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

Fuzzing Firmware Running on Intel® Simics® Virtual Platforms

Jakob Engblom, Intel, Sweden (*jakob.engblom@intel.com*)

Robert Guenzel, Intel, Germany (*robert.guenzel@intel.com*)

***Abstract*— Fuzz testing ("fuzzing") is a widely-used technique used to discover crashes and security vulnerabilities in software. Fuzzing explores the input space to uncover inputs that can cause the system to fail or provide an entry point for an attacker. Fuzzing is well-established and easy to deploy for typical user-level applications, with a wide variety of ready-to-use tools available [1][2]. However, fuzzing other code, such as bootloaders, device firmware, embedded microcontroller software, and operating system drivers, is more challenging. In this paper, we present how fuzzing of firmware and other low-level code running on Intel® Simics® virtual platforms can be performed by connecting the simulator to standard fuzzing tools. The approach does not require modifications to the tool itself but rather builds on top of standard APIs and features.**

Keywords—testing, fuzzing, firmware, virtual platforms, simulation

I. INTRODUCTION

A virtual platform (VP) is a program that simulates the processor cores and other functionality of a hardware platform [3]. The goal of a VP is to run the unmodified software stack that runs on the modeled hardware. The software can be anything from a full software stack, including hypervisors and operating systems, to a small microcontroller binary. One of the primary use cases for VPs is software development, testing, and integration. Fuzzing is a popular software testing technique, and fuzzing using the VP makes it possible to fuzz firmware and software for hardware that is hard to access or does not yet exist.

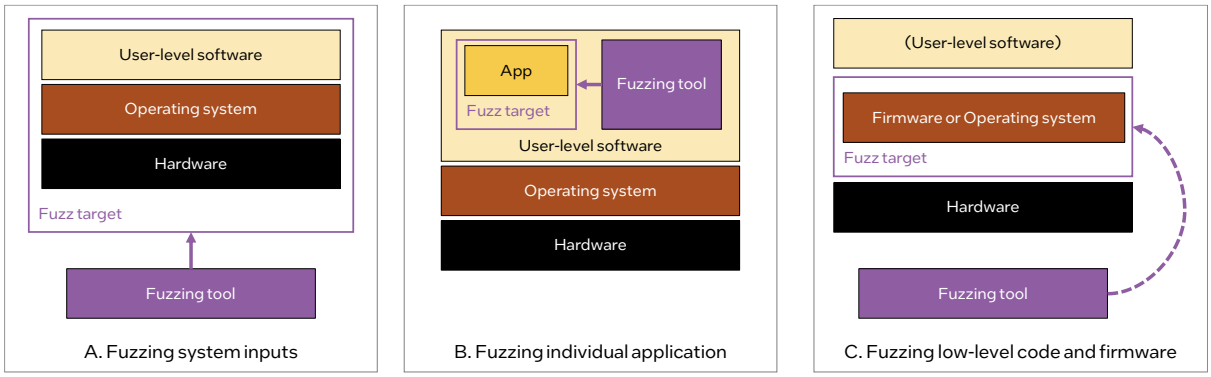

Figure 1

Fuzzing can be done at different levels and with different entry points to the system.

Figure 1A shows *system-level fuzzing*. The entire system is subjected to inputs over an external connection such as Ethernet or USB. Such fuzzing works the same on hardware and VP and requires nothing more than a real-world connection to the VP. The VP is used as a replacement for hardware, and no use is made of VP-specific features.

Figure 1B shows *application-level fuzzing*, the most common way to do fuzzing on hardware. The same setup can run in the same way on a VP. The VP provides an execution platform, but VP features are not used as the process is all internal to the software stack. This kind of setup is useful for cases where the application under test depends on hardware features that are not yet available in hardware, even though it is not low-level software or drivers.

Figure 1C shows *firmware fuzzing*, which is the primary use case for VPs in fuzzing, in our opinion. The VP makes it possible to fuzz code that interacts directly with hardware or that runs without the support of an operating system. Such code is difficult to fuzz using standard tools in user space as they are not simple applications.

