UEFI Firmware Fuzzing with Simics Virtual Platform

Zhenkun Yang, Yuriy Viktorov, Jin Yang, Jiewen Yao and Vincent Zimmer
Intel Corporation
{zhenkun.yang, yuriy.viktorov, jin.yang, jiewen.yao, vincent.zimmer}@intel.com

***Abstract*—This paper presents a fuzzing framework for Unified Extensible Firmware Interface (UEFI) BIOS with the Simics virtual platform. Firmware has increasingly become an attack target as operating systems are getting more and more secure. Due to its special execution environment and the extensive interaction with hardware, UEFI firmware is difficult to test compared to user-level applications running on operating systems. Fortunately, virtual platforms are widely used to enable early software and firmware development by modeling the target hardware platform in its virtual environment before silicon arrives. Virtual platforms play a critical role in left shifting UEFI firmware validation to pre-silicon phase. We integrated the fuzzing capability into Simics virtual platform to allow users to fuzz UEFI firmware code with high-fidelity hardware models provided by Simics. We demonstrated the ability to automatically detect previously unknown bugs, and issues found only by human experts.**

***Index Terms*—Firmware, Security, Simics, Fuzzing, UEFI**

I. Introduction

Computer security is important, with potential exploits ranging from the application space down to the hardware. Among the various layers of the stack, firmware security is assuming a more prominent role [1], [2]. Firmware is the layer of software that ships with the hardware, typically in a semiconductor non-volatile storage, such as SPI flash, and manages a plurality of activities. These activities include initialization of the CPU cores, memory controllers, I/O buses, and ancillary devices. The firmware is typically the highest integrity software in the system that is responsible for maintaining the chain of boot trust, selecting the operating system, and providing various configuration options to the end user. The Unified Extensible Firmware Interface (UEFI), along with standards such as System Management BIOS and Advanced Configuration and Power Interface, provides a consistent environment to the operating systems (OS) and pre-OS applications. To-date UEFI style firmware has shipped on over 4 billion machines.

One common misconception people often have entails the believe that once the BIOS boots the platform and handles the control to the OS, its life time ends. In fact, UEFI BIOS provides critical services while OS is running. For example, on Intel-based platforms, the OS can enter System Management Mode (SMM) by triggering System Management Interrupt (SMI). SMI handlers that are installed in BIOS will be triggered to serve requests from OS. SMM is the most privileged state of execution on Intel-based platforms. SMM is the perfect place to hide a root kit because code running in SMM can read and write everything on the platform, while being invisible to OS and anti-virus software.

Software community has common practices and great tools available for quality assurance. For example, debugging and profiling tools are widely used for software development. More advanced techniques such as fuzzing, symbolic execution and static analysis are becoming popular. However, firmware development and validation community faces numerous challenges applying those tools due to the special execution environments firmware is running on. The execution regime of boot firmware does not resemble any known operating system runtime, such as Linux or Windows, thus requiring custom, bespoke solutions.

These are various challenges in assuring UEFI firmware. The non-volatile SPI flash which stores the UEFI firmware has to support other binary objects, such as management controller firmware, core microcode patches, manufacturing defaults, and redundant BIOS elements to support a fault-tolerant update. This beach-front property dimension of the storage is compounded by the fact that a modern UEFI firmware solution is typically built from over 200 relocatable executable images. Any attempt to provide instrumentation, checkers, or in-line reference monitors will aggravate the storage challenge. For example, binary instrumentation will bloat the image size, thus making the image not fitting in the SPI flash.

Virtual Platforms (VP) are widely used in industry for pre-silicon prototyping. A typical example is how industry uses Simics for early software/firmware development. VP provides a virtualized execution environment for firmware and the underlying hardware. Although VP is mainly used for pre-silicon software and firmware enabling, it has great potential for security testing of UEFI firmware at early development stage. We leverage the full visibility VP provides and the recent advancement of software analysis techniques to test UEFI firmware more efficiently in an automated fashion.

