2023
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
10 YEAR ANNIVERSARY

Fuzzing Firmware Running on Intel® Simics® Virtual Platforms

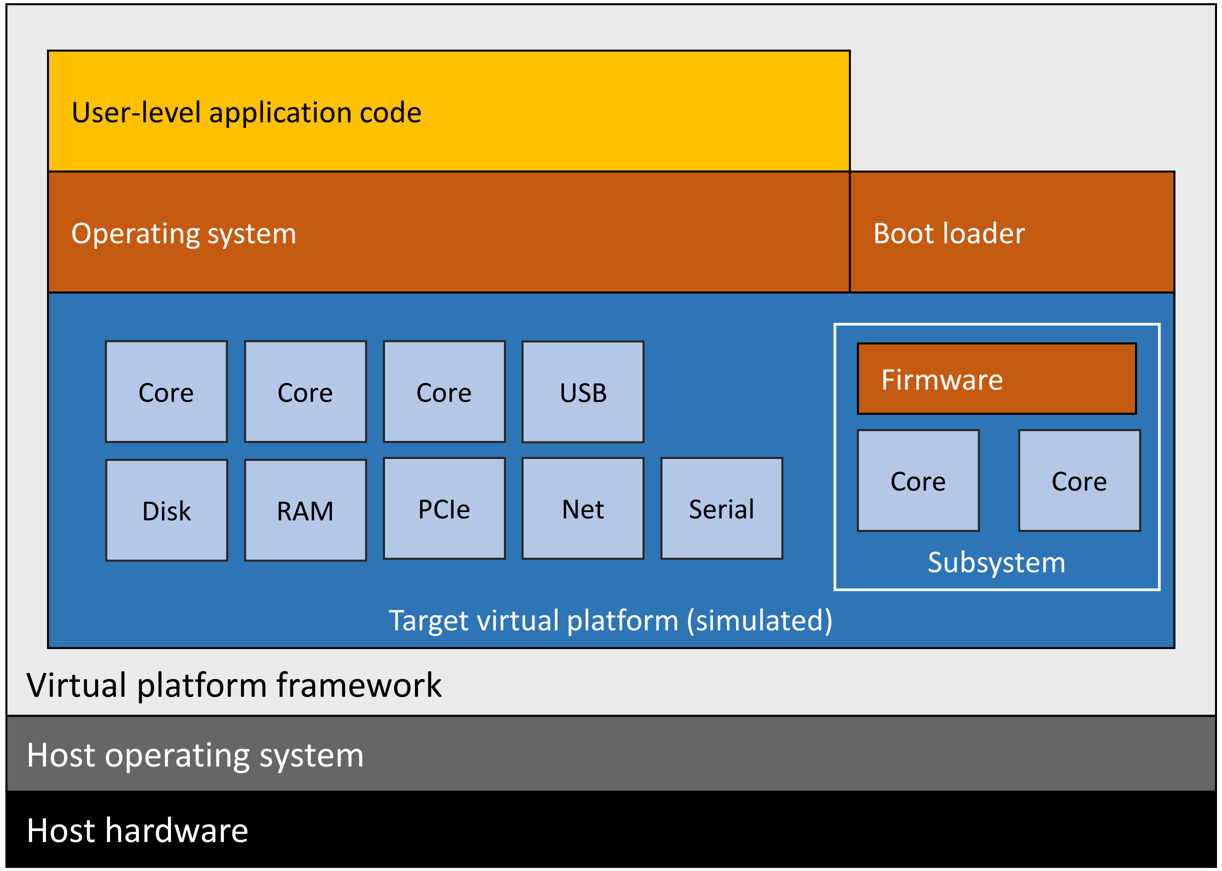
Jakob Engblom, Intel, Sweden

Robert Guenzel, Intel, Germany

intel®

accellera
SYSTEMS INITIATIVE

Virtual Platforms? Why and What?



