2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179421

RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing

Jiawei Yin*‖‡‡, Menghao Li*§‖‡‡, Yuekang Li‡§, Yong Yu‖‡‡, Boru Lin*‖‡‡, Yanyan Zou*‖‡‡, Yang Liu‡, Wei Huo*‖‡‡, Jingling Xue†,

Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*, Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences‖, Beijing Key Laboratory of Network Security and Protection Technology‡‡, UNSW Sydney†, Nanyang Technological University‡

***Abstract*—*System Management Mode (SMM)* is a secure operation mode for x86 processors supported by *Unified Extensible Firmware Interface (UEFI)* firmware. SMM is designed to provide a secure execution environment to access highly privileged data or control low-level hardware (such as power management). The programs running in SMM are called SMM drivers and *System Management Interrupt (SMI) handlers* are the most important components of SMM drivers since they are the only components to receive and handle data from outside the SMM execution environment. Although SMM can serve as an extra layer of protection when the operating system is compromised, vulnerabilities in SMM drivers, especially SMI handlers, can invalidate this protection and cause severe damages to the device. Thus, early detection of SMI handler vulnerabilities is important for UEFI firmware security.**

To this end, researchers have proposed to use hybrid fuzzing techniques for detecting SMI handler vulnerabilities. Particularly, Intel has developed a hybrid fuzzer called Excite and uses it to secure Intel products. Although existing hybrid fuzzing techniques can detect vulnerabilities in SMI handlers, their effectiveness is limited due to two major pitfalls: 1) They can only feed input through the most common input interface to SMI handlers, lacking the ability to utilize other input interfaces. 2) They have no awareness of variables shared by multiple SMI handlers, lacking the ability to explore code segments related to such variables. By addressing the challenges faced by existing works, we propose RSFuzzer, a hybrid greybox fuzzing technique which can learn input interface and format information and detect deeply hidden vulnerabilities which are triggered by invoking multiple SMI handlers. We implemented RSFuzzer and evaluated it on 16 UEFI firmware images provided by six vendors. The experiment results show that RSFuzzer can cover 617% more basic blocks and detect 828% more vulnerabilities on average than the state-of-the-art hybrid fuzzing technique. Moreover, we found and reported 65 0-day vulnerabilities in the evaluated UEFI firmware images and 14 CVE IDs were assigned. Noticeably, 6 of the 0-day vulnerabilities were found in commercial-off-the-shelf (COTS) products from Intel, which might have been tested by Excite before releasing.

***Index Terms*—UEFI; SMM; Fuzzing; SMI handler; SMM Vulnerabilities;**

I. Introduction

The *Unified Extensible Firmware Interface (UEFI)* specification [1] is securing billions of devices worldwide, including personal computers, servers, smart phones, internet of things devices and so forth. When used with x86 processors, UEFI firmware provides two execution modes: the normal execution mode, and the *System Management Mode (SMM)*. The normal execution mode is used for running user-space and kernel-space programs while SMM is used for running programs which need access to highly privileged data or need control over low-level hardware (e.g., serial peripheral interface (SPI) flash). The programs running in SMM are called *SMM drivers* and *System Management Interrupt (SMI) Handlers* are the most important components of SMM drivers because they not only serve as the only channel to receive data from kernel-space programs but also carry out the diverse functionalities of SMM drivers.

The basic mechanism of SMM is to separate the execution environment of SMM drivers from the execution environment of user-space and kernel-space programs by creating a memory space dedicated to SMM drivers. This memory space is called *SMRAM* and it becomes inaccessible for any other programs, including kernel-space programs, after the system is booted up. By isolating the execution environment for SMM drivers, SMM can protect the computer even when the kernel is fully compromised [2], [3]. However, vulnerabilities in SMM drivers can lead to a breakdown of this protection [2], [4], [5]. Worse still, since SMM drivers execute with high privilege, exploiting their vulnerabilities can cause even greater damage than exploiting kernel vulnerabilities. For example, the exploitation of a vulnerability in an SMI handler can lead to UEFI Bootkit installation [2], [4], [5]. Therefore, ***vulnerability detection for SMM drivers, especially SMI handlers, is crucial for UEFI firmware security***.

Fuzzing is an effective technique for discovering software [6], [7], [8], [9], [10], [11], [12], [13] and firmware [14], [15] vulnerabilities. According to how much program intrinsic information is needed, fuzzing techniques can be categorized as blackbox fuzzing, greybox fuzzing and whitebox fuzzing. Among the three types of techniques, greybox fuzzing shines in practical usefulness as it strikes a balance between efficiency and effectiveness [15], [6], [7], [16], [17], [18], [13], [19], [8], [20], [21], [22]. To further boost the capability of detecting deeply hidden vulnerabilities for greybox fuzzing, researchers have proposed the technique of hybrid greybox fuzzing, where symbolic or concolic execution engines are introduced to help

§Corresponding Author

penetrate branch constraints in the programs.

On the one hand, as a testing technique, fuzzing excels in finding vulnerabilities in programs which have complex input handling logic. On the other hand, in order to perform various functionalities, SMI handlers need to receive and handle inputs with diverse formats and they are the only interface for SMM drivers to receive input data from kernel-space. Hence, ***SMI handlers are the most suitable targets for leveraging fuzzing to detect vulnerabilities in SMM drivers***.

Although SMI handlers are suitable for fuzzing, applying existing fuzzing techniques is challenging. Currently, several works [23], [24], [25] have considered fuzz SMI handlers. For Excite [25], its effectiveness is hindered by the pitfall that Excite only processes single-dimensional input space (i.e., the communication buffer between kernel-space programs and SMI handlers) but SMI handlers can also read inputs from hard-coded memory addresses. For [23], [24], [25], inputs are generated with no awareness of the cross-handler variables that may influence the execution of the SMI handlers. Besides the efforts from industry, in academia, the most related work is Syzgen [26]. Syzgen is designed to fuzz closed-source macOS drivers, which are similar to SMI handlers from the fuzzing point of view. Applying the techniques of Syzgen to SMI handlers is faced with even more challenges. For example, most vulnerabilities in SMI handlers are silent corruptions instead of crashes, making it hard for the fuzzer to detect.

To address the challenges faced by existing techniques, we propose a hybrid greybox fuzzing technique called RSFuzzer. RSFuzzer contains two major components: a concolic execution engine which can extract useful information about the SMI handler inputs and a greybox fuzzing engine which can run in single-handler fuzzing mode and cross-handler fuzzing mode. Every time the greybox fuzzing engine finds a new seed [1], the seed is used for concolic execution on an SMI handler before being added to the seed pool. By analyzing the concolic execution trace of the seed, RSFuzzer extracts two types of information related to SMI handler inputs. First, RSFuzzer learns about the input interfaces of the SMI handler, especially the hard-coded memory addresses used for passing inputs from kernel-space programs. Second, RSFuzzer learns about the format of the inputs related to each input interface. The format refers to both the nested object structures and the data types of the object fields. With the two types of input information acquired, RSFuzzer can generate valid test cases for testing a single SMI handler. However, only valid test cases are not enough for testing the deep logic inside SMI handlers. The reason is that some variables are shared among different SMI handlers as they are initialized by one SMI handler and used by multiple SMI handlers. If such variables are used in branch conditions, failing to resolve their values will lead to an early exit during the test case execution, leaving the deeper code untouched. Therefore, when fuzzing a single SMI handler, RSFuzzer employs a strategy to extract cross-handler variable information, which can reflect the producer-consumer relations among the SMI handlers for every cross-handler variable. This information is then used in cross-handler fuzzing mode to help decide the correct sequences of invoking multiple SMI handlers. In RSFuzzer, the greybox fuzzing engine starts with single-handler fuzzing mode at first. If the fuzzing engine cannot detect new basic-block coverage for a certain amount of time, RSFuzzer will consider that the testing of single SMI handlers has reached a bottleneck and it will switch into the cross-handler fuzzing mode. Whenever a seed found under the cross-handler fuzzing mode brings new insights about the input interfaces or input formats, RSFuzzer will switch into single-handler fuzzing mode and test the SMI handler related to the newly extracted insights. By adaptively switching between single and cross-handler fuzzing modes, RSFuzzer can substantially test the SMI handlers and detect deeply hidden vulnerabilities.

We implemented RSFuzzer as a hybrid greybox fuzzing framework for SMI handlers. We applied RSFuzzer to 16 UEFI firmware images from six original equipment manufacturers (OEMs). The results show that RSFuzzer can effectively detect previously-unknown vulnerabilities in SMI handlers of the UEFI firmware. In total, RSFuzzer found 65 SMI handler vulnerabilities (including 58 single-handler vulnerabilities and seven cross-handler vulnerabilities) and we have disclosed all these vulnerabilities to the corresponding vendors. Until now, 33 vulnerabilities were confirmed by the developers and 20 of them were fixed. Additionally, 14 vulnerabilities were assigned with CVE IDs. Moreover, through a comparison with the state-of-the-art interface-aware fuzzer, Syzgen [26], we found that RSFuzzer outperforms Syzgen in terms of both code coverage (617% more basic blocks on average) and vulnerability discovery (828% more vulnerabilities on average).

In summary, we make the following contributions:

- We proposed a hybrid greybox fuzzing technique called RSFuzzer, which can identify SMI handler input interfaces, recover the input formats and perform both single and cross-handler fuzzing. To the best of our knowledge, RSFuzzer is the first fuzzing technique which can detect cross-handler vulnerabilities.
- We implemented RSFuzzer as a hybrid greybox fuzzing framework and thoroughly evaluated it on 16 UEFI firmware images from six vendors. The evaluation results show that RSFuzzer can unveil deeply hidden vulnerabilities in SMI handlers and outperform state-of-the-art fuzzers.
- We detected 65 previously unknown vulnerabilities in the UEFI firmware of commercial-off-the-shelf (COTS) devices with RSFuzzer. We have responsibly disclosed all the vulnerabilities to the vendors. Until now, 33 of them are confirmed by the vendors and 20 are fixed. In addition, 14 CVE IDs have been assigned.
- We plan to open source RSFuzzer to facilitate open-science.

