

FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2

Connor Glosner, Aravind Machiry
Purdue University
{cglosne, amachiry}@purdue.edu

Abstract—Unified Extensible Firmware Interface (UEFI) specification describes a platform-independent pre-boot interface for an Operating System (OS). EDK-2 Vulnerabilities in UEFI interface functions have severe consequences and can lead to Bootkits and other persistent malware resilient to OS reinstallations. However, there exist no vulnerability detection techniques for UEFI interfaces. We present FUZZUER, a feedback-guided fuzzing technique for UEFI interfaces on EDK-2, an exemplary and prevalently used UEFI implementation. We designed FIRNESS that utilizes static analysis techniques to automatically generate fuzzing harnesses for interface functions. We evaluated FUZZUER on the latest version of EDK-2. Our comprehensive evaluation on 150 interface functions demonstrates that FUZZUER with FIRNESS is an effective testing technique of EDK-2’s UEFI interface functions, greatly outperforming HBFA, an existing testing tool with manually written harnesses. We found 20 new security vulnerabilities, and most of these are already acknowledged by the developers.

I. INTRODUCTION

Unified Extensible Firmware Interface (UEFI) [3] is a specification that describes a standard interface between the Operating System (OS) and the platform firmware, along with the interface between the drivers of the platform firmware itself. Specifically, the UEFI defines a set of interfaces and structures available to the OS at boot time and runtime (*i.e.*, when the OS is completely booted and running). UEFI enables platform independent OS booting, which allows the OS to remain independent of the specifics and capabilities of the underlying platform, provided the platform is using the UEFI and not the legacy BIOS. Intel’s EDK-2 [46] is the most widely known open-source implementation of the UEFI. Most platform vendors use EDK-2 (*e.g.*, ARM, AMD, Intel, NVIDIA, Lenovo, Apple, Dell, Microsoft, etc.), which is seen through either their public EDK-2 forks [6], [31], [32] or through public disclosures [5], [36], [44]. Vulnerabilities in EDK-2 have severe consequences and can lead to Bootkits [30] and other persistent malware resilient to OS reinstallations.

Recent works [17], [29], [45] have shown the prevalence of severe security vulnerabilities in EDK-2 components. For instance, recently, Binarly found 24 severe security vulnerabilities in image parsing components of EDK-2, dubbed LogoFail [45]. The lack of exploitation mitigation

mechanisms (*e.g.*, Address Space Layout Randomization (ASLR) [15]) makes it easy to exploit these vulnerabilities. We need effective techniques to detect vulnerabilities in EDK-2. However, the complexity and the hardware-dependent nature of EDK-2 makes it challenging. Specifically, the modular and event-driven nature of EDK-2 makes it challenging to apply traditional flow-based static vulnerability detection techniques [27], [33]. Dynamic analysis, such as Random Testing or Fuzzing [28], is an optimal choice.

Most of the relevant fuzzing work [35], [47], [49] focuses on System Management Interrupt (SMI) handlers, which are special purpose functions that run in System Management Mode (SMM) mode [13] and are installed by EDK-2. SMI handlers are used by the OS to interact with hardware components through a layer of abstraction. Yang *et al.*, [47] proposes a technique to fuzz EDK-2 on a virtual platform. However, their technique requires the need to use their fuzzing framework. Furthermore, their tool is not open-source, which makes it hard to assess its effectiveness. Recent works [20], [45] target specific components of EDK-2, such as image parsing, and focus on fuzzing them. However, such targeted fuzzing isn’t feasible on a large scale and requires a significant engineering effort. We aim to develop an automated coverage-guided fuzzing technique for the EDK-2 implementation of UEFI components, specifically interface functions and drivers. However, the design, semantics, and bare-metal execution of the EDK-2 possess several technical challenges (§ IV) for effective fuzzing. Furthermore, the lack of OS abstractions and runtime support pose engineering challenges in using sanitizers, such as Address Sanitizer (ASAN) [37], which are essential for effective vulnerability finding.

In this paper, we present FUZZUER a coverage-guided fuzzing framework for UEFI interface functions in EDK-2. We use a combination of reaching definition and value-set analysis to identify the data type of parameters accepted by these interface functions. We also identify generator functions that enable us to generate well-formed and stateful objects using fuzzer-generated data. Given an interface function, our analyses will emit a source-level harness (using the types and corresponding generator functions) to exercise the function. Specifically, the harness will be compiled into an UEFI shell application that exercises the target function with well-formed arguments created from input (*i.e.*, fuzzer-generated data). We make engineering enhancements enabling ASAN instrumentation and coverage feedback. Our evaluation of FUZZUER shows that it can detect 66% of previously known vulnerabilities and achieves an average of 50% coverage across various

interface functions. FUZZUER also found 20 new vulnerabilities across 150 EDK-2 interface functions. Our ablation study shows that our analysis techniques, in comparison with the naive approach, improve code coverage by 30% and detect twice the number of bugs. In summary, the following are our contributions:

- We present FIRNESS, which combines static analysis techniques with templates to generate source-level harnesses to test EDK-2s UEFI interface functions.
- We present FUZZUER, a coverage-guided fuzzing system for EDK-2 interfaces using FIRNESS.
- We performed a comprehensive evaluation across 150 interface functions and found 20 new security vulnerabilities. We also show that information generated by FIRNESS greatly improves the coverage and bug-finding ability of FUZZUER, outperforming HBFA [34], current state-of-the-practice testing tool with manually written harnesses.
- We make our framework open-source at <https://github.com/BreakingBoot/FuzzUEr.git> to enable further research in testing UEFI interfaces.

II. BACKGROUND AND MOTIVATION

First, we present the necessary background (§ II-A) on the UEFI components specifications, their usage, and interactions. Next, we present the motivation for our work through vulnerabilities study (§ II-B).

A. UEFI

The Unified Extensible Firmware Interface (UEFI) is a complicated specification defining how components of modern computer bootloaders interact with each other. The goal of UEFI is to allow different OEMs to independently develop drivers for their hardware that will work across many different hardware configurations rather than developing a unique driver for each system.

EDK-2 [46]: is Intel’s exemplary open-source implementation of the UEFI framework. As mentioned in § I, most (rather all) platform vendors’ UEFI frameworks are based on EDK-2. In the rest of the paper, we use UEFI and EDK-2 synonymously; all examples are from the official EDK-2 repository.

1) *Phases:* There are four main phases in UEFI, as shown in Figure 1: Security (SEC), Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), and Boot Device Selection (BDS). The SEC phase is responsible verifying the digital signatures of the firmware to ensure a Secure Boot Environment, along with initializing basic temporary hardware components like a memory controller and a timer. The PEI phase builds upon SEC by initializing more hardware and loading PEI modules that are responsible for further hardware setup. The DXE phase is the main phase of the boot process. It (*i.e.*, *DXE Dispatcher*) is responsible for loading the rest of the drivers needed during the boot process, including permanent drivers that will remain on the system even during runtime. The BDS phase is the selection phase that will choose which operating system to boot into. The DXE phase (our focus) exposes various interfaces for BDS and OS bootloader to interact with the UEFI environment.

2) *DXE Interfaces:* At a high level, interfaces exposed at the DXE layer can be classified into services and protocols.

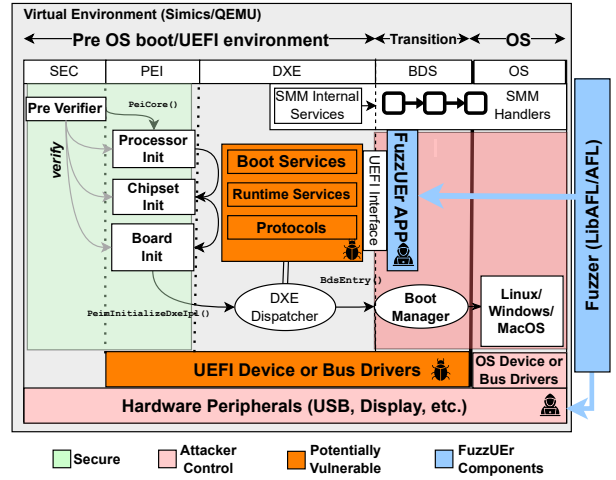


Fig. 1: UEFI Background and FUZZUER components

Listing 1 Example demonstrating Accessing PxeBoot Protocol.

```

1  EFI_PXE_BASE_CODE_PROTOCOL *PxeBoot;
2  Status = gBS->LocateProtocol (
3      &gEfiPxeBaseCodeProtocolGuid,
4      NULL,
5      (VOID **) &PxeBoot
6  );
7  EFI_MTFTP6_PROTOCOL *Mtftp6Prot;
8  EFI_PXE_BASE_CODE_PACKET Packet;
9  // Generate Packet Data
10 Mtftp6Prot->GetInfo(..., (VOID **) &Packet);
11 // Set the packet
12 PxeBoot->SetPackets(..., &Packet);

```

a) *Services:* These are interfaces defined by the UEFI standard and should be implemented in any UEFI-compatible pre-boot environment. The services (*i.e.*, a set of functions) expose the necessary functionality to interact with the UEFI environment, *e.g.*, *AllocatePages* service can be used to allocate pages of a particular type (similar to *malloc*). The Appendix A-A presents the details of DXE services.

b) *Protocols:* These interfaces allow vendors to easily customize existing UEFI components (*e.g.*, services and drivers) and expose new functionality. Each protocol has a Globally Unique Identifier (GUID) and associated interface (a C struct with a set of function pointers). The Appendix A-C shows an example of customizing device drivers using protocols.

Registering Protocol: A protocol can be registered by using `InstallProtocolInterface` or `InstallMultipleProtocolInterfaces` (recommended) boot services. You need to provide a GUID and the pointer to the interface structure.

Listing 7 (in Appendix) shows an example of registering a protocol with `gEfiPxeBaseCodeProtocolGuid` (indicated by \rightarrow_1) containing the GUID and `gPxeBcProtocolTemplate` as the interface, indicated by \rightarrow_2 , is a structure containing a sequence of function pointers. For instance, as indicated by \rightarrow_3 , the pointer for the function `EfiPxeBcUdpRead` is used as a member of the interface.

Accessing Protocol: Listing 1 shows an example of accessing a protocol. First, using the GUID



(gEfiPxeBaseCodeProtocolGuid) of the target protocol, we need to get the pointer to its interface through the LocateProtocol boot service (e.g., Line 2 of Listing 1). If successful, the LocateProtocol service will update the 3rd argument (e.g., PxeBoot at Line 5 in Listing 1) with the address of the interface. The protocol functions can be invoked through the function pointers within the interface, e.g., PxeBoot->SetPackets on Line 14 in Listing 1.

B. Vulnerabilities Study

As mentioned before (in § II-A), DXE interfaces are the primary attack surface of UEFI. Our study determined that the majority (71%) of vulnerabilities are in protocols. Furthermore, 49% of them are memory corruption vulnerabilities. Since protocols constitute most DXE interfaces and also have a high rate of vulnerabilities, our work primarily focuses on fuzzing protocol interfaces. More details on the vulnerability study can be found in Appendix A-B.

III. THREAT MODEL

The Figure 1 highlights our threat model. We assume that phases prior to DXE (i.e., SEC and PEI) are secure, as there is no direct way to interact with these phases during the boot process.

