Received 23 May 2023, accepted 15 June 2023, date of publication 21 July 2023, date of current version 31 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3297888

RESEARCH ARTICLE

Fuzzability Testing Framework for Incomplete Firmware Binary

JIWON JANG[1], GYEONGJIN SON[2], HYEONSU LEE[2], HEESUN YUN[2], DEOKJIN KIM[3], SANGWOOK LEE[3], SEONGMIN KIM[2], AND DAEHEE JANG[4], (Member, IEEE)

[1]Future Convergence Technology Engineering, Sungshin Women's University, Seoul 02844, Republic of Korea
[2]Convergence Security Engineering, Sungshin Women's University, Seoul 02844, Republic of Korea
[3]The Affiliated Institute of ETRI, Daejeon 34129, Republic of Korea
[4]School of Computing, Kyunghee University, Seoul, Gyeonggi 17104, Republic of Korea

Corresponding author: Daehee Jang (daehee87@khu.ac.kr)

This work was supported in part by The Affiliated Institute of Electronics and Telecommunications Research Institute (ETRI) under Grant 2023-062; in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant 2021R1F1A1049957; in part by the Korea Institute for Advancement of Technology (KIAT) Grant funded by the Korean Government through Ministry of Trade, Industry and Energy (MOTIE) (The Competency Development Program for Industry Specialist) under Grant P0008703; and in part by MSIT under the ICAN [Information and Communications Technology (ICT) Challenge and Advanced Network of Human Resource Development (HRD)] Program supervised by the Institute of Information and Communication Technology Planning and Evaluation (IITP) under Grant IITP-2022-RS-2022-00156310, and Convergence security core talent training business support program IITP-2023-RS-2023-00266615.

ABSTRACT Fuzzing is a practical approach for finding bugs in various software. So far, a number of fuzzers have been introduced based on new ideas towards enhancing the efficiency in terms of increasing code coverage or execution speed. The majority of such work predicates under the assumption that they have sound executable binary or source code to transform the target program as a whole. However, in legacy systems, source codes are often unavailable and even worse, some binaries do not provide a sound executable environment (e.g., partially recovered firmware). In this paper, we provide FT-Framework: *fuzzability* testing framework based on *forced execution* for binaries such as firmware chunks recovered in abnormal way so that they are hard to execute/analyze from intended booting phase. The essence of our work is to automatically classify functions inside a binary which we can apply coverage-guided fuzzing via forced execution. We evaluate FT-Framework using PX4 and ArduPilot firmwares which is based on 32-bit ARM architecture and demonstrate the efficacy of this approach and limitations.

INDEX TERMS Fuzzability, firmware binary, coverage-guided fuzzing, fuzzable function, binary fragment, emulation based fuzzing.

I. INTRODUCTION

Since the introduction of coverage guided fuzzers [1], [2], fuzzing became one of the most practical techniques to discover software bugs and ultimately security vulnerabilities [3]. Instead of relying on complex analyses, fuzzing excavates vulnerabilities in a program by repeatedly executing it with auto-generated inputs while monitoring code coverage information and unexpected behaviors. Thanks to its practicality, fuzzing techniques have been applied to complex, real-world software such as browsers [4], [5], [6] and kernels [7], [8], [9], leading to numerous zero-day vulnerabilities. All such previous works provide their respectful insights in terms of improving the efficacy of fuzzers to better find crashes and rapidly explores code coverage, and so forth. However, they mostly require full source code of the target program to apply their idea. Such techniques often transform the entire program using compiler techniques, and require full access to the hardware/software environment to run the target program. Although uncommon, in legacy systems, we often encounter situations where source code is unavailable and even worse, the binary is incomplete to execute (e.g., lack of library files, hardware component, booting script, etc). For example, the firmware recovered via forensic techniques are unsound [10]. Figure 1 depicts hypothetical scenario where Unmanned Aerial Vehicle (UAV) is claimed in military operation and their firmware is partially recovered.