A. *Virtual Platforms or Virtual Machines?*

Various solutions have been built over the years that use complicated workarounds to run firmware-style code on top of general simulation or virtual machine solutions [4][5][6]. Using a virtual platform is technically and conceptually simpler. In our world, VPs are always developed for new silicon platforms, and fuzzing is just another way to use them. Using a VP makes it possible to fuzz software relying on future instruction sets and hardware features, which is, in general, not possible using virtual machines (VM) that essentially provide clones of the underlying hardware. It also avoids introducing additional models or tools into the pre-silicon software workflow.

Using a VP for fuzzing has some distinct advantages over general-purpose VMs regarding getting access to low-level code and the execution state. The VP can be extended to provide additional communication channels that reach into the hardware/software system. The VP also has better observability since it can watch even the internal behavior of hardware models and the transactions between different models. With a VP, it is easy to fuzz code that runs at "below ring 0" on a primary processor, such as UEFI System Management Mode handlers [7].

The VP state encompasses both the **target software and hardware state as a unit**. This means that disks, timers, accelerators, and even networks can be part of the fuzzed system. Resetting the state of the VP for a new fuzzing input rolls back both hardware and software state to a known point, unlike VMs based on real hardware where the state of the hardware is mostly outside the control of the VM [6]. The system under test can include multiple VPs in a simulated network running inside a single Intel® Simics® simulator process [8].

The **determinism** inherent in a well-written VP means that interrupt arrival times, completion flag register changes, and other events asynchronous to the software are still synchronized with the software execution. The same fuzzed inputs passed to the same initial state should result in the same results and sequence of events. This allows the VP to handle **multiple processor cores**, as well as interactions across different firmware-executing subsystems. The determinism also makes it easy to analyze the failures since it is easy to replay a failing scenario with a debugger attached or using a software debugger built into the VP.

II. FIRMWARE FUZZING ON A VP USING LIBAFL

While there are many fuzzing tools available [1], most of the fuzzing that has been done with the Intel Simics simulator has used libAFL [2]. Thus, we use that as the model fuzzer for the remainder of this paper. The structure of a standard fuzzing setup used with libAFL is shown in Figure 2A. Figure 2B shows the Intel Simics simulator-based setup. The fuzzing uses the simulator as the *executor*, using the same interface as other fuzzing executors. The fuzzing tool is not modified, and its logic is used as-is.

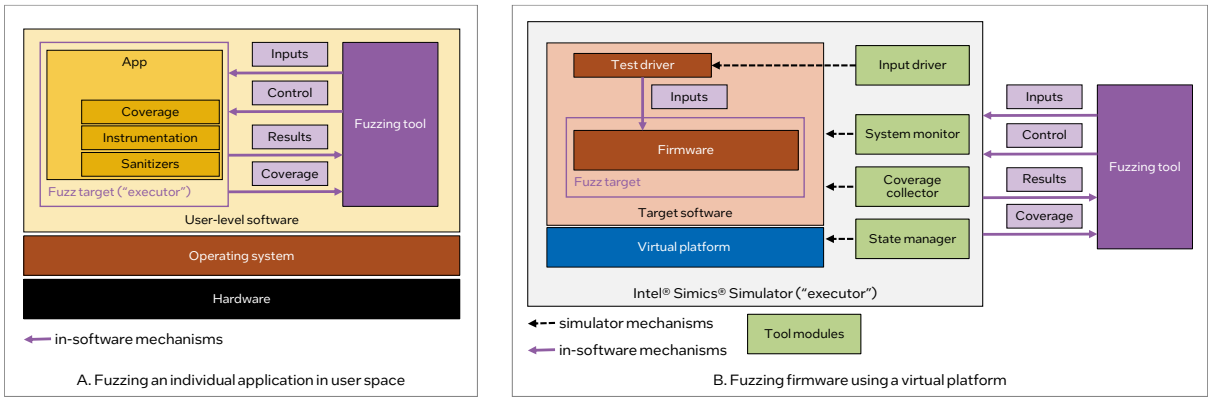

Figure 2

The VP fuzzing solution is built from a set of modules:

A **test driver** (or harness) stub is added to the firmware (software under test). It is used to call into the firmware being fuzzed and report the results of the call. The test driver gets the inputs from the simulator-side **input driver**, using simulator back-doors. The test driver might be elided in case all inputs are provided via hardware interfaces.