There has been work [3] on using *symbolic execution* technique [4], [5] on SMI handlers. The basic idea is to dump SMM memory (SMRAM) from a live system during the UEFI firmware boot process, then memory-map the dumped SMRAM into a user-level application via a test harness. An off-the-shelf binary symbolic execution engine is used to explore paths of the test harness as a user-level application from a given SMI handler's entry point. The idea works great if privileged instructions and I/O operations are absent from the SMI handlers. However, SMM is designed for handling

system-wide functions like power management and system hardware control, therefore privileged and I/O instructions are very common in SMM. Intel has developed a new tool, Host-based Firmware Analyzer [6], for the UEFI community. It allows UEFI firmware developers to run advanced testing tools such as fuzzing, symbolic execution, and address sanitizers in a host environment. The basic idea to extract software logic of the UEFI code to user-level applications and use off-the-shelf state-of-the-art software testing tools to catch firmware issues at the development stage. However, when the code under test interacts with the underlying hardware a lot, lifting the firmware code to user-level applications is nontrivial.

Fuzzing has become a de facto standard in automated software security testing domain recently due to its effectiveness and ease of use. Fuzzing is a powerful technique to ensure the robustness and security of software systems. American Fuzzy Lop (AFL) is one popular coverage-guided fuzzer that utilizes genetic algorithms to permute inputs based on code coverage feedback in an effort to identify specific inputs that cause crashes or hangs in target code. For example, syszkaller [7] focuses on fuzzing OS kernel system calls. kAFL [8] uses hardware assisted technique, *i.e.* Intel Processor Trace, to speed up fuzzing of OS Kernels. Triforce [9] supports full-system fuzzing using QEMU emulator, but it is mainly used for fuzzing OS due to limited availability of hardware models. To the best of our knowledge, there is no prior work on fuzzing UEFI firmware with the help from virtual platforms. This work enables guided fuzzing capabilities within Simics, it provides users the capability to fuzz the software and firmware running inside Simics platform and drive hardware inputs by leveraging the high-fidelity hardware models provided by Simics. This shifts left the security validation of software and firmware, and enables bugs and security vulnerabilities elimination at early stage.

We demonstrate the effectiveness and versatility of the fuzzing framework with three usage scenarios: 1) security researchers who want to test BIOS security with only access to the BIOS binary (no source code available); 2) BIOS development teams that have access of BIOS source code but limited knowledge of virtual platform, and are willing to modify source code to interact with fuzzing engine for more efficient testing; and 3) BIOS validation teams that have access of BIOS source code and knowledge of virtual platform for more comprehensive testing, *e.g.* driving untrusted I/O inputs to the BIOS. Experiments show that our fuzzing approach can detect previously unknown *high-critical* bugs that may lead to privilege elevation, and issues that were only found by security experts with manual code inspection.

II. Background

A. UEFI Firmware Security

UEFI provides a rich set of security capabilities. These are often referred to as the various Roots of Trust (RoT). RoT is a capability that must implicitly be trusted. This includes the *update* capability of the UEFI firmware. This is a set of code that processes potentially attacker-controlled input in order to orchestrate a replacement of existing UEFI elements. UEFI BIOS also serves as a RoT for *verification*, such as Secure Boot [10], which entails a high degree of complexity. There have been many issues found in these regimes over time [11]. Finally, there are other UEFI BIOS based RoTs, namely the RoT for *recovery* and RoT for *measurement*. In the case of *recovery*, any flaws in the UEFI recovery flow can render machine into an inoperable state, or "brick." And in the case of the *measurement*, the UEFI BIOS logic that records the cryptographic hash of the boot executables and data into the Trusted Platform Module can be bypassed [12].