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

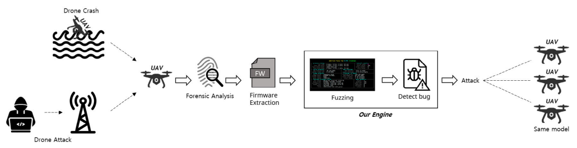


FIGURE 1. A scenario where UAV is claimed in military operation due to crash/hijack and their firmware being forensically recovered. In such scenario, the recovered binary could be incomplete to execute.

Hypothetically, say, such firmware can only be analyzed statically as there is no proper hardware/software (emulator) environment to execute the firmware. Under such premise, we investigate the following research questions: (i) to what extent can we apply forced execution? (ii) how can we automatically identify codes that we can apply forced execution? (iii) can we apply coverage-guided fuzzing technique via forced execution? In this paper, we introduce FT-Framework: *fuzzability* testing framework based on forced execution. High-level methodology is as follow: First, FT-Framework trace all function's execution paths to check external dependency (e.g., system call). If a function (and all possible paths) does not have such dependence, we refer to it as *Executable Function Fragment (EFF)*. Second, we automatically profile *EFF*s to estimate if they have proper code structure to apply fuzzing and integrate with fuzzers, thus *fuzzable*. Finally, we import such *fuzzable* code fragments to coverage-guided fuzzer via AFL-QEMU without source code. This concept (attaching fuzzer to specific functions) is already established in libFuzzer development [2]. However, the key difference is that libFuzzer works based on source code and they require developer's detailed understanding for the target function's protocol; whereas we consider binary fragment without any prior knowledge other than information we gather from automated binary profiling.

To profile *EFF*, we examine all possible code paths of the function using the Depth-first search (DFS) algorithm. To determine if a function meets *EFF* criteria, we search for three cases that violate our purpose: (i) virtual call, (ii) unidentified system calls, and (iii) basic block containing interleaved data. If a function's entire code path (including sub-functions) do not contain any of such cases we accept it as *EFF*. With *EFF*s, we apply automated profiling algorithms. The key idea is to observe the forced execution result based on function's input parameter types. Ultimately, we classify *EFF*s depending on such responses and narrow down *fuzzable* functions. Then, we attach our stub program (as in libFuzzer development) to *fuzzable* functions and run fuzzer based on forced execution. The stub program is responsible for connecting the interface (e.g., parameter passing) between coverage-guided fuzzer and *fuzzable* function similarly to `LLVMFuzzerTestOneInput` entry point in libfuzzer.

For evaluation, we use PX4 and ArduPilot UAV firmware that are based on 32-bit ARM architecture. Additionally, we test FT-Framework using Objdump of GNU Binutils binaries as well. Through the experiments on three types of commodity firmware and binary, we confirmed that FT-Framework enables opportunities of coverage-guided fuzzing against closed-source and incomplete firmware binaries by recovering its executable chucks. Based on analysis, we discuss further issues to be explored towards improving the performance of fuzzability testing in two perspectives: 1) how to expand the coverage of forced execution and 2) how to handle false-positive crashes.

The rest sections of the paper continue as follow: we describe some backgrounds to better understand this paper and summarize related works in section §II. In §III, we explain detailed description and high-level explanation of FT-Framework. In §IV and §V, we describe implementaion details and its evaluation results regarding FT-Framework. Finally, §VI discusses limitations and lessons learned, and concludes in §VII.

II. BACKGROUND & RELATED WORK

A. AFL & AFL-QEMU

AFL is one of popular fuzzers optimized for increasing code coverage, using genetic algorithm and mutation input data properly, record changes to program control flows, and log them to discover crashes. AFL by default requires source code for instrumentation. AFL-QEMU, is a variation of AFL to support binary only fuzzing based on QEMU. In AFL-QEMU mode, code coverage is measured based on QEMU basic-block translation mechanism and do not require special compilation step.