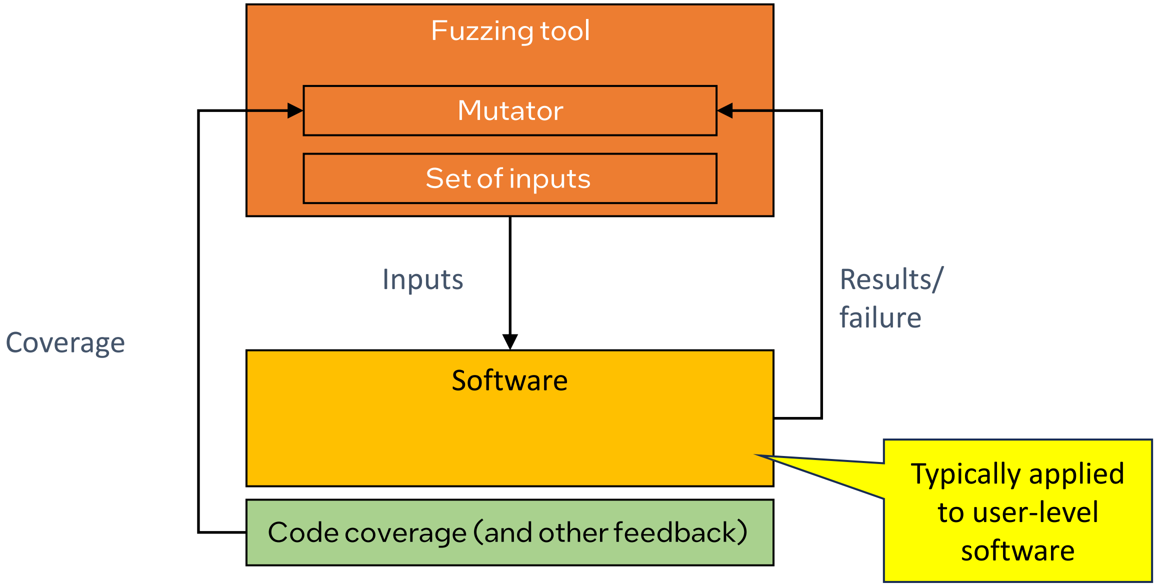
- Technology
  - Software model of hardware
  - Run **the same software** as the hardware
  - In our case, fast transaction-based models (TLM)
- Use case examples
  - Explore system architecture
  - Develop software early
  - Continuous integration of software and hardware
  - Debug and test software

Fuzzing is a test technology

Fuzzing? Why and What?

- Send “random” inputs to software
  - Tests are typically small unit tests
  - Observe software behavior – usually test failures or crashes
  - “Random”: A set (corpus) of inputs is mutated according to various rules
- Coverage-guided fuzzing
  - Improve fuzzing effectiveness
  - Discern tested code
  - Guide mutation rules towards exploring new code/paths
  - Can be done without source code!

- Find more errors than manually written fixed tests
  - Explore corner cases that developers did not think about



Note: Fuzzing Techniques and Tools

- **Huge and active** research field!
  - Constant flow of new concepts, heuristics, algorithms, …
- Our goal: **enable standard fuzzer tools** to be used with a virtual platform
  - Reuse the fuzzer logic as-is
  - Provide the execution platform
  - In practice, done per-fuzzer
- Levels of insight into target
  - Black-box
    - Opaque target
  - Grey-box
    - Some feedback data
    - No source code
  - White-box
    - Source code used

Why do Fuzzing on a Virtual Platform?

Shift-left software quality

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

Fuzzing system-level code

- Possible to fuzz code that interacts closely with hardware
- VP can roll back disk and peripheral device state
- Key enabler: determinism, even for multicore targets