B. Virtual Platform

A VP is a software system that models a hardware system that can run the same software as the hardware it models. A VP is simulated on a host computer that may be different from the hardware modeled by the virtual platform. Popular virtual platforms include open-source QEMU [13] emulator and commercial Simics [14] full-system simulator. VP allows software and firmware (SW/FW) development happen before silicon arrives. The shift-left benefits brought by VP help vendors to shorten the time-to-market of their products.

However, there are a few issues with VP itself and the way how industry is using VP. SW/FW are not easy to be tested in the encapsulated VP environment as compared to software running as user-level applications where engineers have many available tools to test their software, *e.g.* code coverage, memory profiling, and fuzzing. For example, many code coverage tools are readily available in software development industry for years, whereas a good solution to measure UEFI firmware code coverage in Simics was made available only recently. As a result, the SW/FW being developed in Simics is not well tested before it is shipped to the customers. To the best of our knowledge, VPs are mainly used for early platform bring-up, which means that the thorough functional and security validation is still done on real hardware at post-silicon stage. However, security has become more and more important given recent exposure of security vulnerabilities [1], [2]. Performing security validation at post-silicon stage in a compressed timeline results in insecure SW/FW. This paper tries to bring the best testing practices used in software community to firmware security validation into VP to enable early quality assurance and security testing of SW/FW.

C. Fuzzing

Fuzzing is currently one of the most easy-to-use and popular dynamic testing techniques for security vulnerabilities discovery. Conceptually fuzzing generates lots of inputs, expected or unexpected, to the program under test, and monitors the program for abnormal behavior and exceptions. A fuzzer can be categorized as generation-based or mutation-based depending on how inputs are generated. Generation-based fuzzers generate inputs from scratch with or without the knowledge of program input structure or semantics. Mutation-based fuzzers generate inputs by modifying existing ones with some heuristics. AFL is a popular mutation-based fuzzer

which uses compile-time instrumentation to collect coverage feedback of the target program and uses genetic algorithms to permute inputs to maximize code coverage and therefore to cause crashes or hangs in the program. LibFuzzer is another coverage-guided fuzzer that is part of the LLVM compiler framework. However, these fuzzers are designed to fuzz user-level applications (with source code available) that are running on an operating system, therefore fuzzing firmware running in a VP is not possible with the current off-the-shelf fuzzers.

III. Fuzzing Framework

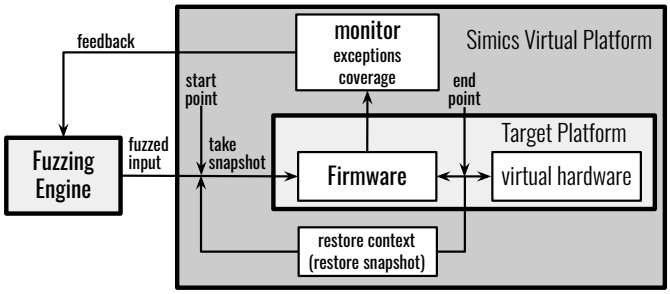

Fig. 1. Architecture of the fuzzing engine and Simics virtual platform. Fuzzing engine and Simics run on *host* machine, and *firmware* and virtual hardware (virtual processor, memory, and devices) run on *target* platform inside Simics. Detailed workflow of the framework is shown in Fig. 2.

A. Design

We now present the high-level overview of the design of our feedback-driven firmware fuzzing framework that supports unmodified and un-instrumented firmware image with Simics virtual platform. When designing a feedback-driven fuzzer, the following questions should be considered:

1) How and where to use fuzzed input: the fuzzer will generate raw bytes as fuzzed input to the target program, It is the user's responsibility to figure out where and how to read the input to the program under test.
2) How to save/store program state: for long-running programs where the functionality under test is in the middle of the program, starting the program from beginning and running the program all the way to the end is going to slowdown the fuzzing process. Some mechanism to save and restore the program state during fuzzing to avoid unnecessary repetition is required.
3) How to collect feedback: execution feedback is crucial for the fuzzing performance. It guides the fuzzer to generate more effective test inputs to cover more branches.
4) How to monitor abnormal behavior: Fuzzer needs to know the program's symptoms if something goes wrong.