The **system monitor** is responsible for detecting test failures and reporting them to the fuzzing tool. The precise set of failure conditions to monitor are application-dependent and defined by the user. The core monitor implementation is application-independent and reusable.

The **coverage collector** collects information on executed code in order to provide feedback to the fuzzer to guide its exploration of the target software behavior. The collector uses simulator inspection features, and the target code does not have to be instrumented or modified. Coverage collectors are typically quite generic and reusable. Since the coverage collector works on the assembly level, it basically does "grey-box" fuzzing.

The **state manager** is responsible for resetting the hardware and software state in the virtual platform between tests. This function is entirely generic and implemented using in-memory checkpoints that encompass the target system state: processors, devices, memories, disks, and operations in progress.

We only cover the pure fuzzing aspects up to the point of recognizing failures. The root-cause analysis and debugging that follows a fuzzing failure is outside the scope of this work.

Note that the Excite solution [7] also builds on the Intel Simics simulator, and the work presented here can be seen as a generalization of that work. Excite was built to specifically deal with UEFI code on an x86-based VP model. Compared to Excite, we use in-memory snapshots instead of forking the simulator. The approach is also generalized to handle non-x86 architectures and any other "below ring 0" firmware.

III. TEST DRIVER AND INPUT DRIVER

The test driver (also known as a test harness) is a piece of code inside the target software stack that calls into the code under test. This component is unavoidable in practice, as generating function calls from the VP itself is impractical. It is much easier to write code that runs as part of the firmware, picks up input data, and then calls into the functions to test. For UEFI, the test driver is typically a UEFI module. The test driver code can also be statically compiled into the target binary – including into a firmware binary, kernel driver, or user-level code that can call into firmware or other low-level code. The precise implementation depends on the context.

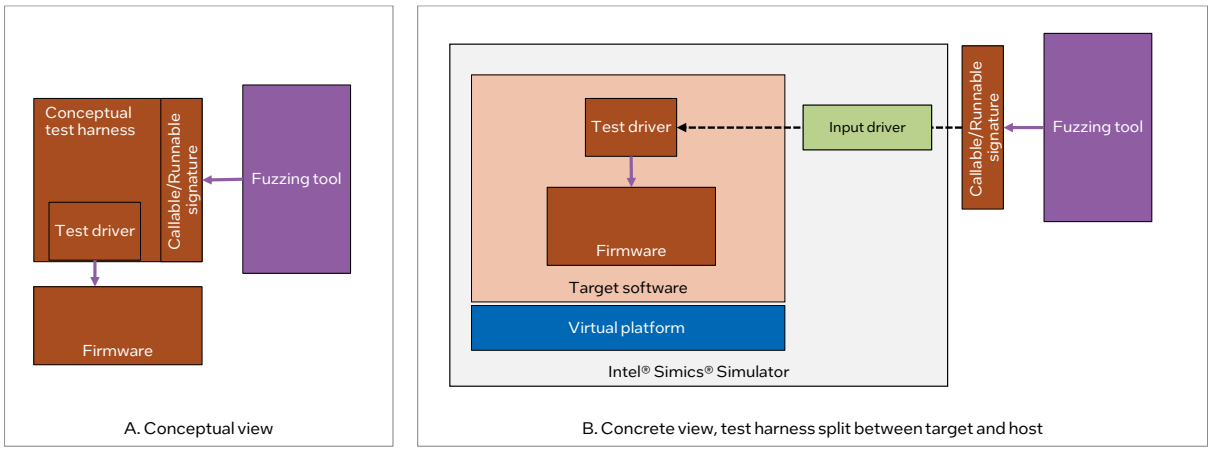

Figure 3

The goal is to make it appear to the fuzzer that it is fuzzing an application on the same host as itself, as illustrated in Figure 3A. This avoids modifying the fuzzing framework itself. In reality, inputs are passed to the test driver inside the target system via the VP, as shown in Figure 3B. The proxy exposed to the fuzzing tool has the same callable and runnable signature as a user-level application being fuzzed.