Fuzzing "hidden" code

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

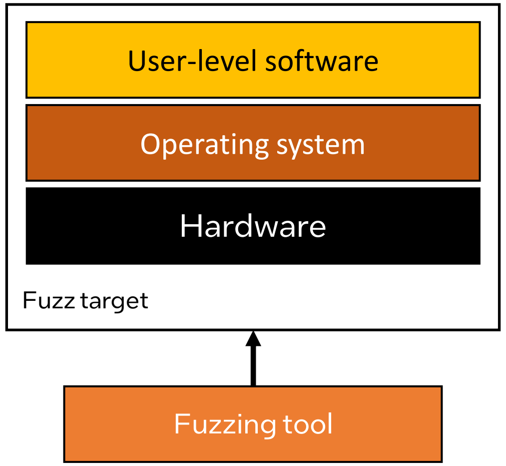
Richer fuzzing environment

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

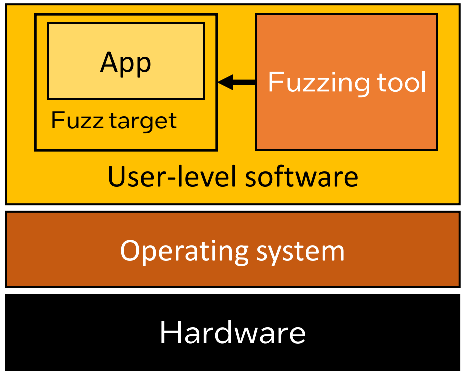
The VP exists anyway

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

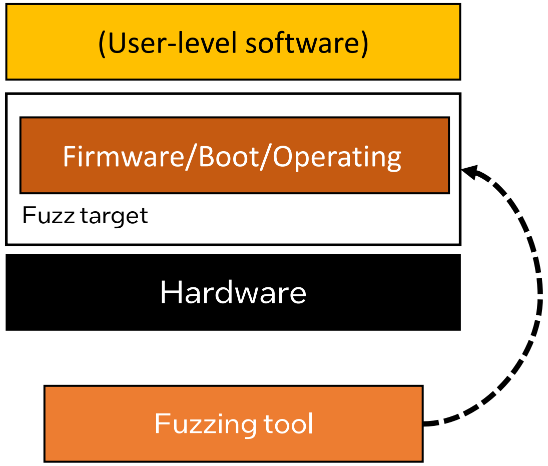
Typical Fuzzing Setups



Black-box fuzzing at target system boundary



Fuzzing individual applications running inside a standard software stack



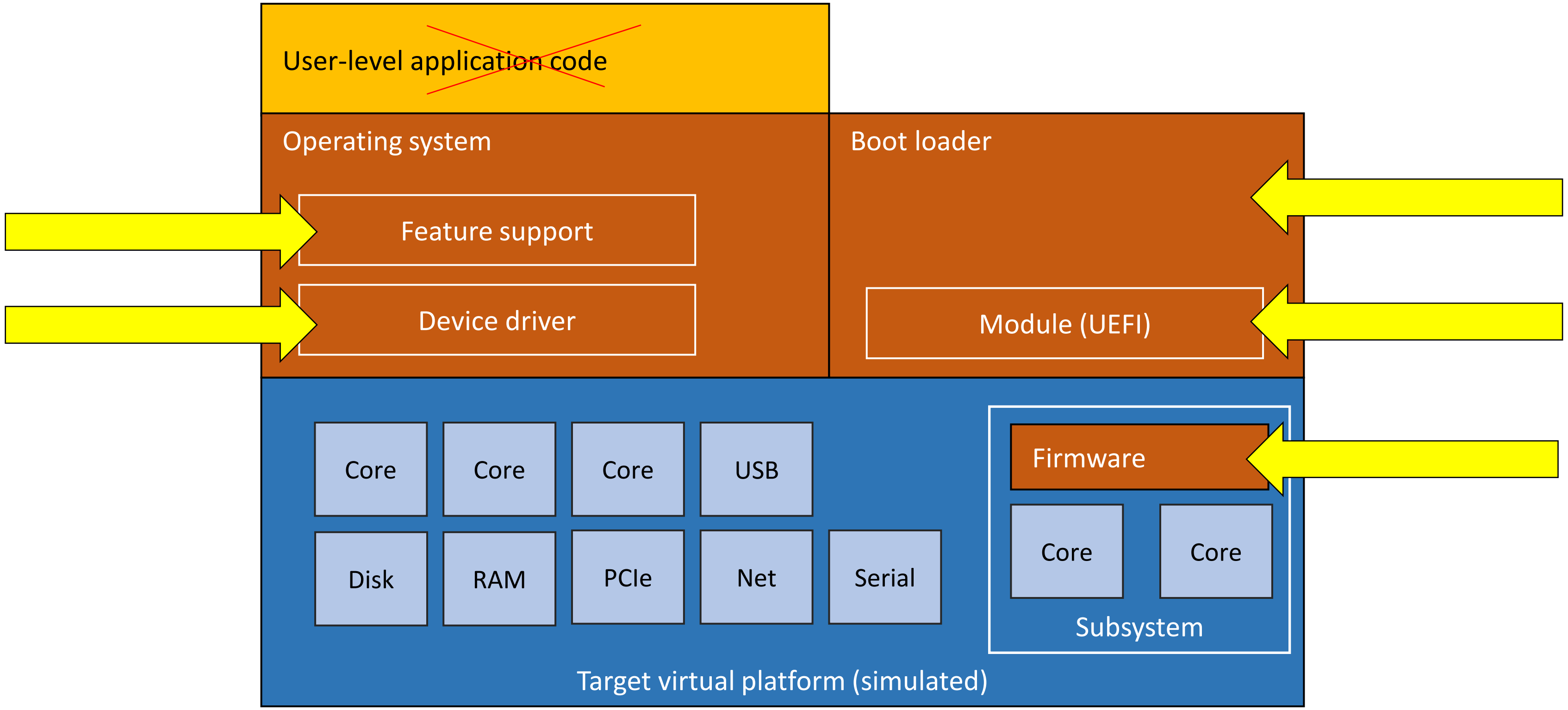
Fuzzing low-level code, firmware, drivers, that run directly on the hardware