Fig. 1 shows the overall architecture of our UEFI firmware fuzzing framework, which includes AFL fuzzing engine, and Simics virtual platform. The target platform, which runs firmware and includes the underlying virtual hardware, is emulated by Simics. From fuzzing engine's perspective, the whole Simics virtual platform is the binary under test. However, we configured Simics in a way that the target CPU is only executing the portion of the firmware that we are interested in testing, thus, we are effectively testing the firmware with Simics virtual environment. We address the questions raised in the beginning of this section as follows:

1) **inject fuzzed input**: This can be done either from Simics to inject the data into correct location, or from firmware by calling custom APIs to request input from Simics (see Section III-B for detail)
2) **save/restore**: This is done by forking (`fork` system call) the entire Simics process to preserve the execution state.
3) **feedback**: We configure Simics to only trace the execution of the firmware. Fuzzer has the illusion that it is fuzzing the whole Simics process, but we only collect the feedback on the firmware execution.
4) **monitor**: We instrument Simics to trap important firmware panic functions, such as `CpuDeadLoop()`.

There are a few reasons why off-the-shelf fuzzing engines do not work for software or firmware running in virtual platforms: 1) The fuzzing engines rely on compiler instrumentation to instrument the source code to collect feedback (execution tracing data) during the fuzzing process, the feedback will be used to determine if a mutation is worth keeping or not. However, software and firmware are compiled with all kinds of compilers, instrumenting the code may not be even possible due to firmware size restrictions, compiler restrictions, performance concerns, or even the availability of source code. 2) Both the fuzzing engine and the program under test run as applications in an operating system. The fuzzing engines usually uses `fork()` system call to preserve program states between each run of the inputs. However, system software and firmware run inside virtual platforms, therefore process hooking and forking is not permissible under virtual platforms.

B. Implementation

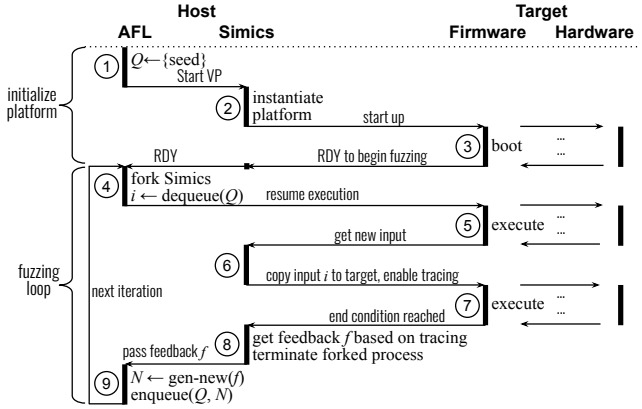

Fig. 2. Workflow of the integration of AFL fuzzer and Simics virtual platform. Arrows from firmware to Simics are implemented by means of magic instructions; state save/restore are implemented via `fork` system call.

Fig. 2 illustrates the overall guided fuzzing process: in step ①, AFL fuzzer starts and initializes the test case queue Q with the user-provided initial input (also called seed, in the

form of file), and launches Simics as a child process. Then Simics instantiates the given target platform with its high-fidelity hardware models and begins to boot *firmware* in step ②. In step ③, firmware starts running and initializes the underlying hardware to the point of interest, and signals Simics that it is ready to begin fuzzing. Simics forwards the signal to its parent process AFL. In step ④ AFL forks the Simics process, which preserves the state of the VP together with the state of the firmware running inside of it. AFL gets an input file i from its test case queue Q, and signals the child Simics process to proceed with firmware execution. In step ⑤, firmware executes to a point where it needs to read the input, then in step ⑥ Simics copies the input i from host to the target platform, and enables tracing the firmware execution with its tracing capability. The tracer that we developed in Simics keeps track of the branching information during firmware execution. In step ⑦, firmware consumes the fuzzed input, executes, and signals Simics until it reaches the user-specified end point. In step ⑧, Simics processes the traced data to get the branch coverage, feeds the information back to AFL, and terminates itself. In step ⑨, AFL generates new inputs based on the branch coverage feedback and the mutation strategy, and transitions to step ④.