II. Background

SMM Driver and SMRAM. The Unified Extensible Firmware Interface (UEFI) is a widely used specification defining the software interface between the Operating System

[1]In this paper, we denote all the input files fed to the SMI handlers by fuzzers as test cases, and only those kept by the greybox fuzzer for subsequent mutations as seeds.

(OS) and the firmware. UEFI is compatible with several mainstream processor architectures such as x86, x86-64, ARM, etc. With x86 or x86-64 processors, UEFI can provide a secure execution environment for running in System Management Mode (SMM). The programs executed in SMM are called SMM drivers. Normally, SMM drivers are used to handle security critical functions such as power management, sensitive data access and system hardware control.

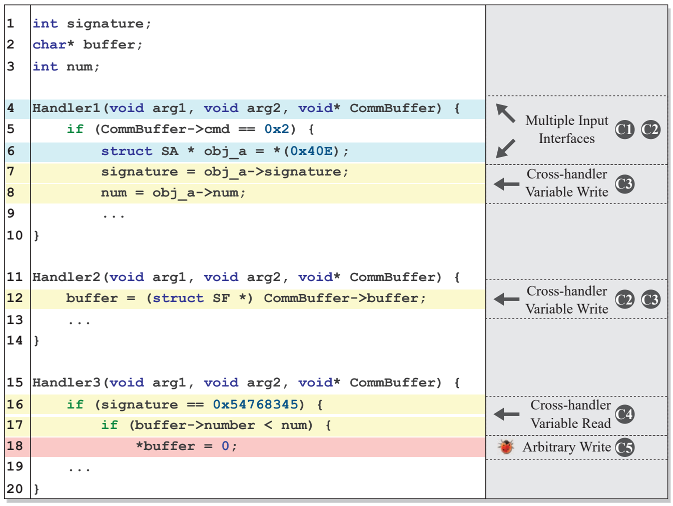
To facilitate secure execution, SMM drivers are loaded from the firmware into a distinct and isolated memory space called SMRAM during the boot up process. Once the boot up process is finished, the SMRAM is locked. After getting locked, SMRAM is not accessible from kernel-space or user-space programs but only SMM drivers can read/write SMRAM content. Therefore, SMM can provide an extra layer of protection when the operating system is compromised.

To allow SMM drivers to access and manipulate security critical data and hardware, SMM drivers are granted with ring -2 privilege. Therefore, SMM drivers have higher privilege than the kernel, which operates with ring 0 privilege. The ring -2 privilege enables SMM drivers to access the entire memory using a physical address.

SMI and SMI Handlers. Usually, each SMM driver contains three components: ❶ A set of *protocols* which work as vendor defined interfaces for communicating with other SMM drivers. ❷ A group of *SMI handlers*. A System Management Interrupt (SMI) handler represents a runtime service, each of which is assigned with a unique ID, namely SMI number. SMI handlers use a memory region called CommBuffer to receive data from kernel-space programs. ❸ An initialization function, which is used to register all the protocols and SMI handlers during the boot up process. After the system has booted up, the registered protocols and SMI handlers of all the SMM drivers in the UEFI firmware are stored in SMRAM.

SMM can be invoked via signaling an SMI, which can be generated by both hardware and software. When a kernel-space program needs to enter SMM, it first needs to prepare the content of the CommBuffer. Specifically, it needs to specify the SMI number to invoke a certain SMI handler. Then, the kernel-space program can signal an SMI to the processor. When the processor receives the SMI signal, it will suspend the normal execution of both kernel and user-space programs and start the execution of the corresponding SMI handler. The processor state is restored in SMRAM before executing the corresponding SMI handler. The resume from system management mode (RSM) instruction restores the processor to the state that it was in prior to a SMM interrupt.

SMI handlers carry out the main tasks for the SMM drivers, such as CPU overheating protection, power management, and SPI flash refreshing. Moreover, they serve as the only interfaces that can interact with kernel-space programs. In order to complete different tasks, SMI handlers need to receive and handle a variety of inputs with different formats. Since SMI handlers have complex input handling logic and are the only interfaces for feeding inputs to SMM drivers, they are the most suitable targets for leveraging fuzzing to detect vulnerabilities for SMM drivers.



```
1  int signature;
2  char* buffer;
3  int num;

4  Handler1(void arg1, void arg2, void* CommBuffer) {
5      if (CommBuffer->cmd == 0x2) {
6          struct SA * obj_a = *(0x40E);
7          signature = obj_a->signature;
8          num = obj_a->num;
9          ...
10 }

11 Handler2(void arg1, void arg2, void* CommBuffer) {
12     buffer = (struct SF *) CommBuffer->buffer;
13     ...
14 }

15 Handler3(void arg1, void arg2, void* CommBuffer) {
16     if (signature == 0x54768345) {
17         if (buffer->number < num) {
18             *buffer = 0;
19     ...
20 }
```

Multiple Input Interfaces C1 C2
Cross-handler Variable Write C3
Cross-handler Variable Write C2 C3
Cross-handler Variable Read C4
Arbitrary Write C5

Figure 1: The running example.

III. Threat Model

In this paper, we assume that the adversary has ring 0 privilege but no ring -2 privilege. This assumption means that the adversary can modify all the physical memory except the SMRAM. Thus, the adversary is able to feed inputs with controllable contents to the SMI handler via CommBuffer. If the SMI handler has a vulnerability, the adversary may exploit it with crafted inputs to hijack its control flow to escalate the privilege and launch malicious attacks. For example, SMI handler vulnerabilities can lead to leakage of private information, permanent lockdown of the computer, or even installation of UEFI Bootkit. Therefore, early detection of the vulnerabilities in SMI handlers is important for system security.

IV. Running Example

Figure 1 shows the code snippet of an arbitrary-write vulnerability. This happens when the program reaches line 18 . The value of buffer is controlled by user inputs. Therefore, attackers can write the value 0 to any memory address.

In order to fuzz an SMI handler, we first need to figure out its input interfaces to which we can feed test inputs. An SMI handler can have multiple input interfaces. For example, Handler1 in Figure 1 has two input interfaces: the CommBuffer variable and the hard coded memory address 0x40E. Note that although CommBuffer is officially specified by UEFI as the communication channel between SMI handlers and kernel-space programs, in practice, a lot of SMI handlers also receive data from kernel-space programs via hard coded memory addresses. Therefore, the first challenge **C1** is that ***some SMI handlers need to receive test inputs from multiple input interfaces***. After identifying the input interfaces, we can try to generate test inputs. However, to test the SMI handlers effectively, these test inputs need to be well-structured. For example, the variable buffer is a struct extracted from the input CommBuffer. If we cannot embed the content of buffer correctly in the test inputs generated for CommBuffer, we cannot sufficiently test the handlers which use the buffer variable due to input parsing failures. Therefore, the second challenge **C2** is that ***the test inputs for SMI handlers need to have certain formats***.

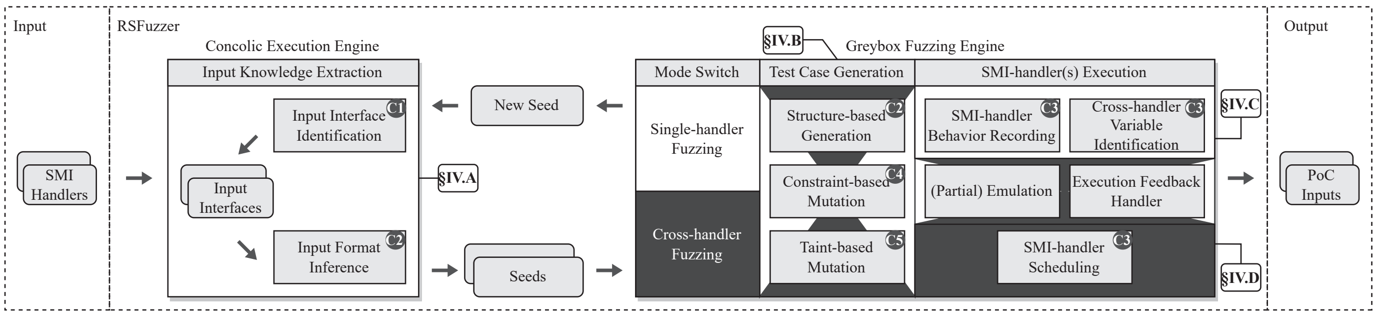


Figure 2: The overview of RSFUZZER.

In order to reach line 18 and trigger the vulnerability, we need to invoke Handler3 and pass the if conditions in line 16 and line 17. These if conditions involve the signature and buffer variables whose values are initialized in Handler1 and Handler2 respectively. Thus, we need to invoke Handler1 and Handler2 before invoking Handler3 to trigger the vulnerability. Therefore, the third challenge **C3** is that ***triggering certain vulnerabilities requires the involvement of multiple SMI handlers***. Since signature and buffer are used in multiple SMI handlers, we call them *cross-handler variables*.

Furthermore, two more challenges are hindering the detection of this vulnerability by fuzzing. For Challenge **C4**, ***some vulnerabilities are guarded by path constraints***. For example, the value of signature must equal 0x54768345 to reach line 18. For Challenge **C5**, ***some vulnerabilities are just silent corruptions which do not crash the SMI handlers***. For programs running with privilege higher than ring -1, writing to an arbitrary memory address is likely to crash the program and thus gets captured by the fuzzer. But SMI handlers have the ring -2 privilege to access any memory location even though sometimes they should not use this privilege. Hence, Handler3 will not crash on line 18 no matter how the value of buffer is tweaked. As a result, a fuzzer can never detect the vulnerability shown in Figure 1 by monitoring crashes.

We are now ready to make the following observations: ❶ To generate meaningful test cases, we need to learn the ***input knowledge***, including the input interfaces and the structure of each input interface. ❷ To sufficiently test the SMI handlers, we need to learn the ***cross-handler knowledge***, including the producer-consumer relations between SMI handlers and the cross-handler variable information. ❸ To increase code coverage, We need to resolve the path constraints. ❹ To detect more memory related vulnerabilities, we need the capability to detect silent corruptions. Based on these observations, we propose RSFUZZER, which can address all the five challenges.