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

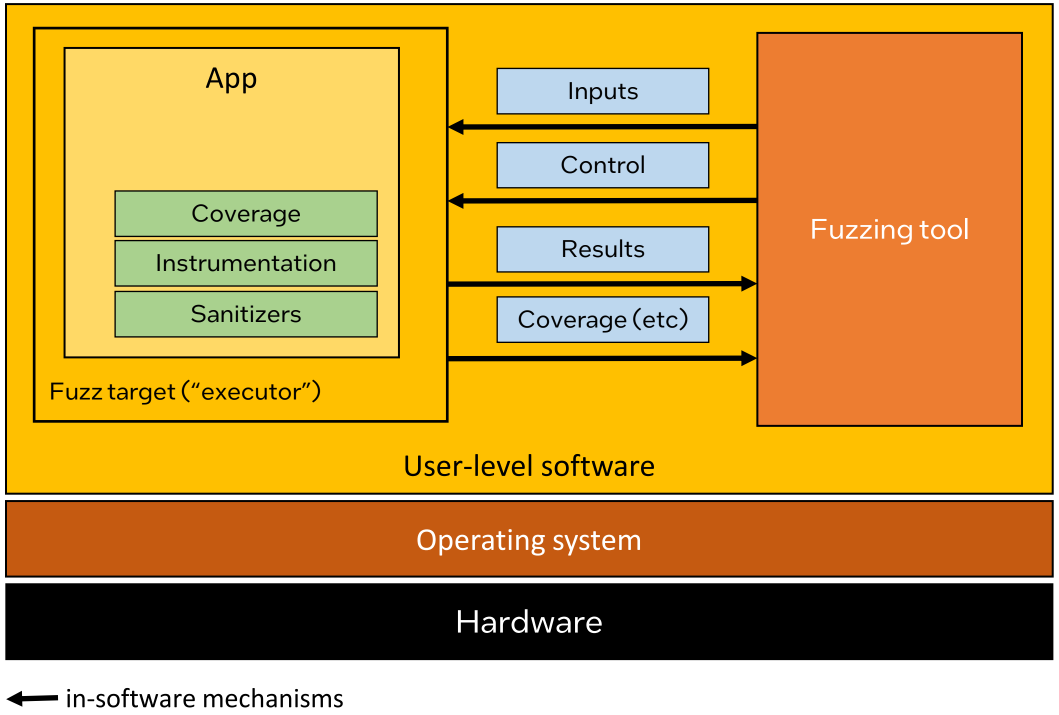
- Easy to do with standard tools
- VP use: when user-level software uses new hardware (instruction sets etc.) – run on VP