Fuzzer repeatedly mutates the input file with fuzzing strategies, and collects coverage feedback based on the execution tracing. If any of the mutations yields new code transition recorded by the tracing instrumentation, fuzzer adds the mutated input into the queue and to the list of generated test inputs. The whole process repeats until the user stops it or a user-specified timeout is reached.

The performance of the process forking critically depends on the memory footprint of the process. For the VP process it can be as big as the entire virtual address space plus VP overhead (tens of GB of RAM). In order to minimize Simics process memory footprint we use the check-pointing mechanism. The checkpoints are loaded lazily (only when a corresponding data are accessed). So, saving and restoring the checkpoint after VP is set to initial state significantly reduces the memory footprint and allows faster process forking.

We also need to efficiently trace the firmware execution in Simics. The actual approach for execution tracing is VP specific and the fastest way to trace the code execution inside a VP depends on the features available. We use Simics' instrumentation framework (with performance in mind) as a mechanism for inspecting various parts of the running simulated system. This approach, compared to the other approaches available in Simics, avoids overhead by avoiding unnecessary thread synchronization.

Communication of the firmware code running on a target machine with the VP is commonly done by magic instructions (the instructions that has a special meaning for VP and is interpreted as a message to VP once executed on target). Simics implements magic instructions via `CPUID` instruction with unused value range for `RAX`. We extended this approach, passing additional message information in `RBX`, `RCX` and `RDX` before calling `CPUID` and passing response back in `RSI` after. This is a foundation for the firmware-to-VP communication protocol used by firmware to: 1) begin the fuzzing process, 2) request a new fuzzed input (`GetFuzzedInput()`), and 3) notify that the start or end point is reached. Our implementation of the firmware-to-VP communication protocol preserves the normal execution of the instrumented code on a host, while adding a special behavior when executed on target.

IV. Experimental Results

In this section we demonstrate the versatility of our fuzzing framework with three different ways (Table I) to instrument the execution and introduce fuzzed inputs to the firmware:

1) Running unmodified and uninstrumented firmware, detecting the start/end point within Simics, and injecting fuzzed input from Simics to a proper location where firmware then consumes it.
2) Modified firmware running on the target signals Simics when execution reaches the start/end point. Simics injects the fuzzed input to a proper location.
3) Modified firmware running on the target signals Simics when execution reaches the start/end point and requests fuzzed inputs from Simics by explicitly calling `GetFuzzedInput()`.

TABLE I
Experiments for different fuzzing scenarios of the framework

#	start/end point	Fuzzed Input	Experiment
A.	Simics detects	Simics injects	Fuzzing SMI Handlers
B.1	Firmware signals	Firmware requests	Fuzzing USB I/O at firmware level
B.2	Firmware signals	Simics injects	Fuzzing USB I/O at memory level

For experiment *A*, we demonstrate that given a firmware binary image without the source code, we are able to locate SMI handlers in SMRAM within Simics, inject fuzzed input from Simics, and fuzz the SMI handlers. For experiment *B.1*, we demonstrate that with access to the firmware source code, firmware can interact with the fuzzing engine by explicitly calling fuzzing APIs to signal the start/end point and request fuzzed input. For experiment *B.2*, since Simics provides full visibility and control over the high-fidelity hardware models, we demonstrate that we are able to fuzz the hardware I/O, *i.e.* driving potentially malicious hardware inputs to firmware from Simics and testing the resiliency of the firmware.