As indicated by  in Figure 1, we aim to find bugs in the code running in the UEFI or pre-OS boot environment, specifically ones that can be triggered through DXE interfaces or device drivers. We use a UEFI application to interact with the UEFI environment through DXE services. Furthermore, we adopt the malicious external peripherals (e.g., USB or Display device) model and interact with device drivers by providing arbitrary data through peripherals. We represent the input injection points by  in Figure 1.

Note that, even though the SMM has been initialized at this point, we do not focus on SMI handlers, which is the focus of several recent works on UEFI security [47]–[49].

IV. CHALLENGES

Given a protocol (i.e., GUID) and (optionally) the target interface function (f) (e.g., SetPackets or GetInfo in Listing 1), our goal is to perform coverage-guided fuzz testing of the function. However, the following challenges exist:

- Input Generation:** We need to use the input data (i.e., fuzzer provided data) to generate arguments of the appropriate type to invoke f . This requires *identifying the data type of parameters* and a *method to generate values of corresponding types*. However, the use of generic types (i.e., `void *`) in function signatures makes it challenging to identify the precise type of corresponding parameters. As shown in Table I, there is a prevalent use of `void *` types in UEFI protocols. e.g., the GetInfo interface function in Listing 1. Even when the type is known, the requirement of *state-dependent objects* makes it challenging to generate corresponding objects from arbitrary data. For instance, in Listing 1, the Packet argument (of EFI_PXE_BASE_CODE_PACKET type) for SetPackets function (at line 14) need to be created by calling GetInfo function, which is another interface function of another protocol.

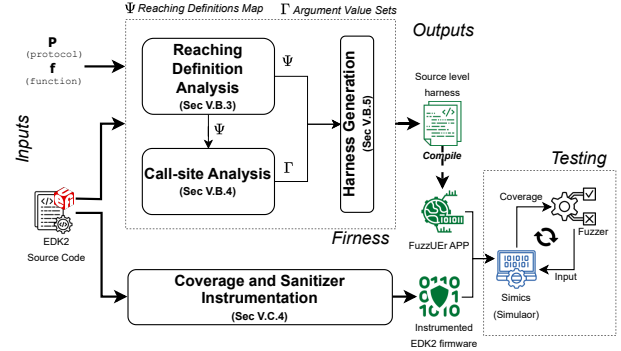


Fig. 2: FUZZUER Overview.

Furthermore, the GetInfo function uses `void**` type for the Packet parameter, which makes existing type-based dependency techniques [22] inapplicable. This is similar to the `FILE*` parameter to fread function in libc. Where we need to call the fopen function to generate a valid object of `FILE*` type. Passing arbitrary objects of `FILE*` type results in error return of fread. We will further demonstrate this problem in § VI.

- Feedback and Bug Identification (Engineering Challenges):** The hardware-dependent nature of UEFI poses engineering challenges in obtaining coverage information and using it as feedback to existing fuzzing techniques. Moreover, the commonly used bug detection enhancements, especially sanitizers (e.g., ASAN [37]), depend on OS abstractions, e.g., virtual memory for shadow memory management by ASAN. Such abstractions are absent in UEFI environment, which has custom memory management and layout. Consequently, these techniques must be customized and re-engineered for the UEFI environment.

V. FUZZUER

We first present a brief overview (§ V-A) and then the details of the individual components (§ V-B). Finally, we will present the necessary implementation details (§ V-C).

A. Overview

Figure 2 shows the overview of FUZZUER, which has three main components: FIRNESS (top-left), instrumentation (bottom-left), and testing (bottom-right).

Given a protocol of interest (P), (optionally) a function of interest (f), and EDK-2 source code, FIRNESS generates a source level harness (i.e., C code) that will invoke f (or all functions in P) with well-formed arguments of the appropriate type. FIRNESS handles the challenge of Input Generation (§ IV) using a set of static analysis techniques (§ V-B) and generates C code that will create well-formed input arguments from the input. The source level harness will be compiled into a UEFI application (FUZZUER APP in Figure 1) for the target firmware and will run in the UEFI shell. The FUZZUER APP will invoke target function (s) by generating well-formed arguments from the input.

We designed a EDK-2 friendly address sanitizer instrumentation (§ V-C4) that will handle the memory layout and allocation semantics of EDK-2. The instrumentation component

adds the necessary compile-time instrumentation to enable our sanitizer and produces the instrumented EDK-2 firmware.

Finally, our testing component will run the FUZZUER APP atop the instrumented EDK-2 firmware in a simulated environment, which consumes input and provides coverage information. We will use existing fuzzers to provide the input and map the coverage information as feedback.

B. FIRNESS

We first present some necessary terminology and then present each component of our technique.

1) Terminology:

- **Functions (F), Function of interest (f_I), and Call Sites ($CSites$):** We use F to represent the set of all functions in the given program and f as an abbreviation for a function. f_I is the function (driver or service function) of interest for which we need to generate the harness. We use $CSites$ to denote all call-sites (*i.e.*, call instructions) in the program.
- **Input Parameters ($In(f)$):** A parameter used by a function f as its input is considered an input parameter. By design, all parameters of scalar types (*e.g.*, `int32_t`) are input parameters. However, pointer type parameters are input only if they are *read* by the function. $In(f)$ is the set of input parameters to f and is represented as a set of non-negative numbers (N) indicating the position of the corresponding parameters. For instance, $In(strncpy) = \{1, 2\}$, as the second (source pointer) and third (length) parameters are used as inputs, *i.e.*, only read from.
- **Output Parameters ($Out(f)$):** Similarly, a pointer parameter used by a function f to write its output is considered an output argument. Specifically, the pointer parameter will only be *written* by the function. Note that an *output parameter should be a pointer (or reference)* – as only updates to parameters passed as references are visible to the caller. Similar to $In(f)$, we use $Out(f)$ to indicate the set of output parameter indices to f . For instance, $Out(strncpy) = \{0\}$, as the data will be written to the first parameter, *i.e.*, destination pointer.
- **Generator Functions (GF , $GF(\tau)$):** These are functions that produce a well-formed (*i.e.*, semantically valid) value of a type τ . We use GF to represent the set of all generator functions, and $GF(\tau)$ is the set of generator functions for type τ . For instance, `fopen` returns `FILE *` and is a generator function for `FILE *` type. Similarly, `png_read_info(p, i)` reads an info struct from `p` (a pointer to png file) into `i` (an `png_info *` type), hence is a generator function for `png_info *` type. We consider a function with a return type or having non-zero output parameters as a generator function of the return type or the type of the corresponding parameter, respectively.
- **Constant Values (C):** These are constant values that can be used in a program. These constants can be numerical values (*i.e.*, integers, floats, chars), composite types (*i.e.*, `structs`), strings (*i.e.*, `char *`). *e.g.*, 1, 3.45, "hello", etc. We use C to indicate the set of all constants in a given program.
- **L-Value Expressions ($LExp$, $Lr(E)$):** These are C-program expressions that can be used as *lvalues* [24]. For instance, `p->value`, `v`, `*(ptr+2)` are L-value expressions, whereas, `(a+b)`, `&h` are not L-Value expres-

Listing 2 Example demonstrating function pointer assignments.

```

1  typedef void (*FuncPtr) ();
2  struct protocol{
3      FuncPtr func1;
4      FuncPtr func2;
5  };
6  void call1(){...}
7  void call2(){...}
8  ...
9  protocol funcptrs = {call1, call2}; 1
10 ...
11 protocol fptr2;
12 ...
13 fptr2->func2 = call1; 2

```

sions. We use $LExp(f)$ to denote the set of all L-Value expressions in function f . For a given expression E , $Lr(E)$ returns the most dominant L-value sub-expression in E . For instance, $Lr(*(a+b))$ returns `*(a+b)`, $Lr(b+2)$ returns `b`. Whereas $Lr(&(ptr->c))$ returns `ptr->c`. However, $Lr("he")$ returns \emptyset , indicating there is no L-value sub-expression.

- **Program Points:** Each statement in a program has a *program point* or *location* l , and $L(f)$ represents the set of all program points in f . Similarly, for an expression e , $L(e)$ gives the program location of e .

2) *Preprocessing:* Here, we collect all the necessary information required for our technique. Specifically, input (In) and output (Out) parameter indices for all functions and targets of indirect calls, *i.e.*, function pointer calls.

We use source-level annotations to identify input and output parameters. EDK-2 coding standards require all function parameters to be marked as input (using `IN` annotation) or output (using `OUT` annotation). This is illustrated on lines 11, 12, and 13 in Listing 7 (in Appendix). These are similar to `__user` annotations [23] in Linux kernel to mark pointers containing addresses from user space. We analyze the function declarations of all functions and consider the parameters annotated as `IN` as input and `OUT` as output parameters, respectively.

To identify function pointer targets, we capture all explicit assignments to the corresponding function pointer. This is similar to steensgaard analysis [40], a flow-insensitive points-to analysis technique. Except that, we do not account for multiple indirect pointers, *i.e.*, `fptr **p`. We also look for constant initializations, such as structure initialization. In Listing 2, the targets for `func2` field of `protocol` structure are $\{call2, call1\}$, because of the initialization (1) and explicit assignment (2). Note that our technique is object-insensitive [38], *i.e.*, all objects of the same type are considered the same.

3) *Reaching Definition Analysis:* For each function f , our goal is to collect possible definitions of its L-Value expressions. Specifically, for a given function f , we collect all of the possible definitions of its L-value expressions (*i.e.*, $LExp(f)$) that can reach various program points in f (*i.e.*, $l \in L(f)$). As mentioned in § V-B, these collected values will be used later in call-site analysis to identify the possible ways to generate inputs to the target function or generators.

Definitions: For pointer L-value expressions (*e.g.*, `ptr`), we consider both direct definitions (*i.e.*, `ptr=NULL`) and through

Listing 3 Example demonstrating valid definitions.

```

1 ptr->fld = NULL; ✓
2 if (...) ptr->fld = p; ✗
3 // Out(read_fld) = {1}
4 if (...) { read_fld(x, ptr->fld); ✓ }
5 // targets(tbl->f) = {func}
6 tbl->f(1, ptr->fld);

```

$$E_1 = E_2, (Lr(E_1) \equiv lr) \wedge ((E_2 \in C) \vee (E_2 \in GF))$$

$$E_2 : f(a_0, \dots, a_n), (f \in GF) \wedge (Lr(a_i) \equiv lr) \wedge (i \in Out(f))$$

Fig. 3: Conditions that need to be satisfied by an expression (E_2) to be considered as a definition for lr .

indirection (i.e., $*ptr = 1$) as definitions of the expression. For non-pointer expressions, we only consider direct definitions. Furthermore, we only consider definitions that are constants or generator functions. In other words, the definitions set can only contain constants or function calls. This design decision enables us to easily generate fuzzing harnesses, which we will explain in § V-B5. Formally, for a given L-Value expression lr , we consider an expression E_2 to be its definition if it satisfies one of the conditions as shown in Figure 3. For instance, the example in Listing 3 shows the definitions of `ptr->fld` that are considered (✓) and ignored (✗). Note that, $Out(read_fld) = \{1\}$ indicates that second parameter (index 1) to `read_fld` is an output parameter and hence the value pointed to by `ptr->fld` will be updated (i.e., defined) by `read_fld`.