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

Types of Software under Consideration



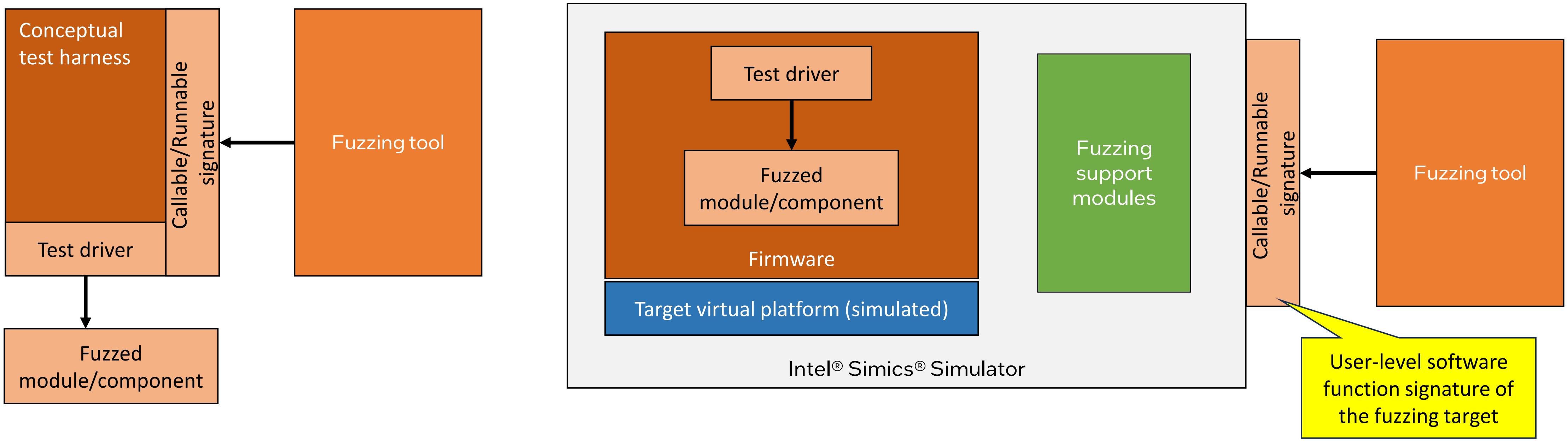
Standard User-Level Guided Fuzzing



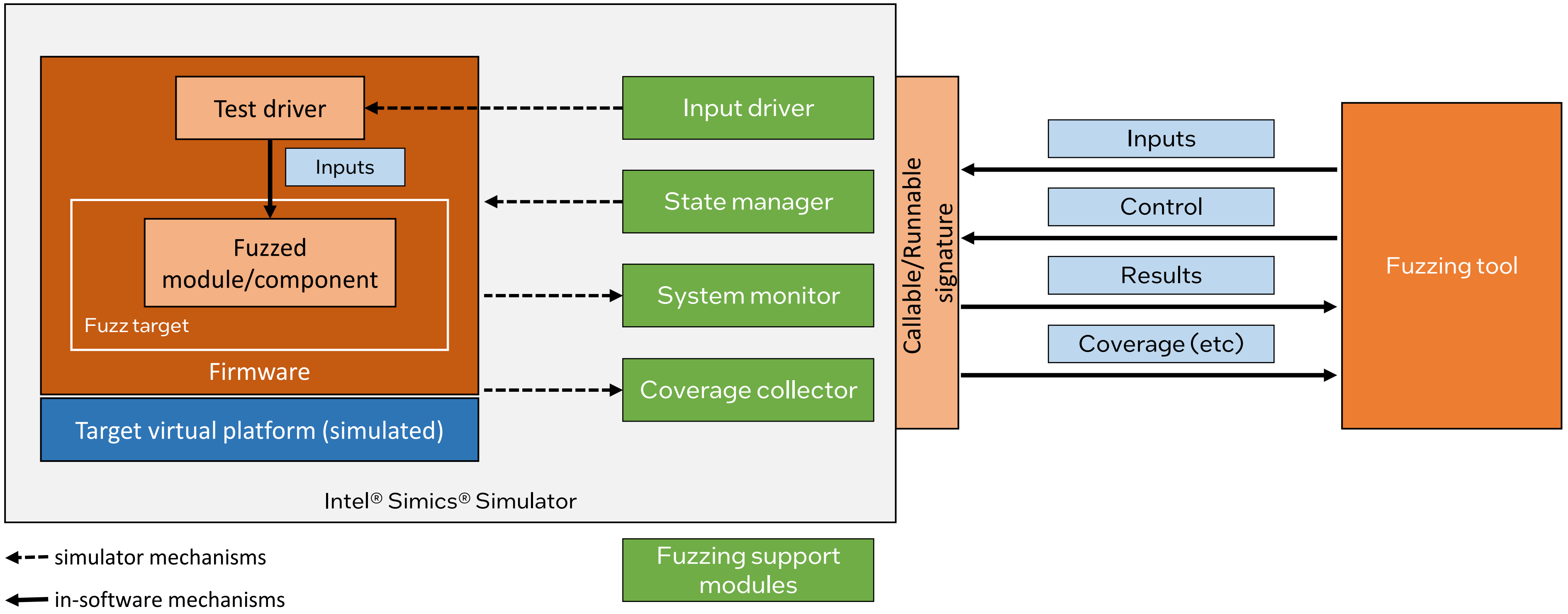
- Fuzzer and fuzzing target run side-by-side on the host
  - Fuzzer uses host operating-system mechanisms to control and track the target