V. APPROACH

Figure 2 shows the overview of RSFUZZER. The overall inputs are all the SMI handlers of a UEFI firmware image and the overall outputs are the Proof-of-Concept (PoC) inputs which can trigger crashes in the SMI handlers of the UEFI firmware. RSFUZZER executes in two modes: the ***single-handler fuzzing mode*** and the ***cross-handler fuzzing mode***.

In the ***single-handler fuzzing mode***, RSFUZZER selects an SMI handler, generates test cases based on seeds from the corresponding seed pool of the SMI handlers and executes the SMI handler with the test cases. Note that the initial seed pools are generated randomly by RSFUZZER. If a test case triggers the SMI handler to crash, it is reported as a PoC input. If a test case reports new basic-block coverage, it will be kept as a seed. Before a seed is added to the seed pool, RSFUZZER conducts input knowledge extraction on it.

Input knowledge extraction is to run concolic execution with the seed on the SMI handlers, where the input-related variables are symbolized and the other variables are concretized [2] (Section V-A). Input knowledge extraction runs in two steps: First, RSFUZZER identifies the input interfaces of the SMI handlers according to predefined rules. Second, for every identified input interface, RSFUZZER infers the corresponding input structure with the symbolic representation of the input. The extracted input knowledge includes both the input interfaces of the SMI handlers and the corresponding input structure for each input interface, and is stored together with the seed to facilitate structure-aware test case generation (Section V-B).

When executing the SMI handlers with a test case in the single-handler fuzzing mode, RSFUZZER conducts the cross-handler knowledge extraction by recording the variable-handling behaviors of the SMI handlers and identifying cross-handler variables (Section V-C). The extracted cross-handler knowledge is used for invoking the SMI handlers properly during the cross-handler fuzzing mode (Section V-D).

In the ***cross-handler fuzzing mode***, RSFUZZER first selects a cross-handler variable, then acquires a queue of SMI handlers related to this variable according to their producer-consumer dependencies. Then, RSFUZZER generates test cases based on seeds from the corresponding seed pool of each SMI handlers and executes the queue of SMI handlers with the corresponding test cases. RSFUZZER switches from single-handler fuzzing mode to cross-handler fuzzing mode when no new basic-block can be found in a given time limit. Empirically, we set this time limit as 5 minutes. RSFUZZER switches from cross-handler fuzzing mode to single-handler fuzzing mode whenever it finds a seed and extracts new knowledge from the seed. After switching to the single-handler fuzzing mode from the cross-handler fuzzing mode, RSFUZZER will select the SMI handlers related to the newly extracted knowledge for fuzzing. By switching between the

[2]During the execution of SMI handlers, some variables are not related to the input but they can affect the execution. We need to use concrete values extracted from the execution of the seeds for these variables.

two modes, RSFUZZER can substantially test the SMI handlers of a UEFI firmware image.

The key novelty of RSFUZZER for a UEFI firmware image is to learn input format and function relations incrementally by switching between single and multi-handler fuzzing modes.

A. Input Knowledge Extraction

Input Interface Identification. SMI handlers can receive inputs from both CommBuffer and hard-coded addresses. For CommBuffer, the solution is straightforward. According to the UEFI specification, CommBuffer is always the third parameter for SMI handlers. Therefore, we do not need to identify the actual memory address of CommBuffer. Instead, we can allocate a new memory region, store the test data inside and replace the third parameters of the SMI handlers during fuzzing to test the CommBuffer handling logic. For hard-coded addresses, we use a heuristic-based approach to identify them. Although CommBuffer is the suggested channel for communication between SMI handlers and kernel space programs, according to [27], the lower 1 MB of the physical memory is reserved to provide fixed resources for working with option ROMs. This memory region is outside the SMRAM and is not part of the CommBuffer. For example, the value in the 0x40E address contains the segment number of a memory region called the Extended BIOS Data Area (EBDA). Therefore, if a hard-coded address in the lower 1 MB of the physical memory is dereferenced by an SMI handler, it will be treated as an input interface by RSFUZZER.

Input Format Inference. Since the inputs related to the input interfaces are struct variables, we need to infer their formats in order to generate meaningful test inputs. In order to recover the input formats, we need two types of knowledge: The data type knowledge and the nested structure knowledge.

The data type knowledge refers to the data types of the variables inside the structures. In RSFUZZER, we separate the variable types into two categories and provide two corresponding strategies for identifying them. The two categories are the pointer type and the basic arithmetic types, where the basic arithmetic types include int, char, float and double. The identification of pointer type variables is straightforward: If a variable is eventually used for memory address dereference through its def-use chain, then it is considered as a pointer. The identification of the basic arithmetic type variables is based on heuristics: If a variable is used for arithmetic operations with another variable of a known basic arithmetic type, then this variable is considered to have the same data type as the known variable. For example, if a variable is used to compare with an integer, then this variable is considered to have the data type of int. Note that the purpose of RSFUZZER to identify the types of the basic arithmetic type variables is only to know their sizes. Therefore, RSFUZZER can tolerate imprecise type identifications as far as the size of the variable is correct.

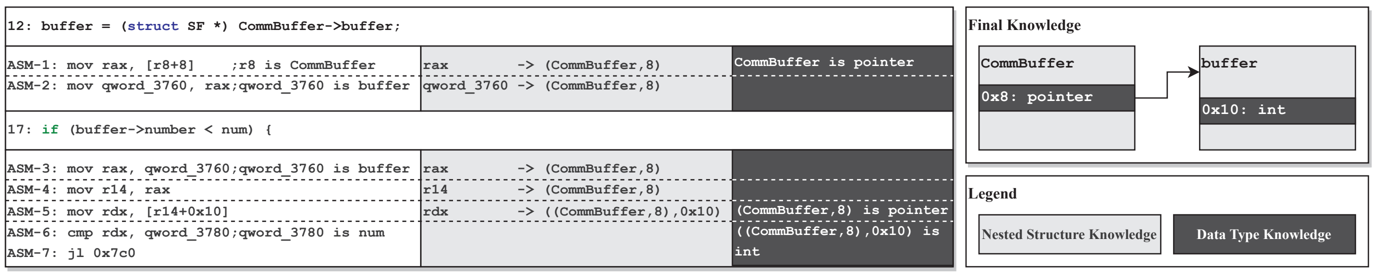
The nested structure knowledge refers to how the structures reside in or contain other structures. Usage of nested structures is common in the inputs of SMI handlers. Therefore, inferring the nested structures of the inputs is important for recovering their formats. For the nested structures, we have two observations: ❶ A pointer variable, which points to a structure, is mostly used in indirect memory accesses (i.e., [base + index * scale + displacement]). ❷ An upper level structure always contains a pointer variable that points to a lower level structure in the nested structures framework. Based on these observations, we propose an approach to recursively recover the nested structures of the SMI handler inputs. The overall input for nested structure knowledge extraction is the concolic execution trace of an SMI handler with a seed input and the overall output is a hash map called sym_ptr in which the symbols of pointers are keys and the corresponding pointers are values. For every instruction in the execution trace, RSFUZZER updates sym_ptr according to the operators and the symbolic expression of the operands.

Algorithm 1 depicts how sym_ptr is updated for every instruction. In Algorithm 1, two details need to be emphasized: ❶ RSFUZZER simplifies the expression of indirect memory accesses from [base + index * scale + displacement] into [base + offset] by calculating the value of index * scale + displacement. ❷ The function get_original_sym is recursively called so that RSFUZZER can not only resolve the aliases of the pointers but also clearly track the base pointer of the root structure. After getting updated by every instruction in the execution trace, sym_ptr contains the nested structure knowledge of how the structures are connected and which structure is the root structure. By combining the data type knowledge and the nested structure knowledge, RSFUZZER can recover the format of the inputs for every input interface.

Running Example Explanation. We use the running example in Figure 1 to demonstrate how RSFUZZER performs input format inference for CommBuffer with the execution traces of Handler2 and Handler3. The lines of code for handling CommBuffer related data are line 12 and line 17 of Figure 1. Figure 3 shows how RSFUZZER recovers the format of CommBuffer by analyzing the instructions in the execution traces. In Figure 3, the assembly codes related to line 12 are ASM−1 – ASM−2. Similarly, the assembly codes related to line 17 are ASM−3 – ASM−7. Assume RSFUZZER already knows that r8 holds the pointer of CommBuffer. After analyzing ASM−1, RSFUZZER will get the information that rax is now pointing to [CommBuffer+8] and CommBuffer is a pointer. After analyzing ASM−2 – ASM−4, RSFUZZER will get the information that r14 is an alias of [CommBuffer+8]. After analyzing ASM−5, RSFUZZER will get the information that [CommBuffer+8] is a pointer since it is used for memory dereference. Lastly, after analyzing ASM−6, RSFUZZER will know that [[CommBuffer+8]+0x10] is an integer variable since it is used to compare with an integer. To this end, the final knowledge learned by RSFUZZER is that the variable at the offset 0x8 of the first structure is a pointer pointing to the second memory region (i.e., the structure buffer in this example) and the variable at the offset 0x10 of the second structure is an integer.