All the experiments are done on a workstation with Intel Xeon E5-2697 @ 2.60 GHz processor and 64 GB of RAM. We use EDK II minimum platform firmware for Simics X58 platform[1]. The performance of the integrated fuzzing solution is about 100 iterations per second. It lies between tens and hundreds of iterations per second, depending on the VP memory footprint, length of the code block to fuzz and how deep execution goes on particular input. The performance is about 10x slower, compared to fuzzing applications in user-space, yet sufficient to fuzz complex firmware code effectively with high-fidelity hardware models in Simics.

[1]https://github.com/tianocore/edk2-platforms/tree/master/Platform/Intel/SimicsOpenBoardPkg

A. *Fuzzing* SMI *Handlers*

SMM is the highest and the most privileged CPU operating mode on Intel-based platforms. Code running in SMM mode resides in a special memory region (SMRAM) that is protected by hardware. Code running in SMM mode has access the whole system memory (including OS and hypervisor) and I/O. However, under normal circumstances, code running in SMM should only access certain address ranges. SMM mode is entered by invoking system management interrupt (SMI). SMM is intended for use only by BIOS, and SMI handlers are installed by BIOS. Upon receiving SMI, BIOS calls the corresponding SMI handler to serve the interrupt request. Each handler communicates with the caller through a buffer and its size. Here is an example of a SMI handler `SampleSmiHandler`. Parameter `CommBuffer` and `CommBufferSize` are used for exchanging data between the handler and non-SMM agents.

```
1  int SampleSmiHandler(EFI_HANDLE DispatchHandle,
2          const void   *RegisterContext,
3          void         *CommBuffer,
4          unsigned int *CommBufferSize)
```

Since `CommBuffer` and `CommBufferSize` are source of untrusted inputs from non-SMM agents, *e.g.* operating system, one of the main attack vectors on SMM is to construct malicious communication buffers, which will cause SMI handlers to corrupt SMM data, branching the execution or accessing memory outside of SMRAM.

To demonstrate the capability of detecting such issues with our fuzzing approach. We fuzz the `CommBuffer` and `CommBufferSize` of the SMI handlers of interests. We instrument Simics to trap every memory access. Upon each memory access, we examine whether the memory location is within ranges permitted for SMM. If the fuzzer generates an input to the SMI handler that triggers an out-of-range memory access, we then have found an issue in the SMI handler.

For SMI handlers, we know that they are in SMRAM, and the location of SMRAM is known. We boot the firmware in Simics until SMRAM is locked, then we stop the execution and scan SMRAM for the exact location of the entry point of each SMI handler. Since the signature of SMI handlers and `CommBuffer` location are known in advance, we can simply put the fuzzed input to the target system memory, put relevant data on the stack (return address *etc.*), and load values to the CPU registers according to calling convention. For example, address of `CommBuffer` is loaded into `R8`, address of `CommBufferSize` into `R9`, handler entry point into `RIP` etc. Therefore, both the start/end (entry point/return address) point and data injection (`CommBuffer` placement) are managed by Simics.

We discovered issues in *two* SMI handlers. The issues are related to memory accesses that are outside of SMRAM. We now explain the issue in `OpalPasswordSmm` SMI handler for illustration. The fuzzing engine generated a "buffer" and "size" which triggered out-of-range memory access. `OpalPasswordSmm` SMI handler takes a fuzzer generated `CommBuffer` as input and eventually calls function `ProblematicFunc`. Note the parameter `node` of function `ProblematicFunc` is derived from `CommBuffer`. So, without proper validation of the pointer, it is casted into another type and then dereferenced (line 3 of the following listing).

```
1  unsigned char ProblematicFunc(const void *node){
2    assert(node != null);
3    return ((EFI_XXXX_PROTOCOL *)(node))->Type;
4  }
```

The bug was filed to Tianocore Bugzilla bug tracking system. According to the triage of the issue by the maintainers, this issue is classified as *high critical* issue which may lead to privilege elevation.