  - Application compiled with instrumentation, coverage, and sanitizers to provide feedback
  - On Linux, use "fork" to quickly rewind fuzz target state

Virtual-Platform-Based Guided Fuzzing

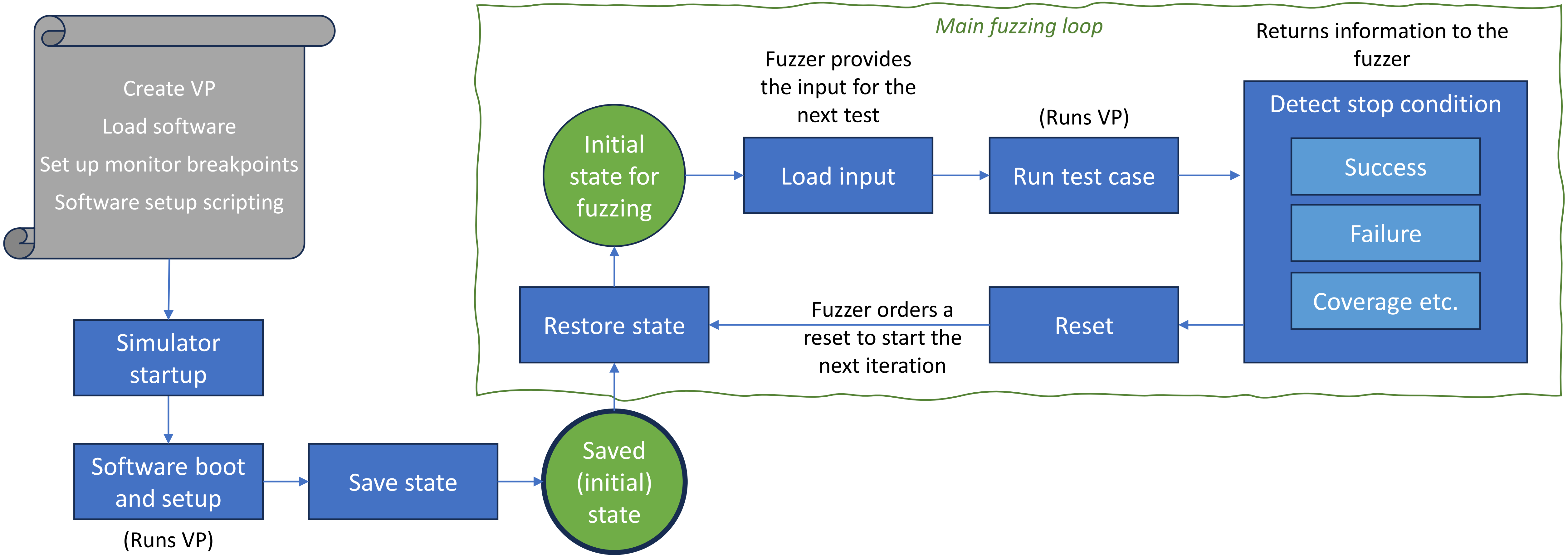
- Concept: Make the virtual platform look like a user-level program
  - Reuse existing fuzzers and their fuzzing logic as-is…
  - … while facilitating access to the firmware using virtual-platform techniques