B. Test Case Generation

RSFUZZER combines the generation-based method with the mutation-based method to generate valid seeds and test cases based on the results of the input knowledge extraction.



```
12: buffer = (struct SF *) CommBuffer->buffer;

ASM-1: mov rax, [r8+8]     ;r8 is CommBuffer       rax         -> (CommBuffer,8)            CommBuffer is pointer
ASM-2: mov qword_3760, rax;qword_3760 is buffer    qword_3760 -> (CommBuffer,8)

17: if (buffer->number < num) {

ASM-3: mov rax, qword_3760;qword_3760 is buffer    rax         -> (CommBuffer,8)
ASM-4: mov r14, rax                                r14         -> (CommBuffer,8)
ASM-5: mov rdx, [r14+0x10]                         rdx         -> ((CommBuffer,8),0x10)   (CommBuffer,8) is pointer
ASM-6: cmp rdx, qword_3780;qword_3780 is num                                              ((CommBuffer,8),0x10) is
ASM-7: jl 0x7c0                                                                           int
```

Figure 3: An example of input format inference for CommBuffer in Figure 1

Algorithm 1: Nested Structure Knowledge Extraction

```
Input: trace: The execution trace.
Output: sym_ptr: The hash map storing symbols and their
        related pointers.
1  Def infer_structure(trace):
2      sym_ptr ← {(r8,(CommBuffer,0))}
3      for inst ∈ trace do
4          src_op, dst_op ← get_operands(inst)
5          use_sym ← get_src_symbol(src_op,
             sym_ptr)
6          def_sym ← get_dst_symbol(dst_op,
             sym_ptr)
7          sym_ptr[def_sym] ← use_sym
8
9  Def get_src_symbol(op, sym_ptr):
10     if op is dereference then
11         (base, offset) ← get_base_offset(op)
12         base ← get_original_sym(base, sym_ptr)
13         variable ← (base, offset)
14     else
15         variable ← op
16     return get_original_sym(variable, sym_ptr)
17
18 Def get_dst_symbol(op, sym_ptr):
19     if op is dereference then
20         (base, offset) ← get_base_offset(op)
21         base ← get_original_sym(base, sym_ptr)
22         variable ← (base, offset)
23     else
24         variable ← op
25     return variable
26
27 Def get_original_sym(variable, sym_ptr):
28     if variable ∈ sym_ptr.keys() then
29         variable ← sym_ptr[variable]
30         return get_original_sym(variable,
             sym_ptr)
31     else
32         return variable
```

Structure-based Seed Generation. Whenever a new piece of knowledge is acquired during the input knowledge extraction phase, RSFUZZER will allocate the well-formatted memory according to the newly learned knowledge, fill the random data in the allocated memory to generate a new seed and put it into the seed pool.

Algorithm 2 describes how RSFUZZER allocates memory for a new seed. It traverses the hash map sym_ptr generated by Algorithm 1 and allocates memory for the pointers (line 5). The allocation is performed recursively. If a structure is not allocated or insufficient allocated (line 11 and line 18), then RSFUZZER allocates the memory based on the offset recorded by its corresponding element in sym_ptr, guaranteeing the sufficient space for the later accessing. The address of the newly allocated structure is recorded in the address offsetting from the base address of its upper level structure (line 20). If an input interface is met, structs records the address of its corresponding memory newly allocated.

Last but not least, for SMI handlers with multiple input interfaces, RSFUZZER generates a separate seed input for every input interface and stores them together as one conceptual seed in the seed pool.

Algorithm 2: Structure Allocation

```
Input: sym_ptr: The hash map storing symbols and their
       related pointers.
Output: structs: The hash map storing input interfaces and
        the addresses of their allocated memory.
1  Def recover_structs(sym_ptr):
2      structs = {}
3      structs_size = {}
4      for sym ∈ sym_ptr.values() do
5          if is_pointer(sym) then
6              recover_recursively(sym, structs)
7
8  Def recover_recursively(sym, structs):
9      base ← sym.base
10     if is_input_interface(base) then
11         if structs_size[base]<sym.offset then
12             structs[base] ←
                 memory_alloc(sym.offset)
13             structs_size[base] ← sym.offset
14             return (structs[base], sym.offset)
15     else
16         if is_pointer(base) then
17             (mem, off) ←
                 recover_recursively(base, structs)
18             if structs_size[mem + off]<base.offset
                 then
19                 temp_mem ←
                     memory_alloc((base.offset))
20                 mem_write(mem + off, temp_mem)
21                 structs_size[mem + off] ← base.offset
22                 return (temp_mem, base.offset)
```

Constraint-based Test Case Mutation. The structure-based generation is not enough for generating proper test cases which can reach the deep logic of SMI handlers. The reason is that most SMI handlers perform sanity checks against the variables of the input structures and malformed inputs will be rejected early by the SMI handlers. As discussed in Section V-A, RSFUZZER performs concolic execution with a seed before putting it into the seed queue. Besides the information needed for input knowledge extraction, new seeds which can pass

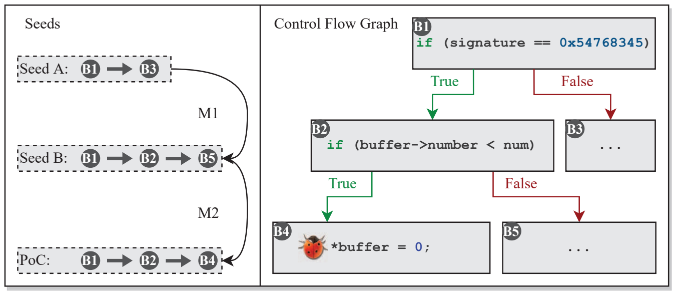


Figure 4: An example of constraint-based test case mutation for Handler3 in Figure 1

sanity checks are also generated during the concolic execution. However, sometimes the SMT solver may take too long or fail to solve the constraints. Therefore, we still need to apply mutations to the seeds to further explore different program states of the SMI handlers.

Instead of mutating all the variables in the input structures, RSFUZZER only mutates the variables that can influence the branching conditions. In RSFUZZER, a branch is marked as *touched* only if both of the true and false branches are executed. Given a seed input, RSFUZZER will mutate the variables which are related to untouched branches with the following mutation strategies.

- **M1**: replace the specific variable with the value used in the branch condition. This strategy is designed to pass equality comparisons (e.g., a == b).
- **M2**: replace the specific variable with variations of the value used in the branch condition with minor changes. This strategy is designed to pass inequality comparisons (e.g., a > b or a < b).

Figure 4 shows an example of how RSFUZZER performs constraint-based test case mutation. In Figure 4, the control flow graph of Handler3 of the running example is shown on the right and the test cases generated by RSFUZZER are on the left. Assuming Seed A is an initial seed which is generated randomly by RSFUZZER and it can only cover basic block B1 and basic block B3. During the concolic execution, RSFUZZER finds that the variable signature is related to the untouched branch of B1. Assuming the SMT solver takes too long to find a satisfiable input for passing the condition, RSFUZZER will apply mutation strategy M1 to replace the value of signature with the value 0x54768345. After mutation, RSFUZZER can generate Seed B, which can pass the branch condition of B1 but cannot pass the branch condition of B2. Similarly, by conducting concolic execution with Seed B, RSFUZZER will notice that B2 contains an untouched branch and buffer−>number is the related variable. This time, RSFUZZER will apply the mutation strategy M2 because the condition is an inequality comparison. By applying M2, RSFUZZER may replace the value of buffer−>number with the value num−1, which leads to the PoC input.

Taint-based Test Case Mutation. Since SMI handlers have ring -2 privilege, they can access entire physical memory. Therefore, unwanted memory access of SMI handlers does not trigger crashes and we need sanitization techniques to capture the silent corruptions. Inspired by the heuristics discussed in [28], we propose a taint-based test case mutation and memory hardening strategy to turn silent corruptions into crashes. First, before fuzzing, RSFUZZER allocates a piece of memory and marks it as unexecutable, unreadable and unwritable even for SMI handlers. This piece of memory serves as the red zone for SMI handlers. Then, during the concolic execution, for each instruction, RSFUZZER takes the following three steps to analyze it: ❶ RSFUZZER checks whether the analyzed instruction is a memory-related instruction (e.g., call to memory manipulation functions such as memcpy, pointer dereference instruction, etc.); ❷ If the instruction is memory-related, RSFUZZER will utilize the symbolic expression of the pointer used in the analyzed instruction to check whether it is user controllable or not; ❸ If the pointer used in the analyzed instruction is user controllable, RSFUZZER will mark the pointer as potentially dangerous. Last, during fuzzing, RSFUZZER will mutate the values of the potentially dangerous pointers so that they will point to the red zone memory. By purposefully making the pointers point to the red zone, RSFUZZER can convert most of the silent corruptions into crashes and capture them.

Algorithm 3: Cross-handler Knowledge Extraction

```
Input: trace: The execution trace.
Output: chvar_pro: The hash map storing cross-handler
        variables and their related producer handlers.
Output: chvar_con: The hash map storing cross-handler
        variables and their related consumer handlers.
1  Def extract_cross_handler_knowledge(trace):
2      chvar_pro ← {}
3      chvar_con ← {}
4      for inst ∈ trace do
5          (l_var, r_var) ← get_variables(inst)
           /* Retrieve the handler that
              contains the current instruction
              */
6          handler ← get_inst_handler(inst)
7          update_live_var(inst, chvar_pro, chvar_con)

8          if l_var ≠ Null then
9              lmem ← get_access_mem(l_var)
10             chvar_pro[lmem] ∪← {handler}
11         if r_var ≠ Null then
12             rmem = get_access_mem(r_var)
13             if rmem ∈ chvar_pro then
14                 chvar_con[rmem] ∪← {handler}
```

C. Cross-handler Knowledge Extraction

When executing the SMI handlers with a test case in the single-handler fuzzing mode, RSFUZZER conducts the cross-handler knowledge extraction by recording the variable-handling behaviors of the SMI handlers and identifying cross-handler variables. The extracted knowledge is used for scheduling the SMI handlers in the cross-handler fuzzing mode. The cross-handler variables are the variables used by multiple SMI handlers while the variable-handling behaviors of an SMI handler refer to if the SMI handler is the producer or the consumer of a particular cross-handler variable.

The following information is crucial for a fuzzer to perform fuzzing tests on cross-handler code segments: ❶ *Cross-handler variable writers*: A set of SMI handlers that write to the cross-handler variable involved in the cross-handler code segments.

❷ *Cross-handler pointers*: For each cross-handler variable writer, it is necessary to know which pointers the target cross-handler variable is transformed or directly copied from. This is because cross-handler variables that are directly transformed or copied from input data are more likely to affect the execution state of the cross-handler code segment.

To this end, we need to identify the cross-handler variables involved in the cross-handler code segments. We find that cross-handler code segments have three key features: (1) the cross-handler code segments always access the same memory location, (2) at least one code segment can write to the memory location (3) the cross-handler variable can be alive across SMI handlers. The complete algorithm used to identify the cross-handler variables is depicted in Algorithm 3. We use these three features to identify candidate cross-handler code segments and cross-handler variables for fuzzing. More specially, we also record the birth time of a variable when the variable is created and update the life cycle of the variable (line 6) to distinguish the multiple variables that occupy the same memory location (e.g., the same stack memory being reused by different functions). For each instruction, we take the following steps to determine whether the instruction contains a cross-handler variable. First, we retrieve the memory location accessed by the instruction (line 9 and line 12). Second, for read operation, the algorithm checks whether the retrieved memory location is occupied by a variable created in another SMI handler (line 13).