For each function, we perform the standard reaching definition analysis [4] on its Control Flow Graph (CFG) by considering definitions that are constants or through generator functions. We record the information into the *reaching definition map* (ψ), which stores the set of definitions of an L-Value expression reachable at a program point, i.e., $\psi : (L \times LRExp(f)) \rightarrow (C \cup GF)$. For Listing 3, at line 6, $\psi(6, ptr \rightarrow fld) = \{NULL, read_fld(x, _)\}$.

4) *Call-Site Analysis*: In this phase, we collect *Argument Value Sets* (Γ) of all input parameters at each possible call-site of the function of interest (f_I) and generator functions (GF). Specifically, at each call site corresponding to f_I or a generator function, we record the possible values that can reach each input parameter. We collect the information in a map, Γ , indexed by the location of the call site, target function (i.e., a target of the call site), and an input argument index. Formally, $\Gamma : (L \times (f_I \cup GF) \times N) \rightarrow (C \cup GF)$.

We use the reaching definition map (ψ) generated from the previous phase (§ V-B3) to compute Γ . For each call-site cs of an interesting function, we collect the values that can reach each input parameter using ψ . If an argument value is a constant, we just record the corresponding value. We provide a formal algorithm of our approach in our extended report [2]. For the code in Listing 3, consider that our f_I is `func`, $In(func) = \{0, 1\}$ and `func` as one of the targets of the call-site at line 6. Using ψ from § V-B3, we compute Γ as: $\Gamma(6, func, 0) \leftarrow \{1\}$, $\Gamma(6, func, 1) \leftarrow \{NULL, read_fld(x, _)\}$.

5) *Harness Generation*: In this phase, we generate the fuzzing harness for f_I and necessary generator functions using the argument value sets (Γ) generated in the previous phase (§ V-B4). The fuzzing harness will use the provided input

(e.g., from a fuzzer) to create well-formed arguments of the appropriate type and use them to invoke f_I . For stateful types (e.g., `EFI_PXE_BASE_CODE_PACKET` in Listing 1), the fuzzing harness might need to use generator functions to create a value of the appropriate type. Specifically, creating a harness for f_I could involve creating a harness for certain generator functions.

We will perform our analysis on both f_I and functions in GF . First, for f_I and all functions in GF , we will merge (and remove duplicates) the argument value sets (Γ) from each call site to create consolidated value sets (Γ_v). Specifically, $\Gamma_v : ((f_I \cup GF) \times N) \rightarrow (C \cup GF)$, which is simply a union of all argument values for each function and their input parameters across various call-sites, i.e., $\forall f \in (f_I \cup GF), i \in In(f)$:

$$\Gamma_v(f, i) = \bigcup_{cs \in CSites(f)} \Gamma(cs, f, i)$$

Second, we will determine the data type of each argument based on the consolidated values (§ V-B5a). Finally, based on the data type (§ V-B5b), we will synthesize methods that will create values of the appropriate type from the input. These values will be used to invoke f_I .

a) *Arguments Type Identification*: For all functions in GF and f_I , our goal is to identify the data type of all its input parameters. As mentioned in § IV, the common approach of determining parameter type through function signature is ineffective because of the prevalent use of generic parameter types (i.e., `void *`).

We capture parameter types from various call-sites of each function. Specifically, for each input parameter, we use the argument values passed at various call sites to determine the expected data type. Specifically, for a function $f \in (GF \cup f_I)$ and its input parameter $i \in In(f)$, we find the set of unique data types of all argument values passed at different call sites, i.e., unique data types of values in $\Gamma_v(f, i)$. If the number of unique data types is less than a pre-defined threshold (Max_τ , we found $Max_\tau = 3$ to work well in practice), then we consider the type of the majority of values as the type of the parameter. For instance, if two values are of type `struct A*`, and another is of `struct B*`, then we consider `struct A*` to be the parameter type. However, if the number of unique types is more than Max_τ , then we consider the parameter as truly generic (i.e., which can accept any type), e.g., the `src` and `dst` parameters of `memcpy` function.

b) *Arguments Value Generation*: We generate values for each input parameter of f_I based on the identified type as described below:

- **Scalar Types**: Primarily, we generate a value of appropriate type from the input data (i.e., provided by a fuzzer). For instance, for `int32`, we will read 4 bytes from input into a `int32` variable and use it as the argument. However, if our call-site analysis identifies constant argument values for the parameter, we randomly (at runtime) choose a value from these constants (from Γ_v) or input data.
- **Composite Types (e.g., `struct`)**: If the type has generator functions, we create harnesses for each generator function and randomly (at runtime) use one of the generator functions to create a value.

We also check whether the composite type is *creatable*, *i.e.*, can be created from raw bytes. Specifically if the type is not recursive and does not contain nested types that are more than two levels deep. For creatable types, we create values by randomly choosing one of the generator functions or from input data.

- **Pointer Types:** We create a heap object of the underlying type and recursively populate the contents of the object based on the type.

c) Harness for Generator Functions: We use the same approach as described above to create harnesses for generator functions. Specifically, we identify the data types (§ V-B5a) and generate values for each argument according to the type (§ V-B5b).

Recursive Dependencies: We discard generator functions that result in recursive dependencies. For instance, consider a generator function f that requires an argument of type τ_1 and generates a value of type τ_2 . Similarly, f_2 requires τ_2 and produces τ_1 and f_3 requires τ_3 and produces τ_1 . Creating a harness for f requires producing a value of τ_1 and $GF(\tau_1) = \{f_2, f_3\}$. However, we only consider f_3 as using f_2 might lead to recursive dependencies. Similarly, to generate values for f_2 , we do not use f .

C. Implementation

As mentioned before (§ II-A), we use EDK-2 as our target UEFI implementation because of its prevalent use. Our implementation targets source-level EDK-2 and generates source-level harnesses; this requirement guided a few of our design decisions – we elaborate more on this in Appendix A-E. FIRNESS is implemented as the set of CLANG-15 source level analyses passes, and the harness generation uses a set of predefined templates. We added support for ASAN instrumentation through a set of source-level patches to EDK-2. In total, our implementation includes 3.5K lines of C++ code (CLANG-15 passes), 151 lines of C code (harness helper), 2.7K lines of Python, and 7.2K line modifications (for ASAN support) to EDK-2 source. Our implementation is available as open source at <https://github.com/BreakingBoot/FuzzUEr.git>.

1) *Input:* The input for our system is the location of EDK-2 source code (we expect the code to be compilable using CLANG-15) and a text file containing the target DXE interface of interest. As mentioned in § II-A2, we categorize DXE interfaces into services and protocols. Services can be specified by their type and the service name, *e.g.*, (`RuntimeService`, `GetVariable`) for `GetVariable` runtime service (as illustrated in Listing 8). Protocols can be specified using their GUID and (optionally) the target interface function name, *e.g.*, (`<GUID for PxeBoot>`, `SetPackets`) for `SetPackets` interface function in Listing 1.

2) *FIRNESS:* During pre-processing, in addition to the function’s input and output parameters, we also record macros, type aliases, enums, structures, and header files where each type is declared. This is needed for our harness generation, which generates compilable C code with necessary `includes`.

The reaching definition and call-site analyses are combined into a single CLANG-15 pass, whose output will be used to perform argument-type identification. Finally, all the

necessary information will be captured in a JSON file (*i.e.*, `firness.json`), which we use to generate the source-level harness. The `firness.json` contains analysis results for each call site of the target function. We present more details of our implementation using an example in Appendix A-D.

3) *Harness Generation:* We use the generated `firness.json` to generate C code (specifically a UEFI application) that will serve as the harness to fuzz the target function. We have created a simple helper library (151 lines of C code) that contains the common functions needed for our harness. This is a one-time effort. For instance, our helper library contains the `ReadBytes(dst_ptr, num_bytes)` function that will read the desired number of bytes (`num_bytes`) from input and store them in the given pointer (`dst_ptr`).

Our harness generation component consolidates the information across different call sites for all relevant functions (*i.e.*, the target function and generators). For each argument, we identify the final argument type (*i.e.*, majority type as explained in § V-B5b) and generate C code to create values of the appropriate type (by using generators when applicable). The Appendix A-F provides more details of our harness generation implementation.

4) *Sanitizer Support:* It is well-known that sanitizers [39], such as Address Sanitizer (ASAN), increase the bug detection capability of testing techniques. Our goal is to add ASAN support for EDK-2. As mentioned in § IV, virtual memory support is not available in UEFI’s pre-boot environment, and also EDK-2 has custom memory management, which makes it challenging to use the default ASAN instrumentation. A previous work [42] tries to add ASAN support to EDK-2. However, it is outdated (6 years old) and does not work with the latest version of EDK-2.

We re-implemented ASAN support for EDK-2 by modifying the memory layout of EDK-2 and configured ASAN shadow memory based on the runtime memory layout. Our modifications also enables ASAN support inside virtual platforms such as Simics.

Additionally, we also need to instrument EDK-2 memory management functions (*e.g.*, `CopyMem`, `SetMem`). We couldn’t simply modify the ASAN compiler passes to instrument all of the function calls to `CopyMem` and `SetMem` because the main memory system isn’t available until the end of PEI, so we manually wrote an instrumented version of the base memory library that contains the EDK-2 memory management functions. Our instrumented version uses the existing base memory library just after the initialization of the main memory at the end of the PEI phase.

VI. EVALUATION

We evaluate FUZZUER by answering the following questions:

- Q1** (*Effectiveness of FIRNESS*): How effective is FIRNESS in identifying necessary information to generate harness? What is the accuracy of the recovered entities? What is the accuracy of the generated fuzzing templates?
- Q2** (*Fuzzing Effectiveness*): How effective is FUZZUER in fuzzing UEFI protocols? What is the code coverage?

Bug finding ability? Did FUZZUER find any new vulnerabilities?

Q3 (HBFA Comparison): How does FUZZUER perform in a best-effort comparison to HBFA (an existing testing technique)?

Q4 (Ablation Study): What is the contribution of each of our techniques on the overall effectiveness of FUZZUER?

We first explain the setup and different configurations of FUZZUER and then investigate our questions.

A. Setup

We used Intel’s TSFFS [21] as our testing platform as it uses libAFL [16] (state-of-the-art fuzzing framework) for input generation and supports virtualized execution of EDK-2 through Simics. We also made minor robustness fixes to handle input generation, and instruction tracing needed to capture the coverage information.

1) *Protocol Selection*: We selected the mainline version of EDK-2 [46] (1eeca0) as our target codebase and used BoardX58Ich10 as our target configuration, as it covered many protocols and is supported by Simics. We selected 25 protocols spanning six categories as summarized in Table I. These protocols cover diverse functionalities supported by EDK-2 and are available in our target configuration. We provide more details of these protocols in the extended report [2]. As mentioned in § II-A2b, each protocol has multiple interface functions. In total, we tested on a total of 150 interface functions.

2) *Known Vulnerabilities*: As our bug dataset, our goal was to forward port previously reported vulnerabilities in our target protocols to the latest version of EDK-2. However, it was challenging to port most of them because of major design changes in the latest version. Furthermore, CVE descriptions missed the patch information, and also commit messages had no CVE information. Nonetheless, we identified patches for three CVEs by painstakingly sifting through commits near the CVE reported date and identified the corresponding patches, which we ported to the latest version of EDK-2. Another major limitation with porting the identified CVEs was the patches that were provided were OEM specific and did not have corresponding patches in EDK-2.