Virtual-Platform-Based Guided Fuzzing: Details



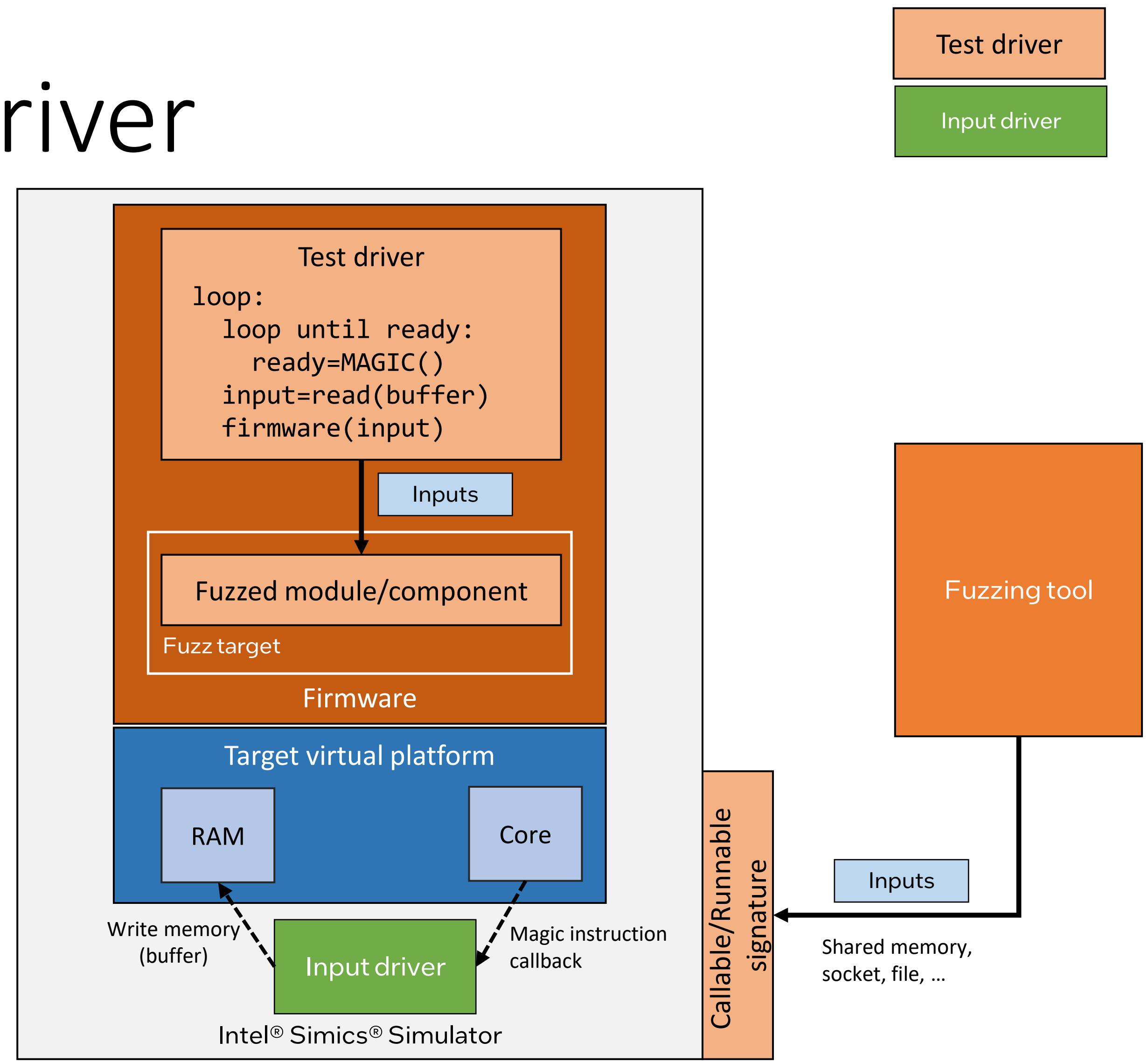
Fuzzing Flow using a Virtual Platform



Test Driver and Input Driver

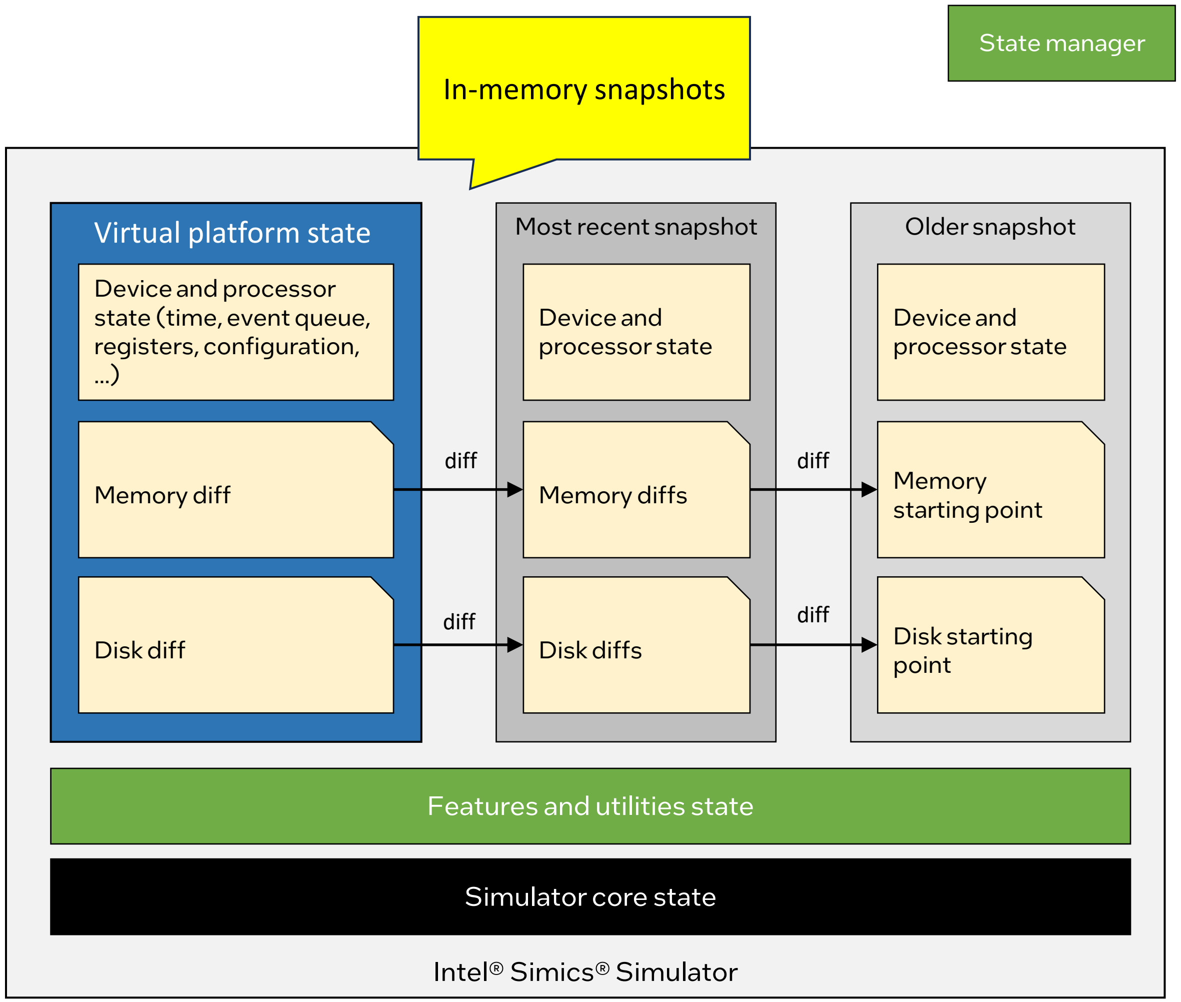
- Test driver (target software)
  - *(Calling into software from VP directly is difficult and brittle)*
  - Depends on target and fuzzing setup
    - Knows how to apply inputs from fuzzer
    - Knows how to call into/activate the target
  - Polling loop, using **magic instructions** to talk to the **input driver**

- 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



State Manager

- State manager functions:
  - Reset the state of the fuzzing target
  - Start the simulation
- Simulation state is restored using **in-memory snapshots**
  - Device state
  - Memory and disk contents
  - Excepting tools and simulator core
  - (Standard Intel® Simics® Simulator feature)
  - Critical for performance
    - Minimize the virtual platform size
    - Optimize the framework (WIP)
  - Why not use forking?
    - Linux fork does not work well with a threaded simulator