D. SMI Handler Scheduling

RSFuzzer switches from single-handler to cross-handler fuzzing mode when no new basic-block can be found in a given time limit (which defaults to five minutes). In cross-handler fuzzing mode, RSFuzzer schedules SMI handlers according to their producer-consumer relations on the cross-handler variables.

For each accessed cross-handler variable, RSFuzzer retrieves the cross-handler variable's producer-consumer dependency by using the extracted producer-consumer handler information. Given these dependencies, RSFuzzer can guide the SMI handler scheduling and find more interesting test cases concerning the cross-handler context. A cross-handler variable should be produced by an SMI handler before it be consumed by another SMI handler. Therefore, RSFuzzer executes the SMI handler that produces the cross-handler variable for each cross-handler variable. Then, RSFuzzer deals with the consumers SMI handler in random order. We can bypass a sanity check against a cross-handler variable by ❶ mutating input bytes used in analyzed sanity check and replacing them imprecisely with expected values extracted from the analyzed sanity check; ❷ executing the cross-handler variable writer with the mutated test case; ❸ executing the SMI handler that contains the analyzed sanity check with the recorded test case that can reach the analyzed sanity check.

In the cross-handler fuzzing mode, on seeing a seed, RSFuzzer will also conduct knowledge extraction from the seed to learn additional SMI behavior and input knowledge. Our approach can improve the quality of the test cases during the subsequent fuzzing test due to two reasons. ❶ The first two steps can guarantee that it is more likely to change the targeted cross-handler variable with a value that is computed from the mutated input bytes. ❷ The third step can guarantee that the subsequent fuzzing test will reach the analyzed sanity check.

As shown in running example (Figure 1), when executing the Handler3 in cross-handler fuzzing mode, RSFuzzer first identifies the signature as a cross-handler variable based on the following observations: ❶ Both Handler1 and Handler3 access the same memory location occupied by signature; ❷ Handler1 produces the signature; ❸ the signature produced by Handler1 is still alive when executing the Handler3. RSFuzzer then recognizes a valid sequence of SMI handlers that includes both Handler1 and Handler3 by leveraging the recorded producer-consumer dependency of signature. In addition, RSFuzzer is able to conduct constraint-based mutation on signature to bypass the sanity check at line 16 by leveraging the extracted input knowledge and constraint information. Following the same procedure, RSFuzzer is able to identify the cross-handler variable buffer and set up the valid sequence of SMI handlers (Handler2 and Handler3 in turn) by leveraging the recorded producer-consumer dependency of buffer.

VI. Implementation

We implement a prototype of RSFuzzer by combining AFL++, a customized UEFI emulator based on Qiling [29], and Triton [30]. The concolic execution engine is built by integrating Triton into our customized UEFI emulator. The greybox fuzzing engine is built on top of AFL++ where the coverage feedback needed is provided by our customized emulator. We trace jump instructions during emulation to collect basic-block coverage information. Instead of introducing every implementation detail, we focus on explaining the design of our customized UEFI emulator which can perform partial emulation for SMI handlers to achieve faster execution speed.

Partial Emulation. An UEFI firmware image contains lots of low-level drivers handling the boot and runtime phases. However, not all are required by SMI handlers. Emulating all the drivers costs extra computation resources and may cause compatibility issues since some drivers may have specific hardware dependencies. Thus, partial emulation can avoid wasting extra resources on other drivers by focusing only on the drivers related to our target SMI handlers. To emulate the SMM drivers at boot phase, we dispose of a high-level abstraction of the SMM-drivers-related UEFI modules to replace the real boot steps on the mainboard. As the standard boot processes are defined in UEFI specification clearly, we can deploy a general boot phase SMM drivers emulation platform by leveraging Qiling [29] to initialize the emulated SMM drivers and support the fuzzing test among different SMI handlers at runtime.

For the runtime services of SMI handlers and related SMM drivers in the virtual SMRAM region, we reuse the original drivers instead of understanding and re-implementing the required dependencies due to the following facts. (1) An SMI handler represents a runtime service implemented by calling exported functions of runtime dependence drivers. (2) SMI handlers' functionalities could be customized by the vendors according to the specific hardware feature. In other words, the exported functions of runtime dependence drivers could be customized by the vendors according to the specific hardware feature. (3) The vendor-specific customized code is more likely to be involved in vulnerable flow.

It is commonly believed that runtime dependencies have two key features [27]: (1) the driver allocates memory space for protocol instance and initializes the instance by filling in the function pointers and the other data declared in the instance's structure; (2) the driver exports functions by registering the protocol instance to the protocol manager through a standard UEFI API (i.e., InstallProtocol [27]). By following [31], we use these two features to find candidate runtime dependency for emulating the targeted SMI handler. For hardware I/O, if an input value affects execution by appearing in path constraints, then we symbolize this value and use symbolic execution to get a needed value. Otherwise, we use a random value.

VII. Evaluation

A. Experiment setup

Dataset Preparation. We obtained 16 UEFI firmware images (in the latest version when tested) from public official websites of 6 popular OEMs including HP, Lenovo, Asus, Dell, Intel, and Gigabyte. These firmware images cover major computer types (including personal computer, Desktop, Embedded Mini personal computer, and workstation).

Evaluated Techniques. We use Syzgen and SPENDER as the baseline techniques for comparison. Syzgen is an interface-aware fuzzing tool for detecting vulnerabilities hidden in closed-source macOS drivers. It automates the generation of interface templates for closed-source macOS drivers. To recover the nested objects, it monitors the key internal API invoked in macOS drivers to perform a deep copy that copies nested objects from user space to kernel space. It infers dependence between interfaces by analyzing execution traces collected from existing applications. We implemented the method of Syzgen based on the RSFuzzer framework to identify the nested object and infer dependence between SMI handlers, replacing corresponding knowledge extraction method (Section V-A). Note that we also implemented the memory hardening strategy used for the taint-based mutation in RSFuzzer (Section V-B). The rationale is that we want both techniques to have the same awareness of silent corruptions so that we can focus on the comparison of input knowledge extraction capabilities of these two techniques. If we do not implement the memory hardening strategy for Syzgen, it can hardly capture any vulnerabilities in SMI handlers. SPENDER is a static framework that detects specific taint-style vulnerabilities hidden in SMI handlers. Since SPENDER is a dedicated technique for SMI handlers and it can be applied to closed-source targets, we include it as a baseline technique.

Evaluation criteria. We use two criteria (i.e., vulnerability discovery and code coverage) to evaluate the effectiveness of RSFuzzer. For code coverage, we consider mainly block coverage (i.e., the number of unique block hits). For vulnerability discovery, we track the number of unique vulnerabilities and the detailed vulnerability types detected by RSFuzzer.

Experiments Settings. For each UEFI firmware image, we repeated each experiment 10 times with time budgets of 24 hours to reduce the effect of fuzzing randomness. The lines of plots (Figure 6) are average basic block numbers. Moreover, we use p-values to measure the statistical significance of the results.

Table I: Our UEFI firmware dataset and evaluation results.

Characteristics of Firmware				#Vulnerabilities		
OEM	Type	Firmware Model	#SMI Handlers	Tot	Single Handlers	Cross Handlers
HP	Desktop	HP Obelisk 875	39	6	5	1
HP	WorkStation	HP Z2 Mini G4	71	4	3	1
HP	WorkStation	HP Z440	39	7	7	0
HP	PC	HP 20-c000	38	3	2	1
Lenovo	PC	Thinkpad X1 Fold	52	4	4	0
Lenovo	Desktop	ThinkStation S30	13	2	2	0
Lenovo	Desktop	Thinkstation P900	38	4	3	1
Lenovo	Desktop	ThinkCentre M700	50	2	2	0
ASUS	PC	ASUS P453UJ	33	5	4	1
ASUS	Mini PC	ASUS UN65U	35	5	5	0
Dell	PC	Alienware X51 R3	43	9	8	1
Dell	PC	Alienware 13 R3	38	5	4	1
Intel	Mini PC	Intel NUC8i3CYSM	40	3	3	0
Intel	Mini PC	Intel NUC10i7FN	49	3	3	0
Gigabyte	PC	Z690 GAMING X	24	2	2	0
Gigabyte	PC	X570 GAMING X	44	1	1	0
Total	-	-	646	65	58	7

Experiment Environment. We run RSFuzzer on a PC with a dual-core Intel Core i5-7260U CPU @2.20GHz, 16 GB of RAM, and Ubuntu 18.04.

B. Vulnerability Discovery

To assess the effectiveness of RSFuzzer, we run it against the UEFI firmware images in our dataset to find new vulnerabilities. For all tested firmware, RSFuzzer was able to automatically identify the input channels of the SMI handlers, and feed fuzzer-generated input data to the analyzed SMI handler. To summarize, RSFuzzer found 65 new vulnerabilities, 20 of which were confirmed by the corresponding vendors. Of these, 14 were already confirmed as Common Vulnerabilities and Exposures (CVEs) and were fixed by the vendors, as shown in Table II. It is worth noting that many firmware images (especially those from Intel, Dell, ASUS, AMD, and Lenovo) have been extensively researched by security researchers and have been found to have many vulnerabilities. Despite this, RSFuzzer was able to discover a significant number of new vulnerabilities, demonstrating its ability to generate highly structured input and drive fuzzing tests into deeper paths under complex sanity checks.

In addition, Table I shows two categories of vulnerabilities: (i) single-handler vulnerabilities, which were detected using the single-handler knowledge extraction module; and (ii) cross-handler bugs, which were detected using the cross-handler knowledge extraction module. In total, RSFuzzer found 58 single-handler vulnerabilities and 7 cross-handler vulnerabilities, demonstrating its effectiveness in detecting both types of vulnerabilities.

As shown in Table II, RSFuzzer has discovered 6 classes of vulnerabilities in SMM, including improper input validation, out-of-bound write, buffer overflows, use of uninitialized

[3]Due to the fact that the vulnerable platform is nearing end of life, the product team is unable to allocate resources for remediations for this issue.

