes data from the fuzzer to the test driver software – dumb pipe



System Monitor

Wait for conditions that indicate success or failure in the system under fuzz

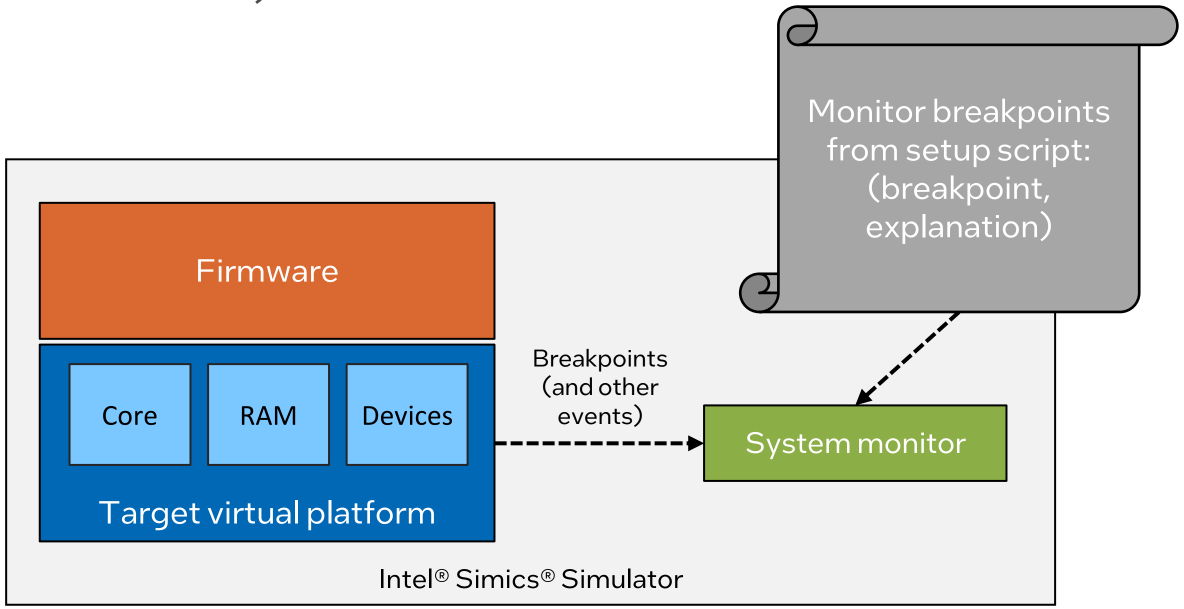
- Watch using breakpoints
- Software is not modified or instrumented
- Success is usually "called function returns OK"

Example conditions:

- Running code outside of allowed ranges
- Memory accesses outside of allowed ranges
- Executing undefined instructions
- Triggering interrupts
- Processor resets and triple faults
- … whatever makes sense …

The set of events to watch depends on the system and software

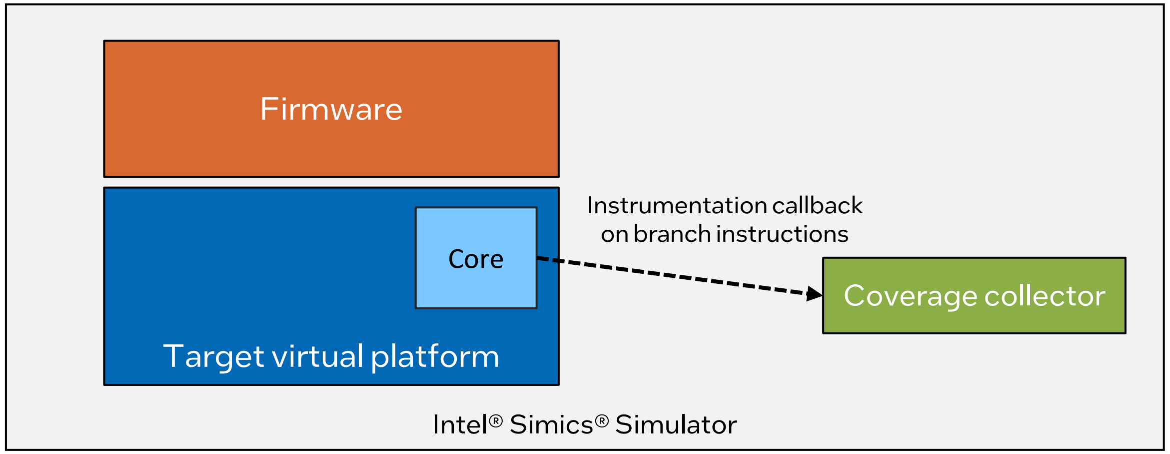
- (It is easier for user-level fuzzing where signals/segmentation faults & sanitizer errors cover most)