3) *FUZZUER Configurations*: As mentioned in § V-B, FIRNESS collect information about types (T), generators (G), function pointers (P), and generate source level harness (F). We created various configurations of FUZZUER to evaluate the contribution of different techniques of FIRNESS. The letters within square braces indicate different techniques in the corresponding configuration.

- **Random Input** (Fz_r) [F]: In this configuration, we do not use any type information or generators in FIRNESS. Our harness just invokes each interface function with random values, ignoring the parameters’ data type. This serves as the baseline indicating direct use of coverage-guided fuzzing on UEFI interfaces. This is, in principle, similar to the existing technique, *i.e.*, SIMFUZZER [47].
- **No generators** (Fz_g) [F,T,P]: Here, we do not use generators in our harness. Specifically, our harness will not use any generator functions to create argument values.

This configuration aids in demonstrating the importance of generator identification in FIRNESS.

- **No type information** (Fz_t) [F,P,G]: Here, we ignore type information while generating argument values. Specifically, we generate argument values using generators (if available) or random data. This configuration aids in demonstrating the importance of type identification (§ V-B5a).
- **No points-to information** (Fz_p) [F,T,G]: As mentioned in § V-B2, we use points-to analysis to determine function targets, which is needed for our call-site analysis (§ V-B4). We ignore points-to information in this configuration, which aids in demonstrating the importance of handling indirect calls. This is, in principle, similar to FUZZGEN [22], which also tries to generate fuzzing harnesses but does not consider function pointers.
- **Complete system** (Fz) [F,T,P,G]: This is the complete system with all the techniques.

4) *Fuzzing Setup*: We tested each protocol (*i.e.*, all its interface functions) individually for 24 hours for each of the configurations following the recommendations by Kless et al. [26]. Specifically, we generated one harness for each protocol and tested it for 24 hours for each configuration (§ VI-A3). The harness includes all the interface functions. We decide which interface function to invoke based on the first byte of the input.

B. Q1: Effectiveness of FIRNESS

There are three main components of FIRNESS: Reaching definitions (*i.e.*, identifying constants and generators (§ V-B3)), Identifying interface function pointer targets (needed for call-site analysis (§ V-B4)), Identifying argument types (*i.e.*, scalars, structure pointers, scalar pointers (§ V-B5a)), and finally harness generation (§ V-B5c).

1) *Metrics*: We evaluate each of these components in terms of accuracy, *i.e.*, number of true entities identified by our analysis. For parameter-based entities (*e.g.*, possible constants passed), we average the accuracy across all function parameters and compute a single accuracy value. For instance, consider an interface function with 2 parameters, and the constant value identification accuracy (CV) is 50% for one parameter and 100% for the other. Then, we compute CV for the function as 75%, *i.e.*, $(100+50)/2$. We do not use precision and recall, as our analysis did not have false positives (*i.e.*, false positive rate is 0%) because our implementation is done at the source level.

a) *Computing Accuracy Metrics*: We manually verify the results of each component for every protocol interface function and compute the accuracy score.

2) *Reaching Definitions*: As mentioned in § V-B3, the goal of our reaching definition analysis is to identify, for each parameter, constants passed as arguments and generator functions that can produce an argument. We evaluate both of these aspects, *i.e.*, accuracy of (i) identifying constant values passed as arguments; and (ii) generators of each argument.

a) *Constant Value Identification Accuracy (CV)*: The Constants chart in Figure 4 shows the Cumulative Distribution Function (CDF) of CV across all interface functions for each protocol category. Specifically, a point (x,y) on a line of a

TABLE I: Protocol Selection

Categories	Protocols	Number of Functions	Functions with at least 1 void* type	Number of Parameters			
				Min	Max	Mean	Median
USB	USB_IO	13	7	1	7	4.2	4
	USB2_HC	13	6	2	12	6.6	4
TEXT	PRINT2S	10	0	3	5	3.8	4
	HII_FONT	4	2	5	12	8	7.5
	UNICODE	6	0	2	4	3	3
	JSON	4	0	2	5	3.5	3.5
	GRAPHICS	3	0	2	10	5.3	4
CONTROLLER	NVME	4	1	2	4	3	3
	DISK_IO	2	2	5	5	5	5
	IDE	6	0	3	4	3.8	4
	SD_MMC	5	1	2	4	2.8	3
	INCOMP	1	1	7	7	7	7
	PCI_ROOT	14	8	1	7	4	4
SMM	S3_SMM	4	3	2	5	3.75	4
	SMM_BASE2	2	0	2	2	2	2
	SMM_COMM	1	1	3	3	3	3
	SMM_CONT	2	0	2	5	3.5	3.5
Driver Helper	FW_VOL	7	0	1	5	2.9	2
	HEALTH	2	2	4	6	5	5
	HII_DECODER	3	2	3	5	4	4
Network	TCP4	10	0	1	6	2.6	2
	IP4	4	4	3	4	3.5	3.5
	IP6	9	0	1	6	3.2	2.5
	SIMPLE_NET	13	4	1	7	3.6	3
	MANAGED_NET	8	0	1	4	2.4	2
Cumulative		150	44	1	12	4	3.5

TABLE II: List of known vulnerabilities used for our evaluation and New vulnerabilities found by FUZZUER. ✓ and ✗ indicate whether the corresponding configuration found it or not, respectively.

ID	Protocol	Function	Bug Type	Status	Found By				
					Fz_r	Fz_g	Fz_t	Fz_p	Fz_z
Previously Known Bugs									
1	IP4	Ip4PreProcessPacket	Buffer Overflow	Previously Known	✗	✗	✗	✗	✗
2	HII_FONT	UefiFileHandleLib	Buffer Overflow	Previously Known	✗	✗	✓	✓	✓
3	HII_FONT	DevPathToTextUsbWWID	Buffer Overflow	Previously Known	✗	✗	✓	✓	✓
New Bugs									
4	DISK_IO	DiskIoCreateSubtaskList	Buffer Overflow	CONFIRMED	✓	✓	✓	✓	✓
5	ALL*	CR()	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
6	PRINT2S	ShellFileHandleReadLine	Buffer Overflow	CONFIRMED	✓	✓	✓	✗	✓
7	PRINT2S	ShellFindFilePathEx	Use After Free	CONFIRMED	✓	✓	✓	✗	✓
8	PRINT2S	InternalsOnCheckList	Arbitrary Pointer Write	REPORTED	✓	✓	✓	✗	✓
9	UNICODE	EngStriColl	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
10	UNICODE	EngStrLwr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
11	UNICODE	EngStrUpr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
12	UNICODE	EngMetaiMatch	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
13	UNICODE	EngFatToStr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
14	UNICODE	EngStrToFat	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
15	USB_IO	UsbIoControlTransfer	Use After Free	CONFIRMED	✗	✓	✗	✓	✓
16	S3_SMM	InternalSmBusExec	Null Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
17	MANAGED_NET	EfiDhcp6InfoRequest	Arbitrary Pointer Read	REPORTED	✗	✓	✓	✓	✓
18	HII_FONT	HiiStringToImage	Arbitrary Memory Write	REPORTED	✗	✓	✓	✗	✓
19	GRAPHICS	FrameBufferBltLibVideoToBltBuffer	Buffer Overflow	REPORTED	✗	✓	✗	✓	✓
20	FW_VOL	FwVolBlockReadBlock	NULL Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
21	FW_VOL	FwVolBlockReadBlock	NULL Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
22	USB2_HC	EhcAsyncInterruptTransfer	NULL Pointer Dereference	REPORTED	✗	✓	✓	✗	✓
23	USB2_HC	EhcAsyncInterruptTransfer	NULL Pointer Dereference	REPORTED	✗	✓	✓	✗	✓

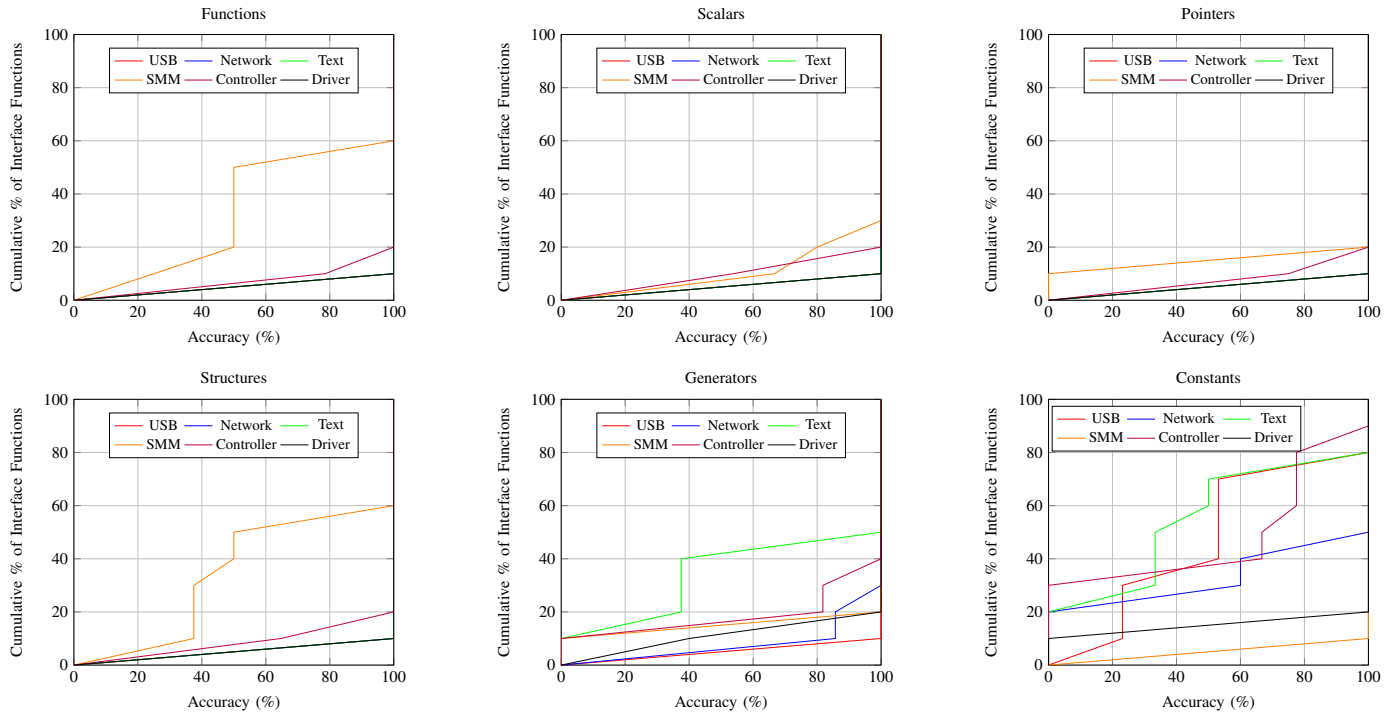


Fig. 4: CDF of Accuracy of different components of FIRNESS.

protocol category indicates that $y\%$ of interface functions have an accuracy of $x\%$ or less. As we can see from almost flat lines, most functions have 100% accuracy. However, there are cases (e.g., Text and Controller) where we fail to identify the constants. The main reason for this is missing value flow information and path constraints. For instance, in the code `if (a==2) foo(a);`, although the argument `a` is not a constant, the value that `a` can hold at the call site is a constant *i.e.*, 2. Our analysis fails to capture such cases as we do not track value flows [43].

b) Generators Identification Accuracy (G): The *Generators* chart of Figure 4 shows the CDF of G across all interface functions for each protocol category. As indicated by almost flat lines, most of the functions have 100% accuracy. The main reason for a failure to identify a generator function is when the type of interest is resolved to a scalar type. An example can be seen in Appendix A-G.