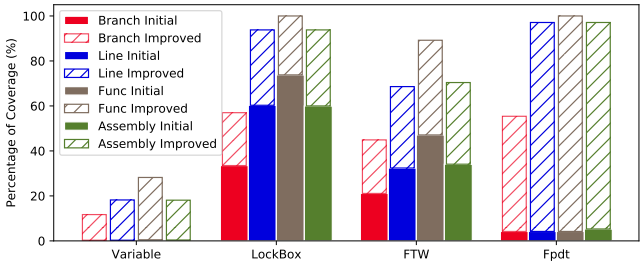

Fig. 3. Coverage improvements after applying fuzzing: solid bars represent the coverage during normal firmware boot flow, and stripe bars represent the coverage improvements with fuzzer generated test cases from initial seeds.

Besides `OpalPasswordSmm`, we also tested 4 other SMI handlers with a 2-hour timeout. Fig. 3 shows the coverage improvements after applying fuzzing with the initial seeds that are captured during normal firmware boot to UEFI shell, which is typically the only testing approach for SMI handlers. We can see the improvements range from about 30% to more than 10x. It is worth noting that some handlers, *e.g.* `Variable`, include the OpenSSL library, that is why the overall coverage is low.

B. *Fuzzing Hardware I/O*

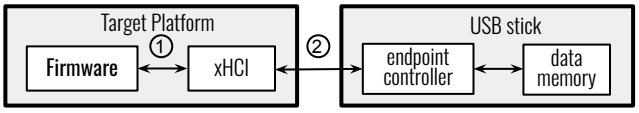

Fig. 4. Diagram to illustrate the communication between a USB stick and the UEFI firmware. Endpoint controller is the source of malicious input; USB host controller xHCI copies malicious input from ② to ① without extra validation, and firmware is the real consumer of the malicious input. ① represents in-system communication, *i.e.* DMA, ② represents USB serial interface, *i.e.* serial protocol message.

The common problem in UEFI firmware development and validation with hardware I/O is the assumption about well-behaving hardware, which leads to insufficient validation of the input from hardware. In other words, firmware developers and validation engineers often trust the hardware. However, when hardware misbehaves, this assumption does not hold with different implications from security risks to system crashes. As an example, consider a malicious USB stick sending unexpected responses to the PC. Such USB sticks are commercially available for about $20 and can be programmed even in Python. Once connected to the PC, firmware needs

to communicate with it. If firmware assumes it is a benign hardware and trusts its response, then firmware may put itself into a vulnerable state.

We reintroduced one of the known issues[2] to demonstrate the ability to automatically detect issues that previously either escaped to final product or were only found by human experts. More specifically, the issue can be described as follows: When a new USB device is connected to a platform (Fig. 4), firmware attempts to configure this device by calling function **`UsbBuildDescTable()`** in the process. This function first calls `UsbHcControlTransfer()` to read the device descriptors initiating a transfer from the device and getting a response from USB endpoint controller (see ② in Fig. 4). Then a buffer with the device descriptors is parsed and further processed by calling function `UsbParseConfigDesc()` (see ① in Fig. 4). The buffer of device descriptors is generated by the hardware. However, it is considered as untrusted input from firmware perspective. Malformed descriptor may result in buffer overflow, array-out-of-bound access etc.

Here is the simplified code snippet showing one of the buffer overflow issues in USB code. In the **`while`** loop at line 9, the check `(Offset < Len)` only ensures that the struct `USB_DESC_HEAD` starts within the `Buf`, but does not guarantee that it fits entirely. As the result, access to `Head->Type` may go out of bound of `Buf`.

```
1  typedef struct {
2    UINT8 Len;
3    UINT8 Type;
4  } USB_DESC_HEAD;
5
6  void *UsbCreateDesc(UINT8 *Buf, UINTN Len, UINT8 Type){
7    UINTN Offset = 0;
8    USB_DESC_HEAD* Head = (USB_DESC_HEAD*)Buf;
9    while ((Offset < Len) && (Head->Type != Type)) {
10     Offset += Head->Len;
11     Head    = (USB_DESC_HEAD*)(Buf + Offset);
12   }
13   DoSomethingUseful(Head);
14 }
```