Coverage Collection

Current solution: **Branch (edge) coverage**

- The virtual platform processor core simulator reports all branch instructions to a coverage tool
- Coverage data looks like it came from code instrumentation
- Hashing-based algorithms
  - Record all branch instructions
  - Address hashing = no need to know where the code of interest is in memory



This is **"grey-box" fuzzing:** coverage measurement without source code

- Watch the code execution "from the hardware"
- No source code needed
- No compiled-in instrumentation
- ... but still looking at the code flow

Technicality: *note that compiling-in the test driver makes that part of the process "open-box", so the overall approach is not fully grey-box*

State Management: In-Memory Snapshots

Recall, fuzzing = many short test runs starting from a reused common starting point

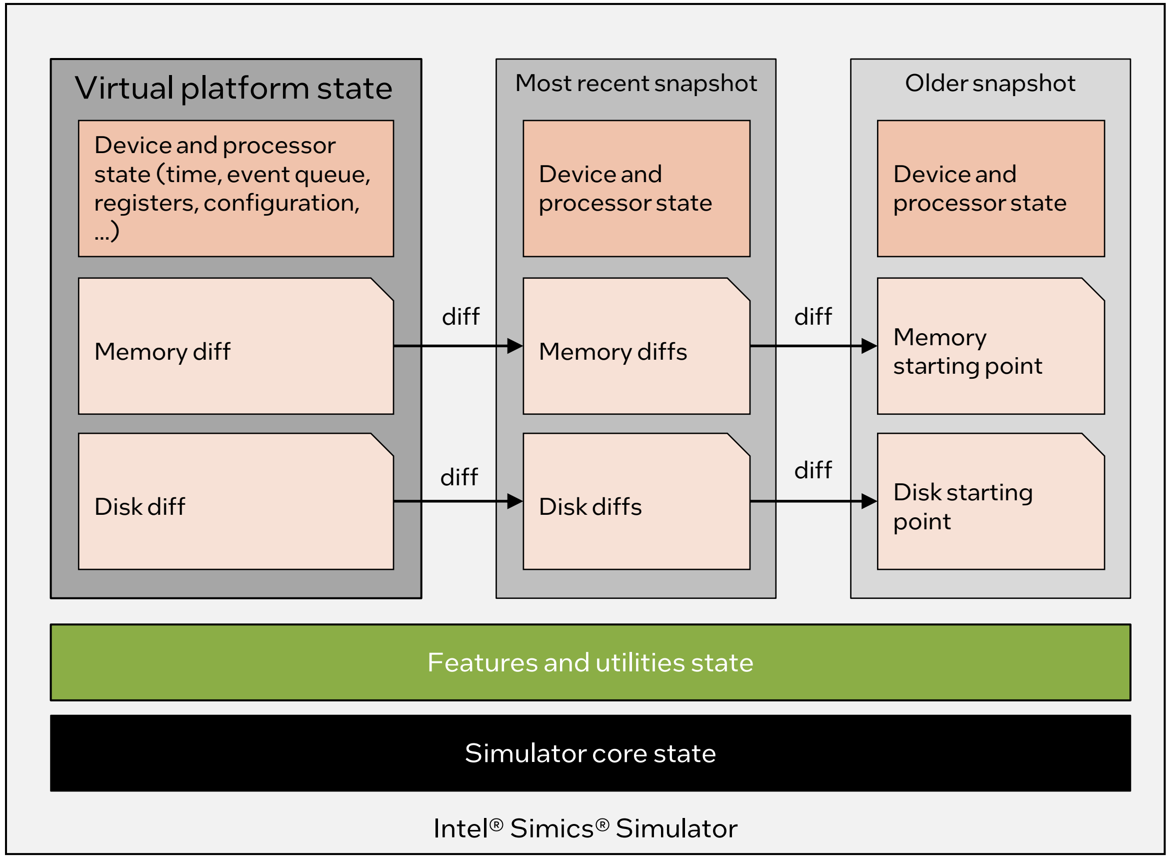
- The system state restore is critical
- Virtual platform = all hardware state is under control