B. COVERAGE-GUIDED FUZZING

Coverage-guided fuzzing has been recognized as an effective software security test method in gray-box fuzzing. Coverage refers to a measure of how much code has been executed by measuring the execution flow of a program, and basic block and edge coverage are commonly used as measures. Coverage-guided fuzzing is useful because it has the characteristic of automatically generating inputs in programs to identify inputs that reach new parts and trigger exceptions or vulnerable parts. Specifically, genetic algorithms generate new inputs for each fuzzing round, which allows more bugs to be discovered. AFL makes representative coverage-guided measurements, and it has attained excellent results in several studies. Several studies have proven excellent effects using coverage-guided fuzzing [11], [12], [13], [14], [15].

It increases the probability of creating a new collision by creating a new input to increase coverage. Methods capable of measuring code coverage are often achieved through instrumentation [16]. The instrumentation code inserted together at the time of compilation can verify which part a fuzzer has searched for and which part has not been reached. The instrumentation code has a characteristic that some overhead occurs. Studies on how to handle this effectively are being actively conducted [17], [18], [19].

C. EMULATION-BASED FUZZING

Several previous studies have adopted emulation methods to fuzz a special environment like firmware binary. As firmware runs, it exchanges numerous interactions with hardware in real time. Emulation solves areas that must interact with real hardware, such as requesting access to peripheral devices [13], [20], [21], [22], [23], [24], [25].

The emulation of firmware often uses QEMU. The QEMU has an overall system-emulation mode and a user-emulation mode. The user-emulation, which can emulate a specific part of the firmware, performs firmware emulation-based fuzzing.

Several problems must be solved to emulate firmware. Classic problems with firmware-emulation fuzzing include performance-degradation problems and compatibility problems caused by emulation. Firm-AFL [21] solved this through augmented process-emulation technology. Avatar [24], a famous framework, developed technologies to improve system performance. Avatar, a combination of hardware and software emulation, reduced the occurrence of problems by processing memory access when analyzing device-dependent firmware.

D. ArduPilot & PX4

ArduPilot is an open-source project to control UAVs [26]. PX4 [27] (Pixhawk project) is also an open-source UAV software that was developed as a sub-project of ArduPilot. These two software are famous in UAV market and their codebase is large, thus we use these software as our test target firmware.

E. FIRMWARE EXTRACTION

The simplest method for extracting firmware is to download it if the manufacturer discloses and provides it via web-site. However, this is often not the case; in such case, analyst utilizes code-extracting tools or firmware-analysis tools. A famous tool for firmware analysis is Binwalk. Binwalk uses a file signature to verify the contained data. Because firmware-binary extraction is quite difficult, various researches exist solely for this direction. For instance, firmadyne [28] is a previous study that supports extracting and analyzed Linux-based embedded firmware binary in efficient way.

F. PARTIAL BINARY FUZZING

The AFL is designed to fuzz programs with source codes. Therefore, if there is only a binary of the object to be fuzzed, a new methodology should be developed instead of the original widely used method.

This situation has several names, such as closed-source code, blob, and professional device (software). This situation is related to intellectual property rights or can occur in specific scenarios such as our paper. Previous studies dealing with these special situations include software from major companies that do not disclose source codes, WINNIE [29], which fuzzed Windows programs, and APICraft [30], which fuzzed macOS SDK, a toolkit used for app builds for MAC. Additionally, papers preparing for the environment, which are various closed-source codes, are being actively studied [31], [32], [33].

Various attempts are being made to fuzz firmware binary covered in our paper. The need to analyze firmware binaries has increased because the era of IoT has opened, and various unmanned mobile devices have been discovered. The most important thing in an environment where firmware binary is fuzzing is that it is a closed-source code and is significantly influenced by hardware. There are several studies, including studies dealing with UEFI Firmware [11], Firmware Baseband [34], [35], [36], and IoT [12], [20], [21], [22].