3) Identifying Function Pointer Targets (FPtr): The *Functions* chart of Figure 4 shows the CDF of $FPtr$ across all interface functions for each protocol category. We are able to accurately recover the function pointer targets for all protocol categories except for SMM. The main reason for missing function pointer targets is the lack of support for multi-level function pointers (*i.e.*, `**fptr`). Our current implementation does not track the loads and stores across these multi-level pointers, resulting in missing function pointer targets.

4) Identifying Argument Types: We separate accuracy across scalar types (*Scalars* CDF in Figure 4), `struct` or `struct` pointers (*Structures* CDF in Figure 4), and scalar pointers (*Pointers* CDF in Figure 4). As indicated by the flat lines, the accuracy of scalars and pointers is 100% for most of the interface functions. However, the structure accuracy is lower for interface functions related to SMM protocol category. The main reason for this is the lack of call sites, which results

in no argument values and, consequently, failure to identify types. For example, the `EFI_S3_SAVE_STATE_PROTOCOL` only has call sites for 1 out of 4 of the functions, and the 3 functions that don't have call sites all have at least 1 `void*` parameter. This lack of argument values results in failure to identify parameter types and corresponding generator functions.

5) Harness Generation: The Figure 7 (in Appendix) shows the CDF of the accuracy of the harness generation across all interface functions for each protocol category. For almost all the interface functions, the harness is 80% accurate.

One of the main reasons for the drop in accuracy is recursive generators. As mentioned in § V-B5c, we ignore generators with recursive dependencies and consequently will not include them in the harness. Another reason is the call-backs; few interface functions (specifically of `USB2_HC` protocol) expect call-backs as one of their arguments. However, our current implementation does not support call-back arguments and results in an inaccurate generator (*i.e.*, we pass `NULL` in place of the call-back.)

C. Q2: Fuzzing Effectiveness

We use two metrics to evaluate the fuzzing effectiveness: code coverage and bug-finding ability. As mentioned in § VI-A4, for each protocol, we generate a harness that invokes all interface functions within the protocol.

1) Code Coverage: We measure code coverage as the percentage of reachable code covered throughout fuzzing. For each protocol, we use the call graph of each interface function to identify all reachable functions. We compute the code coverage as the percentage of code covered within these reachable functions. The Figure 5 shows the code coverage over time for each protocol. Specifically, the legend Fz indicates the

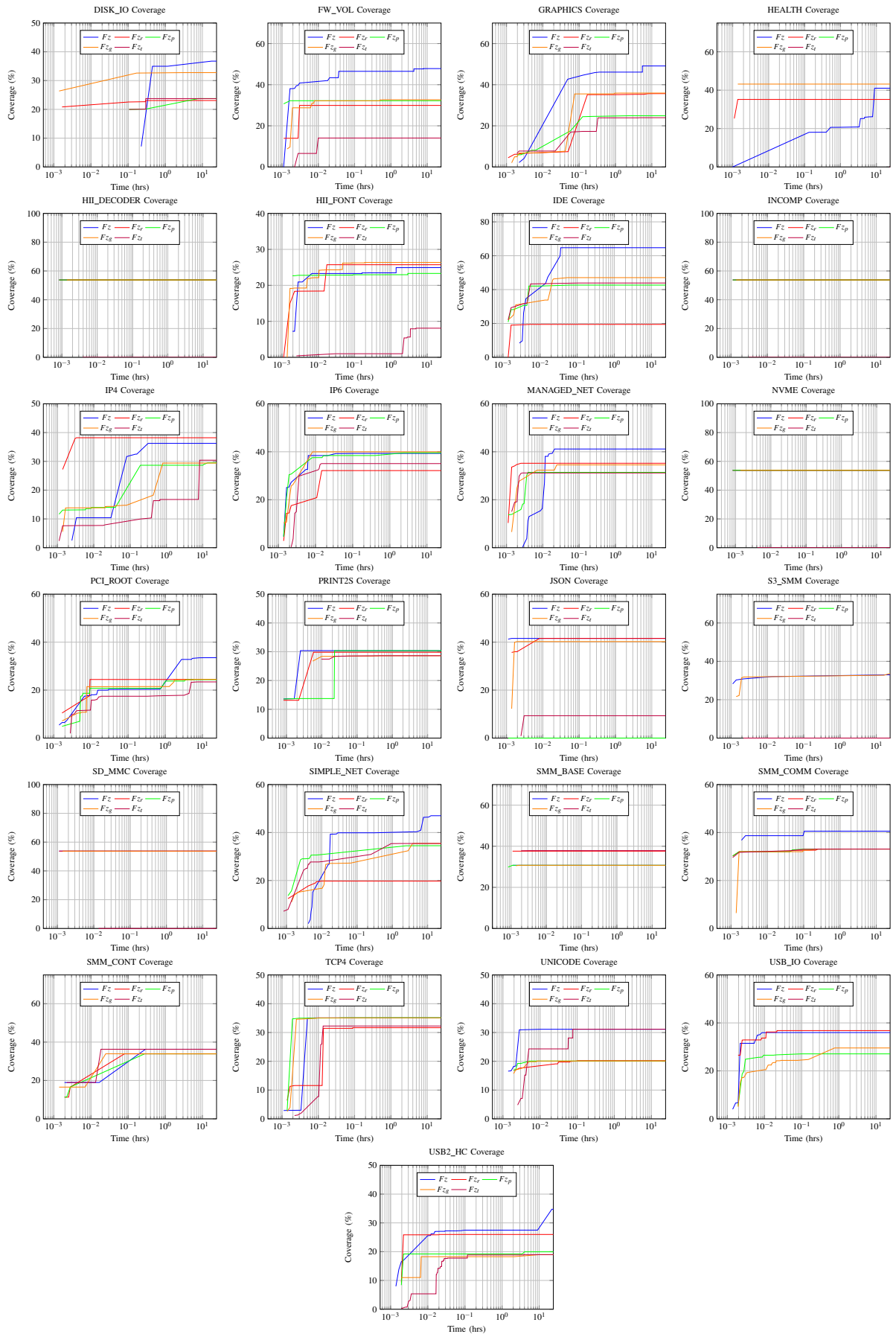


Fig. 5: Coverage Information of various FUZZUER configuration (§ VI-A3) across various protocols.

coverage achieved by FUZZUER with all the information from FIRNESS.

Although we executed each harness for 24 hours, the coverage did not improve after 10 hours. For most of the protocols, FUZZUER quickly (< 1 hour) achieved its peak coverage. This is because our harness contains all the type information and generators, creating well-typed and formatted argument values from fuzzer data. This reduces the possibility of mutation techniques to produce well-typed values as our harness already does the heavy lifting of converting arbitrary data into well-typed values.

On average, FUZZUER achieved ~ 40% coverage across all protocols. We currently are underestimating the coverage because of the extensive use of function pointers and runtime resolution in the UEFI. Specifically, we do not consider function pointers while computing reachable functions. Consequently, the coverage of indirectly (through function pointers) triggered functions will not be captured. We performed a detailed analysis of the lack of coverage and identified complex path constraints as the primary cause. We provide more details with an example in Appendix A-H.

2) *Bug Finding Ability*: The Table II shows the results. As mentioned in § VI-A2, we forward ported three previously known vulnerabilities into the latest version (ours) of EDK-2. The top part of Table II under *Fz* column shows the bugs found by FUZZUER. 66% (i.e., 2 out of 3) of previously known vulnerabilities were found by FUZZUER. Although the absolute number is low, these results still demonstrate that FUZZUER can find previously known bugs.

a) *New Vulnerabilities*: Overall, FUZZUER found 20 new security vulnerabilities across all protocols. The bottom part of Table II shows the details of these vulnerabilities. We reported all these vulnerabilities to EDK-2 developers, and most (i.e., 12 (60%)) of the vulnerabilities have already been confirmed. Although many vulnerabilities are *NULL*-ptr dereferences, there are high-severity vulnerabilities such as buffer overflows and use-after-free. Even the *NULL*-ptr dereferences are severe as these are present in the UEFI's pre-boot environment and could be exploited to make the device unusable (i.e., *bricking* the device). Our results demonstrate that FUZZUER provides an effective technique to fuzz UEFI interfaces.

3) *Case Studies*: We present case studies of two interesting and high-severity vulnerabilities found by FUZZUER.

a) *Incorrect Signature Check*: As shown on line 23 in Listing 4, EDK-2 use `FVB_DEVICE_FROM_THIS` macro to access `EFI_FW_VOL_BLOCK_DEVICE` pointer from the protocol pointer. The macro performs an integrity check (as shown on lines 7-8) to ensure that the pointer is not corrupted. However, the macro fails to perform a *NULL* check, consequently resulting in a *NULL* pointer dereference as indicated by ✖. This bug appears across all protocols with private data, indicated by *ALL** in the corresponding row of Table II.

Unfortunately, fixing this bug is not trivial. We need to gracefully handle cases where the pointer is *NULL*. However, just adding a *NULL* check in the macro is not sufficient because the invalid value should be communicated to the target function where the macro is used (i.e., `FwVolBlockGetAttributes` in

Listing 4 Incorrect signature check leading to *NULL*-ptr dereference (ID 5 in Table II).

```

1  #define BASE_CR(Rec, TYPE, FD) \
2      ((TYPE *) ((CHAR8 *) (Rec) - \
3          OFFSET_OF(TYPE, FD)))
4  #define CR(Rec, TYPE, FD, Sign) \
5      (DebugAssertEnabled () && \
6      (BASE_CR(Rec, \
7      TYPE, FD)->✖Signature != Sign)) ? \
8  #define FVB_DEVICE_FROM_THIS(a) \
9      CR(a, EFI_FW_VOL_BLOCK_DEVICE, \
10         FwVolBlockInstance, FVB_DEVICE_SIGNATURE)
11 EFI_STATUS
12 EFIAPI
13 FwVolBlockGetAttributes (
14     IN CONST
15     EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL *This,
16     OUT EFI_FVB_ATTRIBUTES_2 *Attributes
17 ) {
18     EFI_FW_VOL_BLOCK_DEVICE *FvbDevice;
19     FvbDevice = FVB_DEVICE_FROM_THIS (This▲);
20     *Attributes = FvbDevice->FvbAttributes
21                 & ~EFI_FVB2_WRITE_STATUS;
22 }

```

Listing 5 Arbitrary Memory Write (ID 18 in Table II)

```

EFI_STATUS EFIAPI HiiStringToImage (
...
    IN  OUT EFI_IMAGE_OUTPUT      **Blt,
    IN  UINTN                     BltX ▲,
    IN  UINTN                     BltY ▲,
) {
    Image      = *Blt;
    BufferPtr = Image->Image.Bitmap
              + Image->Width ▲ * BltY ▲ + BltX ▲;
    GlyphToImage (... , &BufferPtr ✖);
}

```

Listing 4). Unfortunately, the macro is used in many places in EDK-2 codebase, and the handling of *NULL* pointer must be explicitly added to every macro use. EDK-2 developers agreed to this and are in the process of making a design change to check for `EFI_FW_VOL_BLOCK_DEVICE` pointer integrity.

b) *Arbitrary Memory Read in Image Parsing*: The Listing 5 shows an arbitrary memory read because the interface function fails to validate the coordinates (i.e., `BltX`, and `BltY`). An attacker can pass very large values and make `BufferPtr` point to any desired address. Later, the function `GlyphToImage` writes to the address pointed by `BufferPtr`, resulting in an arbitrary memory write vulnerability.