Simulation state is restored using simulator **in-memory snapshots**

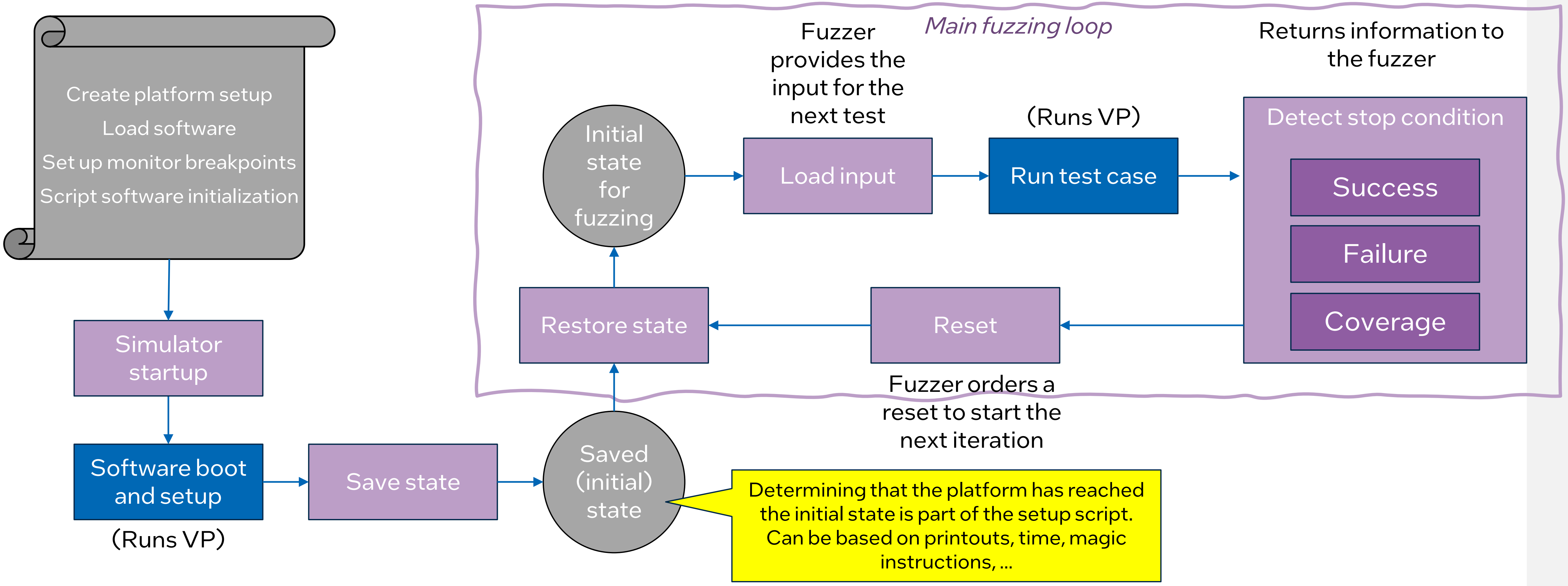
- Processor and device state
- Memory and disk contents
- Not including the simulator core and features

Why not fork the VP process?

- Linux fork does not work well with a threaded simulator
- What to do on Windows hosts?



Complete Fuzzing Flow using a Virtual Platform



Experience

We have applied virtual-platform-based fuzzing internally at Intel

- Cannot talk about concrete application due to sensitivity
- But it has worked

The TSFFS fuzzing setup, https://github.com/intel/tsffs/, is an open-source virtual-platform-based fuzzer for the Intel® Simics® Simulator

- Reports about 200 iterations per second for small virtual platforms

Questions?

Get the Intel® Simics® Simulator
https://developer.intel.com/simics-simulator

Try the TSFFS fuzzing setup (close to what was presented here)
https://github.com/intel/tsffs/

Legal Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete http://www.intel.com/performance.