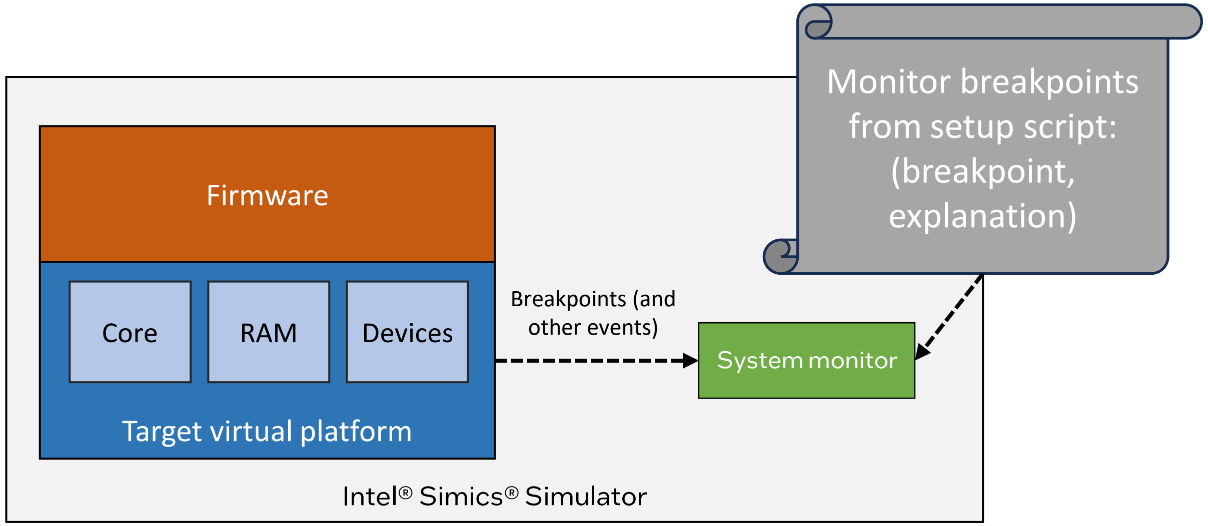


System Monitor

System monitor

- Wait for conditions that designate errors in the system under fuzz
  - Use VP mechanisms to watch the execution – typically breakpoints
  - Not visible to the software
- Specific conditions are set up by the setup script
  - List of conditions + messages
  - Passed to the system monitor
- If a condition is hit:
  - Stop the current run
  - Return to fuzzer, with message

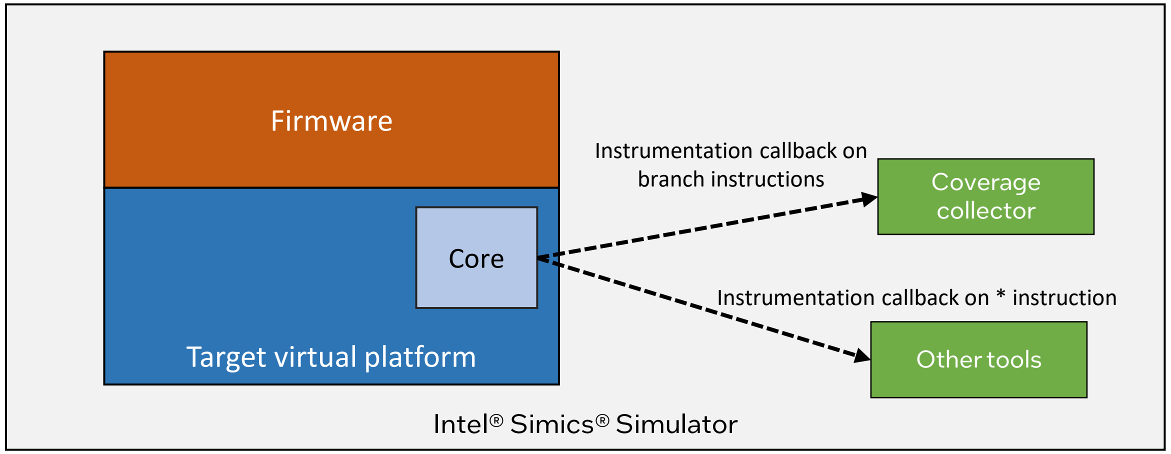
- Example conditions:
  - Running outside of allowed memory
  - Accesses outside of allowed memory
  - Undefined instructions
  - Processor resets and triple faults
  - … whatever makes sense …



Coverage Collector

Coverage collector

- Coverage is key to guided fuzzing
  - Reflect how well the test cases explore the behavior of the code
- Current solution: **Branch (edge) coverage**
  - Coverage data looks like it came from code instrumentation compiled into user code
  - Cover all executed code
    - Hashing approach = unlimited reach
    - Test driver code is small and does not hurt
  - Using the standard Intel® Simics® Simulator instrumentation API to get reports about all branches
- “Grey-box” fuzzing
  - No source code needed
  - No compiled-in instrumentation
  - … but still looking at the code flow



- Branch coverage details
  - Get the address branched to
  - Combine with the previous destination (i.e., current basic block)
  - Hash the result and increment counter
- Reusable and generic as long as the data produced makes sense to the fuzzer

Portability, Summary

Component	Firmware dependent?	Target dependent?
Test driver (target software)	Yes	Yes (registers, magic, …)
Input driver (simulator module)	No (test driver has to adapt)	No
State manager (simulator module)	No	No
System monitor (simulator module)	No	No
System monitor configuration (script)	Yes (reflect error conditions)	Yes
Coverage collector (simulator module)	No	Yes (ID branch instructions)

Note that porting to a new fuzzer tool will likely require updates to all the modules

Possible Future Extensions

- Fuzzing with hardware inputs
  - Current effort mostly using libAFL = built for software fuzzing
  - VP-side this is not very hard
- Adding sanitizers into firmware
  - Compilers generally support it
  - Requires a custom output library (have seen that done)

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.
-
-

The End