D. Q3: HBFA Comparison

Host Based Firmware Analyzer (HBFA) [34] by Intel is an open-source userspace fuzzer for EDK-2. HBFA requires EDK-2 protocol to be rehosted as userspace programs and a harness to exercise interface functions. Consequently, to test a protocol using HBFA, one must implement stubs for all hardware-dependent features and write a harness that can comprehensively exercise the protocol's interface functions — a tedious endeavor. Currently, three protocols are supported by HBFA, i.e., `USB2_HC`, `DISK_IO`, `PCI_ROOT`. The stubs and harnesses for these protocols are developed by EDK-2 engineers and are available in the main repository. We run HBFA with the same setup (§ VI-A4) as FUZZUER.

TABLE III: HBFA(H) vs. FUZZUER(F_z) (§ VI-D)

Protocol Tool	USB2_HC		DISK_IO		PCI_ROOT	
	H	F_z	H	F_z	H	F_z
Harness LoC	63	1,391	597	319	312	1,098
Code Coverage (Number of Unique Edges)						
Total Coverage	319	6,091 (19x)	1,413	8,797 (6x)	762	6,514 (8x)
Driver Coverage	138	2,041 (14x)	595	5,205 (8x)	117	3,690 (31x)
Number of Unique Bugs Found						
Bugs Discovered	0	2 (200%)	0	1 (100%)	0	0

Listing 6 Snippet of HBFA’s DISK_IO Harness

```

VOID
EFIAPI
RunTestHarness(
  IN VOID *TestBuffer,
  IN UINTN TestBufferSize
){
  FixBuffer (TestBuffer);
  DiskStubInitialize (TestBuffer, TestBufferSize,
                     BLOCK_SIZE, IO_ALIGN,
                     &BlockIo, &DiskIo);

  FindUdfFileSystem (
    BlockIo,
    DiskIo,
    &StartingLBA,
    &EndingLBA
  );
  DiskStubDestory ();
}

```

Table III shows the results. As indicated by the LoC column, existing HBFA harnesses are smaller than the ones generated by FIRNESS. Upon manual investigation, we found that all these harnesses are simple and exercise protocols’ functions with mostly fixed data. However, as we explained in § V-B, our techniques enable us to generate well-formed arguments using input data, resulting in diverse arguments.

1) *Coverage*: In Table III, columns *Total Coverage* and *Driver Coverage* show the code coverage (*i.e.*, number of unique edges in CFG) of the overall EDK-2 code and just the protocol code, respectively. On the entire EDK-2 codebase, FUZZUER with our automatically generated harness covered significantly more (11x on average) code than HBFA. One of the main reasons for this is the use of stubs in HBFA, which prevents from exercising low-level EDK-2 code. Even on the protocol code, FUZZUER performed significantly better with 17x (on average) more code than HBFA. This is because the harnesses are simple and only cover known scenarios. For example, Listing 6 shows a snippet of the HBFA’s harness (for DISK_IO protocol), which only covers a simple case.

2) *Bug Finding Ability*: As indicated in the *Bugs Discovered* row of Table III, FUZZUER was able to find three previously unknown bugs while HBFA failed to find any bugs. This is again due to the poor quality and overly simplistic harnesses that fail to adequately test the protocols’ functionality and properly utilize input to generate the necessary arguments. For instance, Listing 6 is the harness used in HBFA. In contrast, Listing 10 (in Appendix) is the harness automatically generated by FIRNESS. Our harness enabled us to find one of the bugs in DISK_IO protocol, *i.e.*, Listing 4 (discussed in § VI-C3), which is missed by HBFA.

E. Q4: Ablation Study

Our goal is to investigate the contribution of different techniques in FIRNESS to the overall effectiveness of FUZZUER. As explained in § VI-A3, we ran FUZZUER under various configurations with increasing amounts of information from FIRNESS.

1) *Coverage*: The Figure 5 shows the coverage information of various configurations for all protocols. Overall, FUZZUER with full FIRNESS information (*i.e.*, F_z) achieves the highest coverage for most of the protocols. On the other hand, not giving any FIRNESS information (*i.e.*, F_{z_r}) results in the least coverage for most of the protocols. The impact of various FIRNESS techniques varies across protocols. For instance, F_{z_g} (*i.e.*, No generators) performs better than F_{z_t} (*i.e.*, No type information) on the FW_VOL protocol. It’s the other way round on USB2_HC. This is because of the diversity in interface functions of different protocols. For instance, most interface functions in the FW_VOL protocol have parameters with no generators. Consequently, not using any generators (F_{z_g}) has less impact and thus higher coverage. Whereas interface functions in the USB2_HC protocol has parameters with generators, and consequently, not using generators has a greater impact and lesser coverage. In a few cases, the coverage of configurations not using complete FIRNESS information is higher than F_z (*i.e.*, using complete FIRNESS information). This higher coverage is because of the error handling code that gets executed because of passing invalid data. Such error handling code will not be covered in F_z as it uses well-formed types. An example of error handling code is explained in further detail in Appendix A-H.

2) *Bug Finding Ability*: The Table II shows the bug-finding ability of different configurations. Using complete FIRNESS information (F_z) finds the highest number of bugs. Whereas the random configuration (F_{z_r}) finds the least number of bugs. The bug-finding ability varies with different amounts of FIRNESS information.

In summary, our results indicate that all components of FIRNESS contribute to the overall effectiveness of FUZZUER.

VII. LIMITATIONS AND FUTURE WORK

In this section, we present the limitations of our work and plans for future work:

- **FIRNESS Limitations**: The current design of FIRNESS cannot handle recursive types and generators, which results in poor harness accuracy for a few interface functions. As part of our future work, we plan to enhance FIRNESS to handle recursive types and also use flow analysis to determine more constant argument values.
- **Low Coverage**: Although FUZZUER is effective at vulnerability finding, it has low code coverage (§ VI-C1). We emphasize that this is not a fundamental limitation of FUZZUER and can be improved by using a concolic fuzzer [41] (instead of the libAFL).

VIII. RELATED WORK

A. Vulnerability Detection on Bootloaders/UEFI

Only a handful of works try to perform vulnerability detection (VD) on UEFI framework. Works that focus on other

TABLE IV: Qualitative comparison FUZZUER and related dynamic analysis and harness generation tools. Symbols \checkmark (yes), \triangle (partial), and \times (no) indicate the availability of the corresponding feature.

Tool	Target Component	Open Source?	Features			
			Supports Arbitrary Types? (T)	Supports Generators? (G)	Supports Function Pointer? (P)	Generates Source level Harness? (F)
Syscall and Library API Tools						
FUZZGEN [22]	Library APIs	\checkmark	\checkmark	\checkmark	\times	\checkmark
HFL [25]	Kernel Syscalls	\checkmark	\triangle	\checkmark	\checkmark	\checkmark
DIFUZE [12]	ioctls	\checkmark	\times	\times	\checkmark	\times
INTELLIGEN [50]	Functions	\times	\triangle	\times	\times	\checkmark
HOPPER [10]	Library APIs	\checkmark	\triangle	\triangle	\times	\times
UEFI Tools						
RSFUZZER [49]	SMI Callouts	\times	\times	\times	\times	\times
EXCITE [14]	SMI Callouts	\times	\times	\times	\times	\times
HBFA [34]	Interface Models	\checkmark	\times	\times	\times	\times
CHIPSEC [11]	SMI Callouts	\checkmark	\times	\times	\times	\times
SIMFUZZER [47]	N/A	\times	\times	\times	\times	\times
FUZZUER (Our System)	UEFI Interfaces	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark

bootloaders, such as Smartphone bootloaders [33], are not applicable to UEFI because of fundamental differences in their software architecture. Existing VD works can be primarily classified into static or dynamic depending on the type of analysis employed.

1) *Static*: Existing static VD tools focus on SMI handlers, which are special high-privilege handlers installed by the UEFI framework and are available to OS at runtime. Yin *et al.*, [48] uses protocol-centric static analysis on UEFI firmware binary to identify privilege escalation vulnerabilities in SMI handlers. There is a series of works by Gu *et al.*, [9], [19] which use network theory and graph algorithms to find logical errors during boot, such as incorrect authentication mechanisms.

2) *Dynamic*: There are several tools [8], [11], [14], [34], [47], [49] that try to perform dynamic analysis (such as Fuzzing) for UEFI. The Table IV shows the summary of all the relevant tools and the components they target, along with supported features. Similar to static techniques, most of the tools [11], [14], [49] focus on SMI handlers. SMI Handlers differ from protocol interfaces because they don't rely on the system to be in a particular state, but instead parse a given input structure to set or change the current system state. CHIPSEC [11] framework provides a suite of tools to perform forensic analysis of UEFI firmware and naive fuzzing of SMI handlers. Bazhaniuk *et al.*, [8] utilizes KLEE to perform symbolic execution on SMM interrupt variables. RSFUZZER [49] combines concolic execution and grey box fuzzing to achieve better test cases and code coverage across SMI handlers. Similarly, Intel's tool EXCITE [14] performs dynamic symbolic execution to achieve better code coverage inside SMI handlers.

One of the first works that tries to perform fuzzing of UEFI interfaces is Intel's HBFA [34]. However, it requires the target interface to be modeled as a standalone application (by stubbing out interactions with the UEFI framework), a tedious manual process, and hard to generalize across different types of protocols with complex parameters. Another work, SIMFUZZER [47], tries to perform automated coverage-guided fuzzing of the entire framework by feeding random data from the simulator. However, SIMFUZZER does not care about the parameter types and generators, greatly affecting its effectiveness. This is similar to the Random Input configuration (§ VI-A3) of FUZZUER. As we show in § VI-E, ignoring types and generators greatly reduces fuzzing effectiveness. Finally, SIMFUZZER is not open-source and is a proprietary

tool. Authors refuse to make the tool for public use or evaluation. In contrast, as shown in Table IV, FUZZUER is an open-source tool that focuses on fuzzing UEFI interfaces by generating well-typed arguments. Table IV summarizes all these techniques and the supported features.

B. Stateful Input Generation

There are several techniques that focus on stateful input generation for the Linux systems [7], [10], [12], [18], [22], [25], [50], [51]. Two of the techniques are based off modeling program points as states to properly mutate those states to achieve greater code coverage [18], [51]. While Ba *et al.*, [7] proposed a generalized analysis to find states in protocol implementations, specifically network related protocols, and use those states to increase the covered state space through fuzzing. FUZZGEN [22] performs whole system analysis to generate an abstract API dependency graph and is used to generate libFuzzer [1] stubs to perform stateful fuzzing across complex systems. FUZZGEN attempts to fuzz library functions, which are conceptually similar to UEFI interfaces. However, unlike FUZZUER, FUZZGEN requires typed parameters and relies on API inference, assuming that the target function declaration matches the library's included definitions. This is not the case in EDK-2, where target functions are called through function pointers. This approach is akin to our Fz_p configuration, which, as demonstrated in § VI-E, does not perform well for protocols in EDK-2. Similarly, INTELLIGEN [50] automatically attempts to locate entry functions and then generate a libFuzzer stub, but lacks support for generator functions and will fail to identify functions that have `void*` as entry functions. HOPPER [10], on the other hand, uses a custom DSL to dynamically create harnesses for a given library, but since it analyses the header files to capture type information it will fail for cases where `void*` is prevalent. Similarly, techniques for Kernel system call testing also lack the required features and have several drawbacks. For instance, HFL [25] performs hybrid fuzzing on Linux Kernel syscalls by resolving indirect calls through Kernel-specific initialization idioms. However, this is unreasonable in most cases for the EDK-2, since protocols can be initialized arbitrarily, *e.g.*, statically or dynamically as illustrated in Listing 2. While DIFUZE [12] is an interface-aware fuzzing tool that analyzes Android source code to generate valid input for kernel drivers, however it lacks the ability to use generators or to infer underlying `void*`.