Table II: Statistics of vulnerabilities IDs found by RSFUZZER.

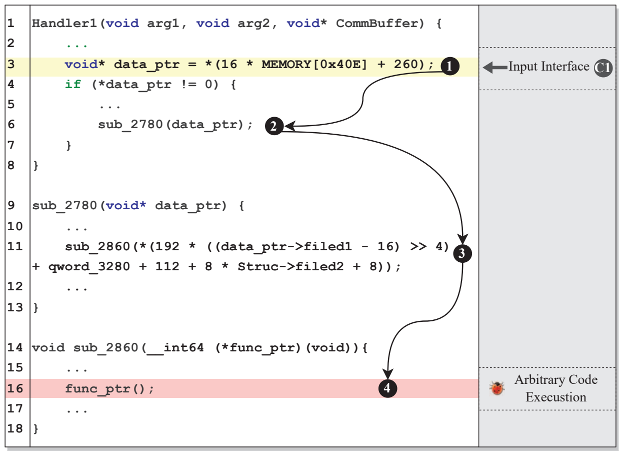
CVE ID	CVSS Score(v3)	Vendor	Type	Vulnerability Status
CVE-2021-41289	7.1	ASUS	Buffer Overflow	Confirmed, Fixed
CVE-2022-21933	7.8	ASUS	Input invalidation	Confirmed, Fixed
CVE-2021-3843	6.7	Lenovo	Input invalidation	Confirmed, Fixed
CVE-2021-3719	6.7	Lenovo	Input invalidation	Confirmed, Fixed
CVE-2021-3661	8.1	HP	Buffer Overflow	Confirmed, Fixed
CVE-2021-3439	7.8	HP	Out of Bound Write	Confirmed, Fixed
CVE-2021-3809	8.8	HP	Use After Free	Confirmed, Fixed
CVE-2021-3808	8.8	HP	Use After Free	Confirmed, Fixed
CVE-2022-24416	8.2	Dell	Input Validation	Confirmed, Fixed
CVE-2021-36343	7.5	Dell	Out of Bound Write	Confirmed, Fixed
CVE-2022-24415	8.2	Dell	Input Validation	Confirmed, Fixed
CVE-2021-36323	7.5	Dell	Input Validation	Confirmed, Fixed
CVE-2021-36324	7.5	Dell	Input Validation	Confirmed, Fixed
CVE-2021-36325	7.5	Dell	Input Validation	Confirmed, Fixed
PSRC-15616	-	Dell	Missing Initialization	Confirmed, EOF[3]
PSRC-15772	-	Dell	Pointer Dereference	Confirmed, EOF
PSRC-15773	-	Dell	Out of Bound Write	Confirmed, EOF
PSRC-15774	-	Dell	Out of Bound Write	Confirmed, EOF
PSRC-15775	-	Dell	Buffer Overflow	Confirmed, EOF
PSRC-15776	-	Dell	Buffer Overflow	Confirmed, EOF

variables, untrusted pointer dereference, and use after free. Among these, improper input validation is the most common type of vulnerability. This occurs when SMI handlers fail to validate input properly, allowing attackers to exploit the vulnerability and launch attacks that could result in altered control flow, arbitrary memory access, or arbitrary code execution.

Furthermore, to demonstrate the practical security impact of these bugs, we acquired two of the affected devices and successfully crafted PoCs for two of the vulnerabilities, e.g., an kernel-space program successfully gains SMM privilege and runs arbitrary code in the SMRAM, enabling malicious capabilities such as modifying the content of SPI Flash normally restricted to SMM, and to perform permanent malicious attacks. Since it was not feasible for us to acquire all affected devices, the POCs of the remaining vulnerabilities are crafted using our emulator and we confirmed that 18 other previously unknown vulnerabilities were also exploitable. As is shown in Table II, for all the confirmed vulnerabilities with assigned CVSS v3 scores, the average CVSS v3 base score of the reported vulnerabilities is 7.7 (except those CVE entries reserved by vendors), indicating the high security impact of the reported vulnerabilities.

Finally, we examine a representative SMM driver (with reverse-engineered code snippet) that contains an SMM-related vulnerability, detected by RSFUZZER.

Case Study. We studied a vulnerability found by RSFUZZER in detail. This vulnerability, which was caused by unexpected improper input validation, was found in an SMI handler of the Intel NUC UEFI firmware. In the experiment, RSFUZZER raised an alert when attacker-controlled data was propagated and dereferenced in an SMI handler, thanks to the memory hardening during taint-based mutation (Section V-B). Figure 5 shows the flow of triggering this vulnerability. ❶ At line 3, the SMI_handler1 reads the untrusted data pointer from the hard-coded memory address (0x40E). ❷ At line 6, the Handler1 invokes the function (sub_2780) that accepts data_ptr as the first argument. ❸ At line 11, sub_2780 retrieves a function pointer from the data_ptr and invokes the function (sub_2860). ❹ At line 16, sub_2860 invokes the function pointed by the function pointer argument and an arbitrary code execution vulnerability can happen if the data_ptr is malformed. The lessons learned from this case are: ❶ Although CommBuffer is the officially specified communication channel between SMI handlers and kernel-space programs, in practice, an SMI handler can receive data from kernel-space programs via hard-coded memory address. Intel Excite is not able to detect this vulnerability due to the fact that it can only mutate CommBuffer. ❷ To trigger the vulnerability, the test input needs to be well-structured. Therefore, random mutating is not enough to trigger this vulnerability.

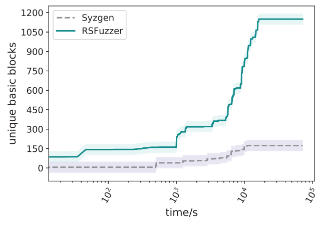


```
1  Handler1(void arg1, void arg2, void* CommBuffer) {
2      ...
3      void* data_ptr = *(16 * MEMORY[0x40E] + 260);
4      if (*data_ptr != 0) {
5          ...
6          sub_2780(data_ptr);
7      }
8  }

9  sub_2780(void* data_ptr) {
10     ...
11     sub_2860(*(192 * ((data_ptr->filed1 - 16) >> 4)
   + qword_3280 + 112 + 8 * Struc->filed2 + 8));
12     ...
13 }

14 void sub_2860(__int64 (*func_ptr)(void)){
15     ...
16     func_ptr();
17     ...
18 }
```

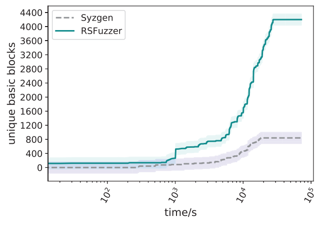
Figure 5: A vulnerability found in the Intel NUC.

C. Comparison with baseline

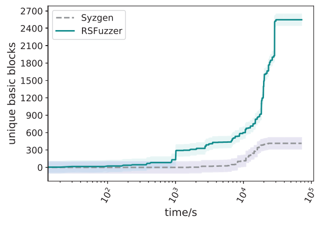
To evaluate the performance of the test cases generated by RSFUZZER, we conducted a fuzzing experiment to compare RSFUZZER with Syzgen. We tracked the growth trend of basic block coverage and Figure 6 shows the code coverage generated by each fuzzer for 16 firmware images. The result shows that RSFUZZER covers significantly more unique basic blocks at a faster pace than Syzgen in all the firmware images since the lower bound of the 95% confidence interval of the coverage of RSFUZZER is higher than the upper bound of the 95% confidence interval of the coverage of SyzGen. Specifically, as shown in Figure 6, RSFUZZER outperforms Syzgen in terms of coverage for all firmware images. Based on Mann-Whitney U-test, we found that all p-values are smaller than 5.00e-2, indicating statistical significance. RSFUZZER improves the basic block coverage by up to 617%. Syzgen is ineffective in our setting due to the following facts. ❶ To recover nested objects, Syzgen keeps track of the internal API that creates a deep copy of the nested object. However, in UEFI firmware, there is not the API function that performs deep copy of nested objects from kernel space to SMRAM. ❷ Syzgen focuses on identifying return and argument relationship between different interfaces. In terms of unique vulnerabilities, as shown in Figure 7, RSFUZZER detected a total of 65 unique vulnerabilities, while Syzgen detected only 7. This represents an improvement of 828% in the detection of unique vulnerabilities by RSFUZZER.