The following is the correct version of the **`while`** loop condition, therefore making the `Head->Type` safe to access:

```
8  ...
9    while (Offset < Len - sizeof(USB_DESC_HEAD))
10 ...
```

We performed fuzzing at two different levels of abstraction. In both cases, the issue was detected by the fuzzer:

1) Fuzzing HW I/O at firmware level: In this experiment, firmware is responsible for requesting a new fuzzed input from VP by calling `GetFuzzedInput()`. The new input is then used to replace the data just transferred from device. This is done after the corresponding call to `UsbHcControlTransfer()` by replacing the returned buffer content. This approach requires firmware modification but is more straightforward.

2) Fuzzing HW I/O at memory level: In this experiment, VP monitors memory writes initiated by the xHCI host controller and replaces the data written to memory before the firmware starts processing it. This approach does not require firmware modification to inject fuzzed input. However, most memory writes done by xHCI do not contain USB endpoint data and are used for maintaining scoreboards, circular buffers, *etc*. Unguided modification of the data will usually break the xHCI protocol instead of triggering interesting behavior on BIOS side. Therefore, in order to fuzz efficiently, we need a model for the xHCI protocol to identify the data of interest within xHCI memory traffic and fuzz selectively.

[2]https://github.com/tianocore/edk2/commit/4c034bf62cbc1f3c5f4b5df25de97f0f528132b2

V. Conclusion

We presented a coverage-driven fuzzing framework for UEFI firmware with Simics virtual platform. This framework uses the tracing capability in Simics to collect branching coverage of the firmware execution as a feedback to the fuzzing engine. We demonstrated the versatility of the framework with three usage scenarios for UEFI firmware security validation, ranging from fuzzing unmodified and uninstrumented firmware to fuzzing hardware I/O interactions with and without source code modification. Experimental results show that our approach automatically detects previously unknown bugs that have been classified as *high critical* potential privilege elevation issues and also issues previously found only by security expert via manual code inspection.

References

[1] J. Loucaides and A. Furtak, "A new class of vulnerability in smi handlers of bios/uefi firmware," in *The 15th Annual CanSecWest Conference (CanSecWest 2015)*, 2015.

[2] X. Kovah and C. Kallenberg, "How many million bioses would you like to infect," 2015.

[3] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer, "Symbolic execution for BIOS security," in *WOOT*. Washington, D.C.: USENIX Association, Aug. 2015.

[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *OSDI*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," in *ASPLOS*. New York, NY, USA: ACM, 2011, pp. 265–278.

[6] B. Richardson, C. Wu, J. Yao, and V. Zimmer, "Using host-based firmware analysis to improve platform resiliency," https://firmware.intel.com/sites/default/files/Intel_UsingHBFAtoImprovePlatformResiliency.pdf, February 2019.

[7] "syzkaller: an unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller, accessed: 2019-11-06.

[8] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[9] "Triforce Linux Syscall Fuzzer," https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, accessed: 2019-11-06.

[10] J. Yao and V. Zimmer, "Understanding uefi secure boot chain," https://legacy.gitbook.com/book/edk2-docs/understanding-the-uefi-secure-boot-chain/details, May 2019.

[11] "Tiano Security Advisory," https://edk2-docs.gitbooks.io/security-advisory/content/, accessed: 2019-11-06.

[12] V. Bashun, A. Sergeev, V. Minchenkov, and A. Yakovlev, "Too young to be secure: Analysis of UEFI threats and vulnerabilities," in *14th Conference of Open Innovation Association FRUCT*, Nov 2013, pp. 16–24.

[13] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[14] D. Aarno and J. Engblom, *Software and system development using virtual platforms: full-system simulation with wind river simics*. Morgan Kaufmann, 2014.