IX. CONCLUSION

We present FUZZUER a coverage-guided fuzzing framework for UEFI interface functions. Given an interface function, we use a combination of static analysis and templated harness generation to craft a source-level harness (using the types and corresponding generator functions) to exercise the function. Our comprehensive evaluation shows that FUZZUER is an effective technique to test UEFI interfaces as compared to HBFA. FUZZUER discovered 20 new security vulnerabilities in the latest version of EDK-2, an open source implementation of UEFI.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments, which greatly improved our paper. We would also like to thank Prof. Milind Kulkarni for his help in the early stages of this work. This research was supported by the Defense Advance Research Projects Agency (DARPA) under contract number N660012224037, and the National Science Foundation (NSF) under Grant CNS-2340548. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DARPA, the NSF, or the U.S. Government.



REFERENCES

- [1] libfuzzer – a library for coverage-guided fuzz testing. <https://github.com/lvm-mirror/lvm/blob/master/docs/LibFuzzer.rst>, 2016.
- [2] Fuzzuer: Enabling fuzzing of uefi interfaces on edk-2 extended report. <https://drive.google.com/file/d/1o7FhmOmFrPx8h6oi1rfy71Vn1dZtOHM/view?usp=sharing>, 2024.
- [3] Unified extensible firmware interface (uefi) specification, 2024.
- [4] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. pearson Education, 2007.
- [5] Apple. Uefi firmware security in an intel-based mac. 2021.
- [6] ARM. Edk-2. <https://gitlab.arm.com/arm-reference-solutions/edk2-platforms>, 2020.
- [7] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, Boston, MA, August 2022. USENIX Association.
- [8] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. Tuttle, and V. Zimmer. Symbolic execution for bios security. *Workshop on Offensive Technologies*, 2015.
- [9] Fei Cao, Qingbao Li, and Zhifeng Chen. Vulnerability model and evaluation of the uefi platform firmware based on improved attack graphs. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 225–231, 2018.
- [10] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen. Hopper: Interpretative fuzzing for libraries. *arXiv preprint arXiv:2309.03496*, 2023.
- [11] CHIPSEC. Chipsec: Platform security assessment framework. <https://github.com/chipsec/chipsec>, 2015.
- [12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.
- [13] Dori Eldar, Arvind Kumar, and Purushottam Goel. Configuring intel active management technology. *Intel Technology Journal*, 12(4), 2008.
- [14] Jakob Engblom. Finding bios vulnerabilities with symbolic execution and virtual platforms, 2019.
- [15] Dmitry Evtushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.
- [16] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS), CCS '22*. ACM, November 2022.
- [17] Iván Arce Francisco Falcon. Pixiefail: Nine vulnerabilities in tianocore’s edk ii ipv6 network stack. 2024.
- [18] Harrison Green and Thanassis Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1070–1081, 2022.
- [19] Yanyang Gu, Ping Zhang, Zhifeng Chen, and Fei Cao. Uefi trusted computing vulnerability analysis based on state transition graph. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1043–1052, 2020.
- [20] Marvin Häuser. Designing a secure and space-efficient executable file format for the unified extensible firmware interface. 2023.
- [21] Intel. Tsffs. <https://github.com/intel/tsffs/>, 2023.
- [22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.
- [23] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004.
- [24] Brian W Kernighan and Dennis M Ritchie. The c programming language. 2002.
- [25] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung 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.
- [26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.
- [27] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}-{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.
- [28] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.
- [29] Alex Matrosov. The untold story of the blacklotus uefi bootkit, 2023.
- [30] Alex Matrosov, Eugene Rodionov, and Sergey Bratus. *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press, 2019.
- [31] Microsoft. Edk-2. <https://github.com/microsoft/mu>, 2018.
- [32] NVIDIA. Edk-2. <https://github.com/NVIDIA/edk2>, 2022.
- [33] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. {BootStomp}: On the security of bootloaders in mobile devices. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, 2017.
- [34] Brian Richardson, Chris Wu, Jiewen Yao, and Vincent J. Zimmer. Using host-based firmware analysis to improve platform resiliency. 2019.
- [35] Mahesh Sarikonda and Shanmugasundaram R. Validation of firmware security using fuzzing and penetration methodologies. pages 1–5, 2022.
- [36] Dell Security. Dsa-2023-191: Security update for dell client bios tianocore edk2 vulnerabilities. 2024.
- [37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

- [38] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, 2011.
- [39] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.
- [40] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.
- [41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
- [42] Shi Steven. Edk-2. <https://github.com/shijunjing/edk2/tree/sanitizer2>, 2022.
- [43] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.
- [44] Lenovo Support. Tianocore edk ii bios vulnerabilities. 2019.
- [45] Binarly Research Team. Finding logofail: The dangers of image parsing during system boot, 2023.
- [46] Tianocore. Edk-2. <https://github.com/tianocore/edk2>, 2007.
- [47] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer. Uefi firmware fuzzing with simics virtual platform. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.
- [48] J. Yin, X. Wang, D. Wei, J. Chen, Y. Zhou, Z. Li, and Y. Liu. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637, 2022.
- [49] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. Rsfuzzer: Discovering deep smi handler vulnerabilities in uefi firmware with hybrid fuzzing. pages 2155–2169, 2023.
- [50] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327, 2021.
- [51] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, Boston, MA, August 2022. USENIX Association.

APPENDIX A

A. Details of DXE Services

As shown in Listing 8, services are accessed (marked as ) through `EFI_SYSTEM_TABLE` pointer, passed as the second argument (marked as ) to the entry point of all UEFI components, *i.e.*, applications, drivers, OS loaders, etc. The services can be further classified into boot and runtime services.



- **Boot Services:** Boot services are used during boot time and unloaded when control is handed over to the operating system. These services provide the functionality necessary during booting – memory allocation, console input, and output, handling boot variables, file and device I/O (*i.e.*, USB or SATA drives), loading and starting UEFI applications/drivers, and managing the boot order. These services are accessed through `BootServices` field of the system table as indicated by  in Listing 8. These services are removed from memory by calling `ExitBootServices()` function, which is usually done at the end of the BDS phase, right before the operating system takes over control. In all our examples, we use `gBS` to indicate `ST->BootServices`. For

TABLE V: UEFI Components and Categorization of Known Vulnerabilities.

DXE Interfaces		Vulnerabilities		
Type	Percentage Contribution	Memory Corruption	Others	Total (% of Cummu)
Services (Boot and Runtime)	30%	3 (7%)	39 (97%)	41 (29%)
Protocols	70%	48 (49%)	50 (51%)	98 (71%)
Cumulative (Cummu)				139

instance, `ST->BootServices->foo` will be represented as `gBS->foo`.

- **Runtime Services:** These services provide functionality that is available even after the OS takes control. These services provide access to non-volatile EFI variables (*e.g.*, `BootOrder`), system’s real-time clock, hardware resets, etc. In general, Runtime Services provides a generic interface to access platform-dependent resources. These services are accessed through `RuntimeServices` field of the system table as indicated by  in Listing 8. Similar to `gBS`, we use `gRS` to indicate `ST->RuntimeServices`.

B. Vulnerability Study

We studied previously reported vulnerabilities (over past 14 years) to understand the prevalence of different kinds of vulnerabilities in different DXE interfaces. The left half of Table V summarizes the two types of interfaces and their contribution to the total interfaces. As mentioned before (§ II-A2), services are fixed by the UEFI standard. Consequently, any UEFI-compatible environment has a fixed set of services. Expectedly, protocols comprise the majority (*i.e.*, 70%) of the DXE interfaces. We categorized vulnerabilities into memory corruption (*e.g.*, buffer overflow, use-after-free, etc.) or other (*e.g.*, incorrect configuration). The right half of Table V summarizes vulnerabilities and the interfaces responsible for

Listing 7 Example demonstrating Protocol Registration.

```

1  →1 gEfiPxeBaseCodeProtocolGuid = \
2      { \
3          0x03c4e603, 0xac28, 0x11d3, \
4          {0x9a, 0x2d, 0x00, 0x90, \
5          0x27, 0x3f, 0xc1, 0x4d } \
6      }
7  →3 EFI_STATUS EfiPxeBcUdpRead
8      (
9      ...
10     IN OUT EFI_PXE_BASE_CODE_UDP_PORT *SrcPort,
11     IN OUT UINTN *BufferSize, IN VOID *BufferPtr
12     ...
13     ) { ... }
14     EFI_PXE_BASE_CODE_PROTOCOL
15     →2 gPxeBcProtocolTemplate = {
16         ...
17         EfiPxeBcMtftp,
18         EfiPxeBcUdpWrite,
19         →3 EfiPxeBcUdpRead,
20         ...
21         NULL};
22     ...
23     gBS->InstallMultipleProtocolInterfaces (
24         ...
25         →1 &gEfiPxeBaseCodeProtocolGuid,
26         →2 &gPxeBcProtocolTemplate,
27         NULL);

```

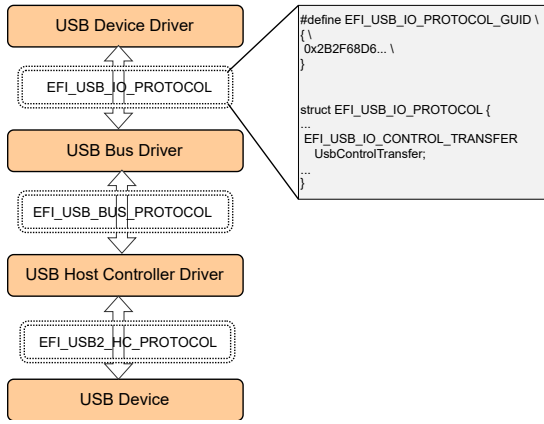


Fig. 6: USB Device Model Overview

them. We are able to conclude that 71% of the vulnerabilities are in the protocols and 49% of those vulnerabilities are due memory corruption bugs.

C. Driver Customization Example

The Figure 6 shows the high-level layered organization of the USB device driver. Each layer interacts with the other layer through a protocol. The top device driver layer interacts with the bus driver through `EFI_USB_IO_PROTOCOL`. Similarly, the controller interacts with the device through `EFI_USB2_HC_PROTOCOL`. The use of such well-defined interfaces allows vendors of a new device to easily customize the USB driver by exposing corresponding protocols.

As indicated in Figure 6, `EFI_USB_IO_PROTOCOL_GUID` and `struct EFI_USB_IO_PROTOCOL` represents the GUID and interface for `EFI_USB_IO_PROTOCOL` protocol, respectively. There are several predefined protocols with fixed GUIDs. Vendors can customize the standard protocols and define custom protocols with new GUIDs, e.g., `EFI_USB_IO_PROTOCOL_GUID` is the fixed GUID for the USB IO protocol.