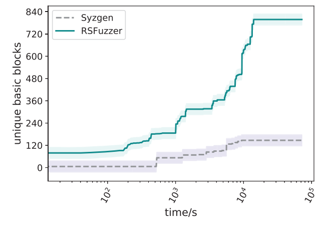
(a) HP Obelisk 875



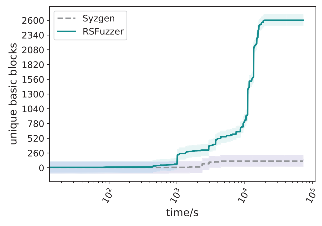
(b) HP Z2 Mini G4



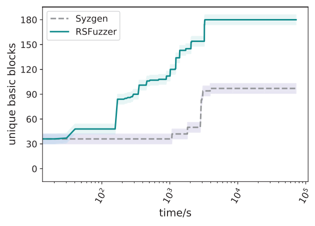
(c) HP Z440



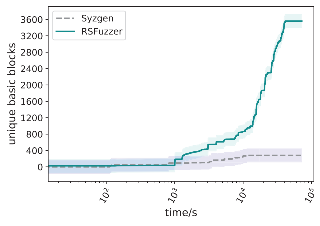
(d) HP 20-c000



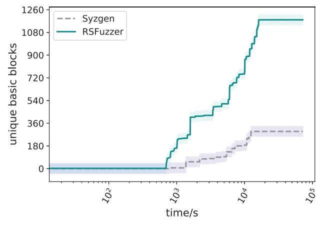
(e) Thinkpad X1 Fold



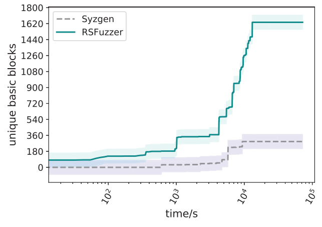
(f) ThinkStation S30



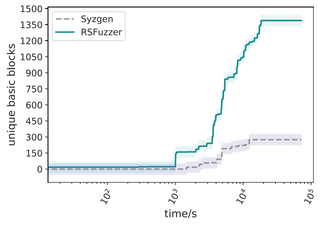
(g) Thinkstation P900



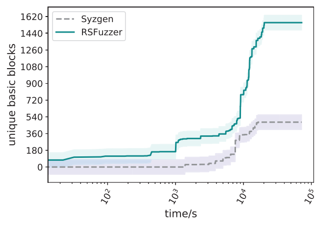
(h) ThinkCentre M700



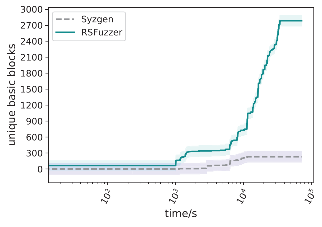
(i) ASUS P453UJ



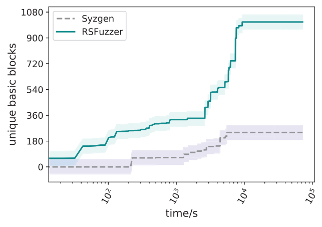
(j) ASUS VivoMini UN65U



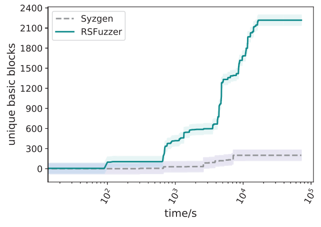
(k) Alienware X51 R3



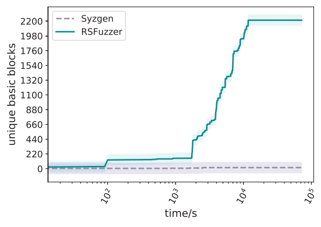
(l) Alienware 13 R3



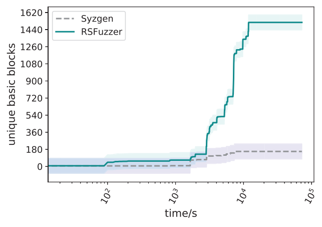
(m) Intel NUC8i3CYSM



(n) Intel NUC10i7FN



(o) Z690 GAMING X



(p) X570 GAMING X

Figure 6: The basic block coverage over time for RSFuzzer and Syzgen. The lines are the mean values of the basic block coverage. The x-axis is the time in seconds and the y-axis is the number of basic blocks.

We also applied SPENDER [31] to the evaluated targets as a baseline. The results show that SPENDER was able to detect 13 unique vulnerabilities, while RSFuzzer was able to detect 65. Our manual analysis reveals that SPENDER is limited by the taint-style vulnerability model and the code patterns it uses. In contrast, RSFuzzer is able to find non-taint-style vulnerabilities such as cross-handler vulnerabilities. Additionally, we found that all of the bugs found by Syzgen and SPENDER can also be found by RSFuzzer. Overall, our results demonstrate the effectiveness of RSFuzzer in detecting a wider range of vulnerabilities compared to Syzgen and SPENDER.

D. *Effectiveness of Knowledge Extraction Module*

We evaluated the effectiveness of the extracted knowledge for fuzzing. We used two metrics (vulnerabilities and code coverage) to confirm the impact of the extracted knowledge on RSFuzzer. We ran RSFuzzer with and without the extracted knowledge and compared the results. For knowledge-aware fuzzing, we simply ran RSFUZZER with the knowledge extraction module. For knowledge-unaware fuzzing, we removed the knowledge extraction module and only randomly

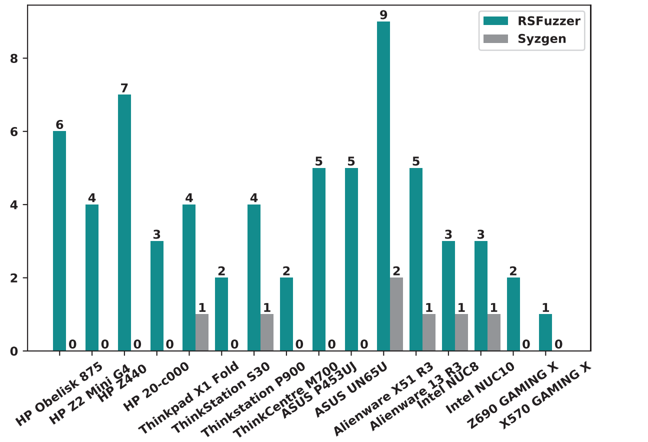


Figure 7: **Vulnerabilities found by RSFuzzer and Syzgen.**

Table III: Statistics for "Knowledge Extraction Module" in terms of the number of basic block and vulnerabilities found.

Device	#Unique Basic Blocks		#Vulnerabilities	
	Knowledge-Aware	Knowledge-Unaware	Knowledge-Aware	Knowledge-Unaware
HP Obelisk 875	1108	181	6	0
HP Z2 Mini G4	4112	934	4	0
HP Z440	2428	436	7	0
HP 20-c000	764	156	3	0
Thinkpad X1 Fold	2489	117	4	2
ThinkStation S30	162	97	2	0
Thinkstation P900	3502	313	4	2
ThinkCentre M700	1150	312	2	0
ASUS VivoMini UN65U	1293	309	5	0
ASUS P453UJ	1515	297	5	0
Alienware X51 R3	1492	529	9	0
Alienware 13 R3	2723	252	5	0
Intel NUC8i3CYSM	1022	252	3	1
Intel NUC10i7FN	2207	218	3	1
Z690 GAMING X	2128	17	2	0
X570 GAMING X	1516	166	1	0
Total	29611	4586	65	6
Improve	545%	-	983%	-

mutated the input data.

Table III shows the results for the number of unique basic blocks and vulnerabilities. RSFuzzer found 65 vulnerabilities with knowledge-aware fuzzing, but only 6 vulnerabilities with knowledge-unaware fuzzing. In other words, the knowledge extraction module helped RSFuzzer find 983% more vulnerabilities than knowledge-unaware fuzzing. In terms of code coverage, knowledge-aware fuzz testing outperformed knowledge-unaware fuzz testing by 545%. These results strongly indicate the effectiveness of the knowledge extraction module.

To further assess the contribution of input format knowledge and cross-handler knowledge to the detection of vulnerabilities, we analyzed the relationship between the detected vulnerabilities and these two types of knowledge. As shown in Table IV, input format inference is necessary for detecting vulnerabilities involving complex input formats. Extracting cross-handler knowledge is also important for scheduling SMI handlers and detecting cross-handler vulnerabilities.

Table IV: The ablation study of RSFuzzer.

Input Format Knowl.	Cross-Handler Knowl.	#Vulnerabilities
Y	Y	7
Y	N	52
N	Y	0
N	N	6

VIII. Discussion

Limitations of RSFuzzer. One of the main limitations of RSFuzzer is the overhead and slow performance of the emulator, despite our use of a partial emulation strategy. This is a common challenge for all emulation-based fuzzing tools for stripped binaries. RSFuzzer achieves a fuzzing throughput of 18-42 exec/sec, which is much lower than the throughput of native execution fuzzing tools such as AFL and Libfuzzer, which can achieve thousands of executions per second on the same hardware setup. To assess the impact of this overhead, we performed an in-depth investigation of the time spent at each stage of fuzzing. We calculated the average execution time over 10,000 rounds of execution for each firmware image. On average, the knowledge extraction phase was the most time-consuming phase, due to the combination of symbolic execution and emulation to extract single/cross-handler knowledge. RSFuzzer is also limited by the availability of UEFI firmware images. We downloaded firmware images from the official websites of vendors, but many of these were in the form of update bundles that contain only the modules that need to be updated for efficiency. These bundles often miss important modules that are not updated, reducing the number of potential input SMI handlers in the analysis. Additionally, the dependency drivers that produce the protocol instances are not complete in the update bundles.

Bugs that require further research. During the manual investigation of the analyzed SMM services, we have observed that there are two categories of vulnerabilities, information leak and improper access control, which cannot terminate the SMM service execution by a fatal signal. These vulnerabilities are merely benefited from our fuzzing test now. For the information leak bug, some SMM services use the SetVariable service, whose fifth argument is a pointer that points to a buffer storing the content of the UEFI variable and fourth argument records the size of the UEFI variable to write an UEFI variable. Commonly, an SMM service initializes a buffer and writes the buffer to NVRAM by calling the SetVariable service. However, if an uninitialized variable is passed as the fourth argument, calling SetVariable will leave a large portion of the buffer uninitialized. These uninitialized bytes will be manifested to NVRAM, where they can be queried by attackers running with kernel-level privileges. The UEFI variables are used to configure hardware by the UEFI firmware at boot up stage. Moreover, some UEFI variables are leveraged to configure hardware security (e.g., Secure Boot). These UEFI variables should not be modified by the kernel-space programs. However, we observe that some UEFI firmware images do not protect these UEFI variables correctly in practice. An attacker with kernel-level privileges can modify these UEFI variables to disable the security features (CVE-2022-26863) or lock down the entire computer system (CVE-2022-26862) during the next power cycle. RSFuzzer cannot detect these vulnerabilities, because they do not lead to any observable and immediate crash of the system before rebooting the system. Further research is needed to detect these silent vulnerabilities.