Furthermore, there has been a significant amount of research focused on fuzzing only specific parts of binary. Partial binary fuzzing refers to intensively fuzzing the desired part of the user instead of fuzzing the entire system. If fuzzing is performed on the entire system, temporal and computing resources are unnecessarily consumed. At this time, efficient bug search is possible by performing binary fuzzing for a specific part [37]. To increase coverage, the method of weighing on less visited coverage [34], partial binary fuzzing can be performed in numerous ways, such as identifying areas where bugs are expected to occur and performing them further [37], [38], [39].

It is also necessary when the entire emulation is unrealistic. For example, when it is restored by forensics or when the binary is partially damaged, partial binary fuzzing may need to be performed. Therefore, a method for fuzzing only a specific part of the entire system is being studied. Previous studies have proposed a method for performing fuzzing by designating the part to be explored [40]. Our study performs fuzzing on the functional unit of the part responsible for a specific function of the entire system. Through this, it is possible to prepare for situations in which binary is not intact.

On the other hand, executing partial binaries can cause False-Positive problems, which poses a major challenge and research question from the perspective of accurate bug detection. Specifically, False-Positives might occur when control flow information for target code is not sufficient. For example, without understanding the semantic of preprocessed input data (in parent function) before reaching the target function (fuzzed one), it is hard to determine if a crash is indeed a bug or if the given input can be passed to the target function in real situation. In this study, we partially addressed such False-Positive issue by adding a preprocessing stage that heuristically guessing the expected input format through parameter profiling(See §III-C).

FOCAL is a recently proposed Concolic software testing methodology, which is a Concolic testing technique for detecting software bugs, and it is a study that increases efficiency in bug exploration by hybridizing unit tests and system-level tests. In this research, unit testing is performed for all functions within the program, but this approach also raises false alarms mainly due to parameters exceeding the expected range. FOCAL tries to eliminate such

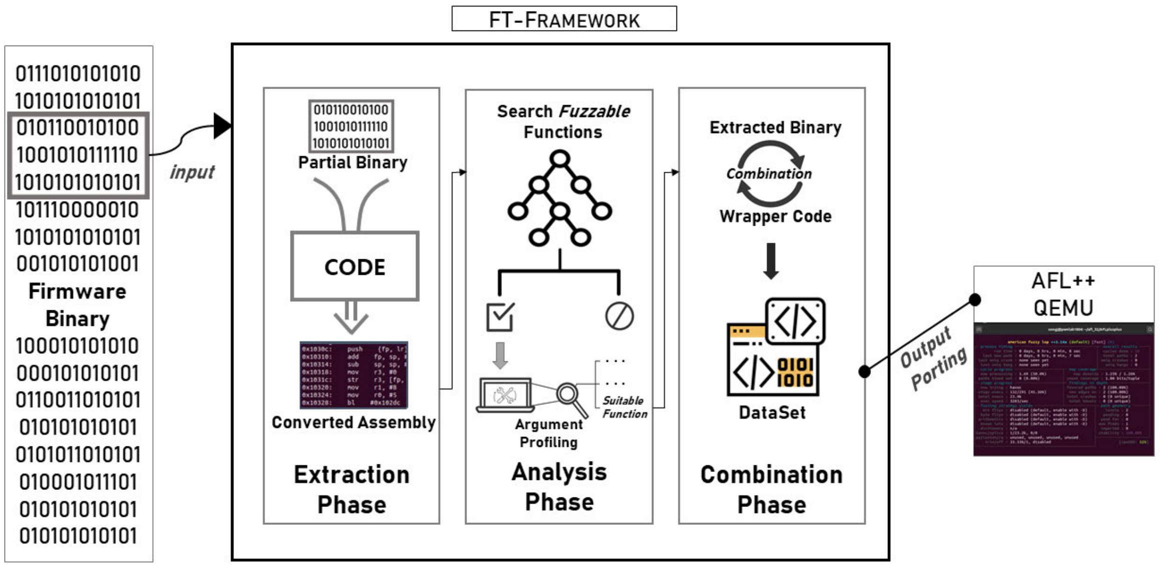


FIGURE 2. FT-Framework overview. The target firmware binary is given to FT-Framework to apply fuzzing against small portions of the binary. There are three phases in processing the binary fragment for fuzzing. First, we pre-process the binary to enumerate all functions. Second, we determine if they are *fuzzable* in our perspective. After analysis, selected functions are ported to apply coverage-guided fuzzing.