D. FIRNESS Implementation Example

Listing 11 shows the snippet for the call site of the `SetPackets` function in Listing 1. The snippet shows the argument information for `Arg_12` (Argument 12), the `Packet` argument. `arg_dir = IN` indicates the argument is input. `arg_type` indicates the data type of the argument.

Listing 8 Examples of using DXE Services.

```

1  EFI_STATUS EFIAPI
2  EntryPoint (IN EFI_HANDLE IH,
3             IN EFI_SYSTEM_TABLE* ST)
4  {
5      EFI_STATUS St;
6      EFI_PHYSICAL_ADDRESS BufPtr;
7      void *data;
8      ...
9      St = ST->BootServices->AllocatePages(...,
10                                         &BufPtr);
11      ...
12      St = ST->RuntimeServices->GetVariable(...,
13                                           &data);
14  }

```

Listing 9 Path Constraint Example for Error Handling Code

```

EFIAPI
EfiPxeBcUdpRead (
    IN     EFI_PXE_BASE_CODE_PROTOCOL *This,
    IN     UINT16                      OpFlags,
    IN OUT EFI_PXE_BASE_CODE_UDP_PORT *DestPort,
    IN     UININ                       *HeaderSize,
    IN     VOID                        *HeaderPtr,
    IN OUT UININ                       *BufferSize,
    IN     VOID                        *BufferPtr
)
{
    ...
    if (This == NULL) return EFI_INVALID_PARAMETER;

    Private = PXEBC_PRIVATE_DATA_FROM_PXEBC (This);
    Mode    = Private->PxeBc.Mode;

    ...
    if (((
        (OpFlags & PXE_BASE_CODE_ANY_DEST_PORT) == 0)
        && (DestPort == NULL)) ... )
        return EFI_INVALID_PARAMETER;

    if (((HeaderSize != NULL) && (*HeaderSize == 0))
        ((HeaderSize != NULL) && (HeaderPtr == NULL)))
        return EFI_INVALID_PARAMETER;

    if ((BufferSize == NULL) (BufferPtr == NULL))
        return EFI_INVALID_PARAMETER;

    if (!Mode->Started) return EFI_NOT_STARTED;
    ...
}

```

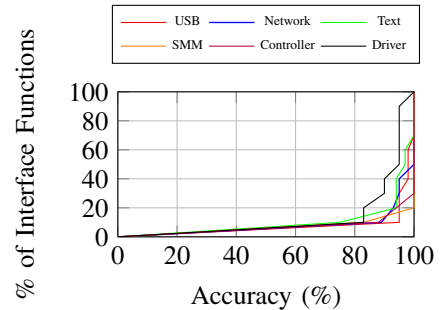


Fig. 7: CDF of Accuracy vs. % of Interface Functions

This might be different from the declared type of the corresponding parameter (i.e., `param_type`) because (as explained in § IV) the parameter can be a generic type (i.e., `void*`). The `potential_outputs` field indicates the possible definitions of the argument that can reach the call site. In the target example (i.e., Listing 1), there is only one definition of the argument (i.e., `Packet`) that can reach the call site. Hence, `potential_outputs` field contains only one entry, i.e., `GetInfo`. However, as illustrated in Listing 12, if there were multiple definitions that can reach the call site, then `potential_outputs` field will contain `["ReadPacket", "GetInfo"]`. The other fields (e.g., `usage`, `variable`, `Include`) will be used by the harness generation to emit the target harness in C.

E. Design Rationale

Our goal is to automatically generate fuzzing harness in source code (i.e., C code) form so that the harness can be generated once for a given protocol and compiled to different architectures (e.g., ARM). This also allows the generated har-

Listing 10 Snippet of Generated Harness for SetPackets in Listing 1

```
EFI_STATUS
EFI_API
FuzzSetPackets(
    IN INPUT_BUFFER *Input,
    IN EFI_SYSTEM_TABLE *SystemTable,
    IN EFI_HANDLE *ImageHandle
    ...
    EFI_PXE_BASE_CODE_PROTOCOL * ProtocolVariable = NULL;
    Status = gBS->LocateProtocol(&gEfiPxeBaseCodeProtocolGuid, NULL, (VOID *)&ProtocolVariable);
    /// Input Variable Initialization
    BOOLEAN * SetPackets_Arg_1 = AllocateZeroPool(sizeof(BOOLEAN));
    EFI_PXE_BASE_CODE_PACKET * SetPackets_Arg_12 = AllocateZeroPool(sizeof(EFI_PXE_BASE_CODE_PACKET));
    /// Fuzzable Variable Initialization
    ReadBytes(Input, sizeof(SetPackets_Arg_1), (VOID *)SetPackets_Arg_1);
    /// Generator Function Initialization
    EFI_MTF6_PROTOCOL * ProtocolVariable2 = NULL;
    Status = gBS->LocateProtocol(&gEfiMtf6ProtocolGuid, NULL, (VOID *)&ProtocolVariable2);
    EFI_MTF6_OPTION * EFI_MTF6_PROTOCOL_GetInfo_Arg_5 = AllocateZeroPool(sizeof(EFI_MTF6_OPTION));
    /// Generator Struct Variable Initialization
    ReadBytes(Input, sizeof(EFI_MTF6_PROTOCOL_GetInfo_Arg_5->OptionStr),
        (VOID *) (EFI_MTF6_PROTOCOL_GetInfo_Arg_5->OptionStr));
    ReadBytes(Input, sizeof(EFI_MTF6_PROTOCOL_GetInfo_Arg_5->ValueStr),
        (VOID *) (EFI_MTF6_PROTOCOL_GetInfo_Arg_5->ValueStr));
    Status = ProtocolVariable2->GetInfo(
        ...
        EFI_MTF6_PROTOCOL_GetInfo_Arg_5,
        EFI_MTF6_PROTOCOL_GetInfo_Arg_6,
        (EFI_MTF6_PACKET **) &SetPackets_Arg_12
    );
    Status = ProtocolVariable->SetPackets(
        ProtocolVariable,
        SetPackets_Arg_1,
        ...
        SetPackets_Arg_12
    );
);
```

Listing 11 FIRNESS analysis results for SetPackets function in Listing 1.

```
[
{
  "Arguments": {
    "...",
    "Arg_12": {
      "arg_dir": "IN",
      "arg_type": "EFI_PXE_BASE_CODE_PACKET",
      "param_type": "EFI_PXE_BASE_CODE_PACKET",
      "potential_outputs": ["GetInfo"],
      "usage": "&Packet",
      "variable": "Packet"
    }
  },
  "Function": "SetPackets",
  "Include": [ ],
  "ReturnType": "VOID",
  "Service": "protocol"
}
]
```

Listing 12 Example with branching

```
if(input == 1){
    ReadPacket(..., &Packet);
} else {
    Mtf6Prot->GetInfo(
        ...,
        (VOID **) &Packet
    );
}
```

Listing 13 Type Resolution for Generator Function Resolution

```
typedef CHAR16 *EFI_STRING;

/// Potential Generator Function
EFI_STATUS
EFI_API
HttpBootFormExtractConfig (
    IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN CONST EFI_STRING Request,
    OUT EFI_STRING *Progress,
    OUT EFI_STRING *Results
);

/// Target Function
EFI_STATUS
EFI_API
HiiStringToImage (
    IN CONST EFI_HII_FONT_PROTOCOL *This,
    IN EFI_HII_OUT_FLAGS Flags,
    IN CONST EFI_STRING String,
    IN CONST EFI_FONT_DISPLAY_INFO *StringInfo,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
    OUT EFI_HII_ROW_INFO **RowInfoArray,
    OUT UINTN *RowInfoArraySize,
    OUT UINTN *ColumnInfoArray
);
```

ness to be shared across multiple vendors working on different platforms and architectures. This requirement of source-level harnesses guided most of our design decisions, e.g., using L-value expressions and ignoring certain definitions (§ V-B3).

We acknowledge that performing our analysis on simpler representations, such as LLVM IR, could potentially enable us to perform more sophisticated analyses. However, using such low-level representations makes it very challenging to emit compilable C source-level harnesses because of the lack of appropriate type names (*i.e.*, pre `typedef` expansion) and other source-level artifacts.

F. Harness Generation Example

The goal is to create well-formed values of each argument from input and invoke the target function. For each argument, we identify the final argument type (*i.e.*, majority type as explained in § V-B5b) and determine whether it is scalar or not. If scalar, we declare a variable of the corresponding type and read the corresponding number of bytes from the input using `ReadBytes`. For `struct` types, if it is creatable, we create a variable of the `struct` type and initialize the contents with input using `ReadBytes`. If a `struct` has generators, we invoke the generator by creating arguments of appropriate type. When there are multiple ways to create an argument, we select one based on the value from the input. For instance, consider that a `struct` type t is creatable and also has 3 generators, which results in a total of 4 ways to create a value of t , *i.e.*, 3 generators and 1 from input data. We create a switch case statement with 4 cases (*i.e.*, `case 0`, `case 1`, ..., `case 3`) with three generators under three cases and reading from input (*i.e.*, `ReadBytes`) under one case. We read a byte from input and jump to one of the cases by using the byte in the switch condition, *i.e.*, `switch(input_byte % 4)`. Our method, in addition to being random selection, also enables deterministic testing, *i.e.*, a given stream of input bytes always results in a specific execution path. The Listing 10 shows the snippet of the C harness generated by our technique.

G. Common Generator Function Failure

The most common failure when capturing generator functions is when the type of interest gets resolved to a type that doesn't require a generator function (*e.g.*, scalar). An example of this can be seen in Listing 13, where the type of interest is `EFI_STRING` that can be resolved to `CHAR16*`, yet it is a unique type that has 15 generator functions that don't get captured. The current analysis resolves all types to their root type to eliminate ambiguity with type classification, the only exception is when the underlying type is `void*`.

H. Path Constraint Example

Some of the protocols take scalar values that are actually constants, but instead of passing in the constants there are checks at the beginning of each function to verify they are the correct value. These constants aren't captured by our analysis, and therefore, would limit the coverage if the checks can't be passed within the 24hr test period. For example, in one of the PXE boot protocol functions, `EfiPxeBcUdpRead`, a number of checks are performed at the beginning of the function that can't be passed without the private data structure for that protocol to be modified or the correct constant to be passed in to compare against the macro, Listing 9.

A. Description & Requirements

The artifacts are contained inside several Github repositories, which are submodules of the main project, linked below. Each of the submodules contains an explanation of what is within it to provide better context, but the main repository is capable of building the necessary docker container to run experiments within. The artifacts are also located in permanent storage at Zenodo, also linked below.

1) *How to access:* All of the source-code for FUZZUER can be found at permanent storage <https://doi.org/10.5281/zenodo.14257287> or at the Github Repository <https://github.com/BreakingBoot/FuzzUEr.git>.

2) *Hardware dependencies:* None.

3) *Software dependencies:* Requires `docker`.

4) *Benchmarks:* None.

B. Artifact Installation & Configuration

After downloading the files from Github as described above, you can then follow the instructions inside the README inside the main repository. If using Zenodo, you will need to uncompress the artifacts before following the README.

C. Experiment Workflow

N/A

D. Major Claims

N/A

E. Evaluation

N/A

F. Customization

N/A

G. Notes

N/A