The data being exchanged between the fuzzer and the firmware is highly application-dependent, and the VP should not need to know anything about its format. Instead, the VP provides general mechanisms and extensible modules that allow the test driver code on the target to talk directly to the fuzzer on the host. Basically, some form of paravirtual or backdoor interface that connects from the simulated world out to the simulator and the real world and that does not interpret the data.

In the case of Intel Simics simulations, the low-level primitive of choice is the "magic instruction." This is an instruction that is a no-op on real hardware but which is detected by the simulator, allowing simulator-side code to run in the middle of a target instruction. Using the magic instruction, we have built mechanisms that allow the efficient exchange of data buffers between the test driver and input driver –the target software (test driver) sets up a buffer in memory and activates the input driver with a magic instruction. The input driver will then write data to the buffer or read data out, depending on the operation being requested. The communication is driven by polling from the target software, which is a perfect fit for fuzzing operations.

In the case that the target is running an OS like Linux, Windows, or UEFI, the Intel Simics simulator offers the "magic pipe," providing a pipe-like data exchange between the host and the target. For deeply embedded firmware, customized implementations using the same mechanism will need to be written and inserted into the firmware.

Finally, the input driver in the simulator needs to talk to the fuzzing tool. There are several options here, such as shared memory, standard inter-process communication, or even running the virtual platform as an embedded part of the fuzzing tool process. The implementation of this interface can have an impact on the performance.

Note that there are other ways to inject data into the system under test in cases where the use of a testing harness is not appropriate. The VP can inject operations into device models or overwrite data in memory in response to target software operations [6][8]. Such implementations would require a different input driver, as well as redefining the successful end of a test and test failure criteria in the system monitor.

IV. System Monitor

The system monitor watches the execution of the code in the VP and detects both successful completion of tests and test failures. It runs inside the VP framework and is invisible to the code running on the target (this is a benefit of using a VP instead of instrumented code). The precise set of conditions to detect and their meaning depends on the software stack and system – but handling the detected conditions and communicating them to the fuzzing tool is a common generalized functionality.

In order to build a generalized monitor, the smallest common denominator for what constitutes a test failure must be identified. This obviously depends on the simulator framework and how models are built. In the Intel Simics simulator, the simplest way to detect failures is to use (generalized) breakpoints. Breakpoints can be (conditional) software breakpoints that trigger when the software reaches software exit points, virtual time breakpoints to detect stuck software, model log breakpoints to detect when the hardware models in the VP report unwanted activity, (conditional) memory access breakpoints on black-listed memory regions to detect out-of-bounds memory accesses, exception breakpoints to detect conditions such as triple faults, and breakpoints on simulator notifications to detect events like processor resets.

The end of a "successful" run is detected either by a breakpoint at the end of the test driver code or by the test driver code issuing a specific magic instruction that is captured by a magic instruction breakpoint.

The breakpoints are application-dependent and are created in the setup script that creates the system under test. For each breakpoint, the script also provides a description string. The description is used by the fuzzing tool to make sense of the breakpoint. It provides the semantic mapping from simulation events to fuzzing events.

The system monitor is provided with a list of pairs *(breakpoint ID, description)*. The system monitor attaches a callback to each breakpoint ID, and if the breakpoint is hit, it emits the associated description into a "stop criterion" buffer and stops the simulation. The simulator stop is detected by the state manager (see below), which takes the

stop reason from the stop criteria buffer and passes it to the fuzzer. This allows us to keep the system monitor entirely generic. Note that it is possible that multiple breakpoints hit within a single simulation step – in this case, it is, in general, enough to provide a single reason to the fuzzer, as it cannot typically make use of more information.

The external fuzzer program typically features a real-time timeout that is used to detect the target application locking up. For the VP case, such a time-out essentially detects that the simulator itself is hanging or crashing. To recover from such a situation, the fuzzer kills the VP process and starts a new process. Note that the case of the target software getting stuck is covered by virtual-time breakpoints.

Technically, the VP setup script could set a breakpoint in real time and use that as a stop condition. However, that is of fairly limited value as such a time-out has no relationship to what the software under test is doing, and it would likely trigger on conditions like the simulator running unusually slowly due to interference from other software on the host.