IX. Related work

Hybrid Fuzzing. Hybrid fuzzing is a technique to combine fuzzing with symbolic execution so that these two techniques can complement each other and achieve better performance [10], [32], [9], [33], [34], [35], [36], [37], [38], [39], [26]. Among all the hybrid fuzzing techniques, the most related work is Excite [39], which is a hybrid fuzzer dedicated to SMI handlers. Although Excite can partially solve the challenges **C2**, **C4** and **C5**, it fails to address challenges **C1** and **C3**. Therefore, Excite cannot detect cross-handler vulnerabilities or vulnerabilities involving inputs from hard-coded memory addresses (e.g., the vulnerability shown in

Figure 5). Compared with Excite, RSFUZZER can detect more vulnerabilities. We did not compare RSFUZZER with Intel Excite directly through experiments because Excite focuses on detecting vulnerability at the source-level and it is a closed-source commercial product with limited access. However, RSFUZZER can find 0-day vulnerabilities in Intel products, which might have been tested by Excite. Another closely related work is Syzgen [26], which is designed to fuzz closed-source macOS drivers. The closed-source macOS drivers are similar to SMI handlers from the fuzzing point of view since the macOS drivers also require well-formatted inputs. In other words, Syzgen focuses on addressing **C2** and **C4** but cannot address **C1**, **C3** and **C5**. In our experiments, RSFUZZER has demonstrated great advantage over Syzgen by covering 617% more basic blocks and detecting 828% more vulnerabilities on average even if we gave Syzgen the ability to address **C5**. For other hybrid fuzzing techniques, most of them only address **C4** but not **C2** as they only focus on penetrating branch constraints and do not care about recovering input formats. In summary, none of the existing hybrid fuzzing techniques can address all the five challenges we identified, which limits their vulnerability detection capabilities on SMI handlers.

SMM Security Analysis. Several techniques have been proposed to implant traditional automatic binary analysis approaches onto the UEFI firmware images analysis. SPENDER [31] is a recent work which is able to scan the potential SMM privilege escalation vulnerabilities in the striped UEFI binaries. As a static detection approach, SPENDER can find single-handler vulnerabilities in the firmware images. But it is not capable of detecting cross-handler vulnerabilities, which requires dynamic information such as the actual values of the cross-handler variables during execution. Chipsec [23] is a blackbox fuzzer that directly tests firmware running on physical machines. [24] is a greybox fuzzer that combines AFL and SIMICs [40] to fuzz the firmware. It uses code coverage as feedback. In contrast, RSFuzzer is a hybrid-fuzzer that involves gray-box fuzzing and concolic testing. [23] and [24] cannot find cross-handler vulnerabilities and the vulnerabilities that require complex input formats. In contrast, RSFuzzer can detect cross-handler vulnerabilities and the vulnerabilities involved in complex input formats by learning input format and function relations incrementally by switching between single and multi-handler fuzzing modes.

Fuzz Driver Generation. Building function-call sequences in fuzz driver generation techniques [41], [42], [43] is similar to how RSFUZZER detects cross-handler vulnerabilities. However, the difference is that RSFUZZER does not learn function relations from existing consumer programs. Instead, it learns function relations from specific mechanisms of SMI handlers. Compared with these techniques, the novelty of RSFUZZER is that it can learn function relations incrementally by switching between single and cross-handler fuzzing modes. Moreover, RSFUZZER is the first hybrid-fuzzer for SMI handlers and we are the first to point out the existence of cross-handler vulnerabilities. KSG [44] is a fuzzer for Linux kernels. It hooks key resource(e.g.,file,socket)-handling functions to identify the producer-consumer relations between system calls. First, KSG relies on domain knowledge about resource-handling functions, while RSFUZZER requires no domain knowledge. Therefore, RSFUZZER switches between single and multi-handler fuzzing modes to incrementally learn function relations. Second, unlike OS-level resources, the variables in SMI handlers require liveness checks.

Firmware Re-hosting. HALucinator [45] provides generic implementations of Hardware Abstraction Layers (HALs) functions to decouple the hardware from the firmware. P2IM [46] re-hosts firmware to facilitate fuzz testing. LuaQEMU [47] is a QEMU-based framework exposing several of QEMU-internal APIs to a LuaJIT core injected into QEMU itself. Unicorn [48] is a pure CPU emulator without any awareness of operating systems built on it. Qiling [29] is built on top of Unicorn, which is an advanced binary emulation framework written in Python. Qiling contains OS-level utilities (such as executable format loaders) and supports UEFI emulation [49]. Therefore, RSFuzzer delegates the emulation to Qiling and focuses more on the fuzzing part. UEFI APIs can be categorized as either boot dependencies or runtime dependencies for SMI-handlers. However, Qiling only provided the abstractions of some key UEFI APIs which serve as boot dependencies and a lot of other UEFI APIs were not included. Based on our experience, some SMI-handlers cannot be emulated due to the lack of abstractions of some other boot-dependency APIs. In RSFuzzer, we provided abstractions for more boot-dependency UEFI APIs. As for the runtime-dependency APIs, we identify and reuse the original APIs instead of using abstractions.

X. CONCLUSION

In this paper, we have identified five key challenges for fuzzing SMI handlers. By addressing these challenges, we propose RSFUZZER, a novel hybrid gray-box fuzzing framework to detect SMI handler vulnerabilities. RSFUZZER is the first fuzzing technique that can detect cross-handler vulnerabilities. Our evaluation shows that RSFUZZER can effectively discover vulnerabilities in COTS UEFI firmware images. In total, RSFUZZER found 65 previously unknown SMI handler vulnerabilities and we have reported them to the corresponding vendors. Until now, 33 of them are confirmed by the vendors and 20 are fixed. In addition, 14 CVE IDs have been assigned. Lastly, the evaluation also shows that RSFUZZER can significantly outperform state-of-the-art input inference hybrid fuzzer in both code coverage and vulnerability detection.

XI. ACKNOWLEDGMENTS

We thank reviewers for their constructive comments. This work is supported in part by National Key R&D Program of China (2022YFB3103900), Chinese National Natural Science Foundation (U1836209, 61802394, 62032010, 62102411, 62202462), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), and ARC Grants DP180104069 and DP210102409.

References

[1] statista, "Total unit shipments of personal computers (PCs) worldwide from 2006 to 2020," 2021, https://www.statista.com/statistics/273495/global-shipments-of-personal-computers-since-2006/.

[2] Bruno, "Through the smm-class and a vulnerability found there." https://www.synacktiv.com/posts/exploit/through-the-smm-class-and-a-vulnerability-found-there.html, 2020.

[3] f secure, "A Security Issue in Intel's Active Management Technology (AMT)," 2018, https://blog.f-secure.com/intel-amt-security-issue/.

[4] O. Bazhaniuk, "A new class of vulnerabilities in smi handlers," http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf, 2015.

[5] Dmytro, "Exploiting smm callout vulnerabilities in lenovo firmware," http://blog.cr4.sh/2016/02/exploiting-smm-callout-vulnerabilities.html, 2016.

[6] M. Zalewski, "american fuzzy lop (2.52b)," 2019, https://lcamtuf.coredump.cx/afl/.

[7] LibFuzzer, "LibFuzzer: A library for coverage-guided fuzz testing," 2019, https://llvm.org/docs/LibFuzzer.html.

[8] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE : Fuzzing windows applications with harness synthesis and fast cloning," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/winnie-fuzzing-windows-applications-with-harness-synthesis-and-fast-cloning/

[9] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-5_Zhao_paper.pdf

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[11] A. Humphries, K. Cating-Subramanian, and M. K. Reiter, "TASE: reducing latency of symbolic execution with transactional memory," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/tase-reducing-latency-of-symbolic-execution-with-transactional-memory/

[12] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[13] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing." in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 754–768. [Online]. Available: http://dblp.uni-trier.de/db/conf/sp/sp2019.html#JeongKSLS19

[14] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," *CoRR*, vol. abs/2105.05445, 2021. [Online]. Available: https://arxiv.org/abs/2105.05445

[15] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, pp. 1099–1114.

[16] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: a versatile, configurable, extensible fuzzing framework," *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[17] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, 2018.

[19] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2325–2342. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[20] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2095–2108. [Online]. Available: https://doi.org/10.1145/3243734.3243849

[21] ——, "Hawkeye: Towards a desired directed grey-box fuzzer," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[22] Z. Du, Y. Li, Y. Liu, B. Mao, L. Chen, J. Guo, Z. He, D. Mu, C. Pang, R. Yu *et al.*, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[23] "Chipsec: Platform security assessment framework," https://github.com/chipsec/chipsec, 2022.

[24] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer, "Uefi firmware fuzzing with simics virtual platform," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[25] I. Safonov and A. Matrosov, "Excite project: all the truth about symbolic execution for bios security," https://github.com/REhints/Publications/blob/master/Conferences/ZeroNights_2016/Excite_Project_ZN.pdf, 2016.

[26] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 749–763.

[27] uefi.org, https://uefi.org/sites/de-fault/files/resources/UEFI_Spec_2_8_final.pdf, 2019.

[28] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *NDSS*, Feb. 2018. [Online]. Available: http://s3.eurecom.fr/docs/ndss18_muench.pdf

[29] "Qiling framework," https://qiling.io/, 2021.

[30] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, 2015, pp. 31–54.

[31] J. Yin, M. Li, W. Wu, D. Sun, J. Zhou, W. Huo, and J. Xue, "Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 1570–1570. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00141

[32] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[33] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1580–1596. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00002

[34] Y. Chen, M. Ahmadi, R. M. farkhani, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 77–92. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/chen

[35] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 515–530. [Online]. Available: https://doi.org/10.1145/3319535.3354249

[36] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1613–1627.

[37] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "Cab-fuzz: Practical concolic testing techniques for COTS operating systems," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 689–701. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/kim

[38] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: hybrid fuzzing on the linux kernel," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[39] J. Engblom, "Finding bios vulnerabilities with symbolic execution and virtual platforms," https://www.intel.com/content/www/us/en/developer/articles/technical/finding-bios-vulnerabilities-with-symbolic-execution-and-virtual-platforms.html, 2017.

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

[41] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456

[42] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.

[43] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "{APICraft}: Fuzz driver generation for closed-source {SDK} libraries," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2811–2828.

[44] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "{KSG}: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[45] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting through Abstraction Layer Emulation," August 2020.

[46] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," *29th USENIX Security Symposium*. [Online]. Available: https://par.nsf.gov/biblio/10213165

[47] "Emulation and exploration of bcm wifi frame parsing using luaqemu," https://comsecuris.com/blog/posts/luaqemu_bcm_wifi/, 2017.

[48] "Unicorn: The ultimate cpu emulator," https://www.unicorn-engine.org/, 2015.

[49] A. Carlsbad, "Moving from manual reverse engineering of uefi modules to dynamic emulation of uefi firmware," 2020. [Online]. Available: https://labs.sentinelone.com/moving-from-manual-re-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/