V. Coverage Collector

Keeping with the principle shown in Figure 2 and Figure 3, the coverage collector runs in the simulator but collects information that looks like it came from instrumentation compiled into the binary (mimicking the data provided by user-level standard fuzzing). We have found branch coverage to be the most useful coverage model. There might be cases where other coverage criteria are more useful or required by a certain fuzzing tool, and in such cases, it would be necessary to write different coverage collectors.

Branch coverage is implemented using the Intel Simics simulator *instrumentation interface*. Using this interface, the coverage collector requests to be notified whenever a new instruction is decoded by an instruction-set simulator (ISS). If the instruction is considered an "interesting instruction" (i.e., a branch instruction), a callback is installed that is called whenever the instruction is executed. The callback takes the destination address of the branch instruction as an identifier for the basic block that is entered. It combines this with the start address of the current basic block (i.e., the destination address of the previous branch) and increments the associated entry in the AFL hash map. This is how AFL measures the control flow coverage of the fuzzing run. This coverage collector is essentially the same as that used in the Excite project [7].

Coverage can be applied to all code or only to code executing within a certain address range. In general, collecting coverage across all code is simpler and works well for most cases. Collecting coverage only for a certain address range makes sense for cases where the really interesting code is reached via a complex of other code that cannot be stubbed out.

Like the system monitor, the collector is generally reusable and does not require any additions to the instruction-set simulator (ISS) or the simulator core. The coverage collector can even be reused across multiple instruction-set architectures by specifying the set of interesting instructions relevant to the architecture.

VI. State Manager

The state manager is responsible for resetting the state of the VP before each fuzzing test, as well as starting the simulator and stopping it. For maximum generality, the state manager only accepts two commands from the fuzzer: *reset* and *run*. Neither the commands nor the stop detection are application-specific, and hence, the state manager is generally reusable, just like the coverage collector and system monitor.

The state manager detects simulation stops resulting from the actions of the system monitor as described above. Effectively, the simulator run/stop state is used to communicate between the system monitor and state manager, with no need for an explicit link between the two modules. The state manager does not itself stop the simulation, but it is responsible for starting it after a run command is received.

A. State Reset: Snapshots

Fuzzing is all about rapidly running many tests from the same starting point. Thus, a mechanism is needed to quickly get back to a certain system state. This state is usually not the reset state of the system but rather somewhere

after the system/software under test has started and some software has been run. For user-level fuzzing on Linux, the typical approach is to fork the process under test after it has reached the point where fuzz tests are to be performed. For fuzzing on VPs, it is necessary to restore the whole VP state. This is implicitly contained in the VP process state, and forking the VP process itself has been tried [7]. However, forking does not work very well with threaded programs in general, and a highly-optimized multi-threaded simulator like the Intel Simics simulator cannot be reliably forked.

Instead, we use the simulator's in-memory checkpointing feature, commonly known as *snapshotting or snapshots*. As shown in Figure 4, the state of the target processors and devices are captured as an absolute state dump, while memories and disks use a diff mechanism that tracks changes between successive snapshots. Compared to forking, snapshots provide a mechanism that is under simulator control, that can be optimized for better performance, and that works regardless of the threads used in the simulator. Note that the software state is not shown specifically in Figure 4, as it is part of the state of the hardware and the contents of memories and disks.

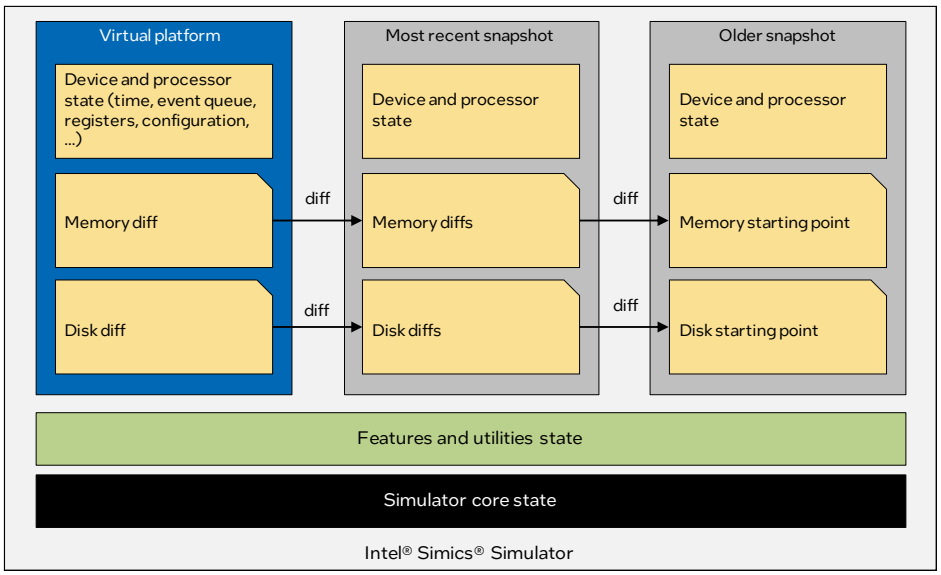

Figure 4

Restoring a snapshot is faster than reading a checkpoint from disk since the VP process is already running and all simulation objects already exist. Compared to full reverse execution [9], snapshot handling is much lighter-weight and faster [8].

B. Snapshot Size and Performance

The overall performance of the fuzzing solution is often determined more by the time it takes to perform the state reset operation than the time it takes to execute the code under test. Thus, performance can be improved by minimizing the size of the VP snapshot. Reducing the snapshot size comes down to two aspects: minimizing the size of the state saved for each VP model and minimizing the size of the VP overall.

Models should make use of memory images and diff saving for all large data and avoid large fixed-size arrays that go into the absolute state. The size of the VP overall can be achieved by creating a VP setup with just a single firmware-executing subsystem or by stripping down an existing larger platform model to the minimum needed to run the fuzzed software. Creating different VP setups customized to different use cases is common practice, especially in pre-silicon development.

C. Detailed Reset-Run Flow

To realize the fuzzer flow, we employ the so-called *persistent mode* of AFL. This means that AFL will not use forking and instead expects the test harness to manage the fuzzing loop by itself. The test harness consists of a setup phase followed by the main fuzzing loop.

To keep the setup part simple and reusable, the following protocol is used: When the simulator is started, the simulator fuzzing-test-specific setup script must run the simulator up to the fuzzing start point (detected by the test-specific script). When that point is reached, the script creates a snapshot and stops the simulation (with a specific stop message, just like any other stop). The stop is detected by the state manager and communicated to the fuzzer. The setup script also creates the breakpoints and configures the stop conditions for the system monitor.

It is important to note that the state manager is not aware of the setup phase (and neither is any other module). This is purely between the application-specific host-side part of the test harness and the application-specific simulator setup script.

From the fuzzer side, the setup phase consists of starting a new Intel Simics simulation session in a separate process or thread (using the setup script mentioned above), waiting for the communication with the state manager to be established, and then waiting for the first stop. At this point, the main fuzzing loop is entered.

The main loop starts with a *reset* command being sent to the state manager, which restores the in-memory snapshot (sets the VP state to the saved initial state). Next, data is exchanged with the target-side test driver, as discussed above. The third step in the loop is the run command, which blocks until the state manager detects a stop in the simulation. After the run command returns, the stop message can be fetched from the state manager.

The overall flow is generic and can be reused for different software/firmware setups. The application-specific aspects are confined to the simulator setup script, the mechanism for exchanging data between the test driver and fuzzer, and the checking of the return messages.

D. Using On-Disk Checkpoints

While on-disk checkpoints are not typically suitable for state reset in the main fuzzing loop, they can still be used to provide the initial state. In this case, the state would be established in a separate simulation session and saved as a checkpoint, and the setup script would simply open this checkpoint before saving the initial snapshot. The advantage is that multiple fuzzing runs can start from the same state without each fuzzing session having to do the redundant work of running the platform to the initial state. A generalization of this idea is to run a large software stack through multiple phases in a single preparation run, saving a checkpoint at each point that could be a starting point for fuzzing.

VII. SANITIZERS

Experience indicates that adding **sanitizers** to the fuzzed code improves error detection [1] and simplifies error triage and root-cause analysis by providing insight into the software behavior (beyond “it failed”). However, using sanitizers with firmware is not as simple as using them with user-level code. In particular, it is necessary to provide a sanitizer back-end library that communicates the results directly to the fuzzer via the VP, as shown in Figure 5. The sanitizer output receiver identifies failures reported by the sanitizer code and stops the simulation, adding information to the stop buffer. The system monitor or other parts of the system are not affected.

The code-size increase resulting from the instrumentation can cause complications for code that runs out of limited-size on-chip memory. Increasing the size of the memories might not be possible as that would change memory maps and have knock-on effects on other parts of the firmware. Thus, it might be necessary to apply sanitizers to only a small part of the firmware at a time.

It should be noted that using instrumentation like this requires the availability of source code for the firmware being tested and thus moves VP-based fuzzing into “white-box” territory.

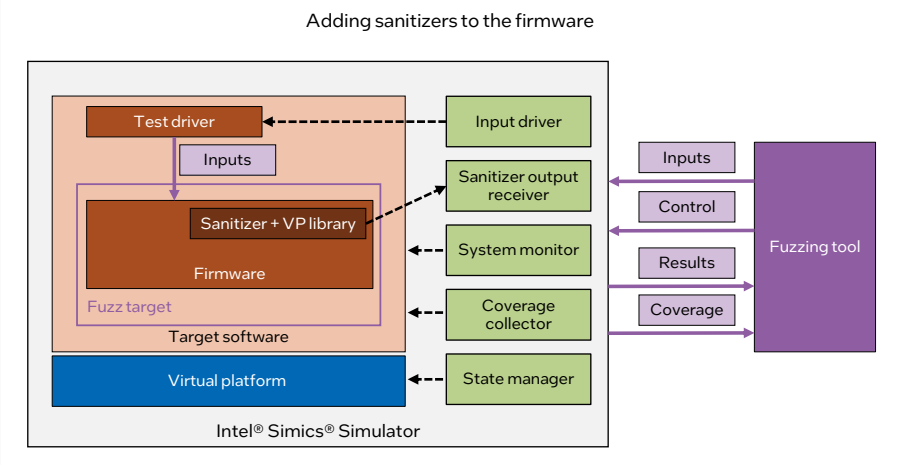

Figure 5

VIII. SUMMARY

This paper has presented our learnings from applying the Intel Simics simulator to the fuzzing of firmware code and other low-level code across several internal projects. A clear and reusable pattern has been observed, featuring common components and overall structure. We have made several of the components generic and reusable. In particular, the system monitor, state manager, and coverage collectors are 100% reusable independent of the software under test and the platform being used. The input driver is mostly reusable but may need extensions when specific HW inputs are needed. The adaptations needed for any particular application are localized to the simulation setup script and the on-target test driver software. VPs are incredibly useful for fuzzing since they allow fuzzing to be applied to deeply embedded code, low-level code, and firmware that are not fuzzable using standard execution platforms.

ACKNOWLEDGMENT

Thanks to Brandon Marken and Rowan Hart for their input and feedback.

REFERENCES

[1] Clément Poncelet, Konstantinos Sagonas, and Nicolas Tsiftes: “So Many Fuzzers, So Little Time - Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack”, *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, October 10–14, 2022.

[2] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. “LibAFL: A Framework to Build Modular and Reusable Fuzzers”, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[3] Daniel Aarno and Jakob Engblom: *Software and System Development using Virtual Platforms – Full System Simulation with Wind River Simics*, Morgan Kaufmann Publishers, 2014.

[4] Dominik Maier and Fabian Toepfer. 2021: “BSOD: Binary-only Scalable Fuzzing of Device Drivers,” *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[5] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. “PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation”. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[6] Tamas Lengyel, “Fuzzing Linux with Xen,” *DEFCON 29*, August 2021

[7] Zhenkun Yang, Yuriy Viktorov, Jin Yang, Jiewen Yao, and Vincent Zimmer: “UEFI Firmware Fuzzing with Simics Virtual Platform,” *57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[8] Chad Bollmann and Mike Thompson, *Improved Cyber System Digital Twinning, Reverse Engineering, and Vulnerability Analysis through RESim, a High Fidelity, Full System Simulator*, White Paper, February 2022

[9] Jakob Engblom. “A review of reverse debugging,” *Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2012)*, Wien, Austria, September 19-20, 2012.