'infeasible unit execution' problem with their idea 'context stitching'.

Within the FOCAL Framework, there is a phase that measures the relevance between functions by analyzing functions that are frequently called in close sequence and assume they have high relevance (i.e., it would be a caller-callee relationship). Based on this, the previous functions are gradually inferred, the size is extended a bit from the initial single function, and the candidate input (which may have been a false alarm) that caused the failure is tested repeatedly. We added discussion in §VI regarding how we can utilize FOCAL idea with our approach.

G. PARAMETER PROFILING

A parameter profiling infers a parameter of a function. This task is conducted to provide the correct input seed based on the format. For instance, unlike general variables, a pointer has an address value and utilizes this address; thus, an address value that does not exist should not be defined. Because there is such a difference in expected input values, errors may occur if the pointer is not identified. Previous studies for identifying the input format include FuzzGen, WINNIE, PMP, and Ramblr [29], [41], [42], [43].

FuzzGen is a study of Linux systems and is a tool that automatically matches the interface of the expected API. FuzzGen deliberately continues to summon calls for api to infer the effective api interaction, and through this process, the fuzzer stub is configured to match the expected interface [41]. WINNIE, which performed a study on Windows systems, deduced the number and type of parameters using parameter static and dynamic analyses, and executed more than twice for an argument that was preemptively considered a pointer, considering the characteristics of aslr. If different addresses were indicated, a method for identifying pointers through heuristics considered to be pointers was taken [29]. PMP is also a method for considering a pointer if the function to be determined points to somewhere in the memory address using heuristic [42]. Reassembly, a technology used to make a new program that includes the binary of closed-source code situations as an additional function, requires a task to infer parameters. This is because symbolization, in which the relative symbol is readjusted according to the absolute address, must be performed to rewrite the desired binary. This is a task that has problems to be solved, and Ramblr presented a detailed methodology for solving it [43]. For successful symbolization, a content classification, such as a profiling operation is performed. By recovering and analyzing binary CFGs (control flow graphs), instructions and data access patterns are analyzed to form a jump table, and through this process, data types, such as pointers, integers, shorts, floats, and doubles are identified.

Parameter profiling towards closed-source code is being actively studied. This study classifies functions with high crashable possibility by verifying the results of function execution based on candidate and inferring parameters and internal structure of functions. Through this work, meaningless fuzzing targets were selected.

III. DESIGN

In this section, we explain the overall design and provide detailed descriptions of FT-Framework. Figure 2 shows the overall description of FT-Framework. There are three phases in FT-Framework: extraction, analysis, combination phases.

A. EXTRACTION PHASE

In the extraction phase, we use existing firmware extraction tools to correctly disassemble the target binary for proper analysis to classify functions. Most firmware including our evaluation target (PX4, Ardupilot) is often compressed/encoded with various packing algorithms. Decompressing/decoding such binary fragments is challenging because we do not have the complete information/environment of the binary.

Accurately recovering the entire binary from undocumented data format is a challenge itself; however, if our goal is running a function via forced execution, complete code dependency is not required. One of a major problem in incomplete binary information is *interleaved data problem*. This issue is not a problem in other binary-analysis frameworks where they have complete binary. We provide a detailed explanation of this issue in the following subsections.

B. ANALYSIS PHASE

The analysis phase determines if a function is classified as *EFF*. For reminder, *EFF* means that all reachable codes for the function do not depends on external/additional binary (i.e., runtime shared library such as Windows DLL or Linux binaries) or external data source other than memory (e.g., specific I/O operation, network/disk-related system calls). A function is classified as *EFF* if all instructions in potential execution paths are independently executable satisfying the aforementioned requirements.

To find *EFF*, FT-Framework utilizes blacklist approach by identifying functions that violate the classification requirement and removing them from the *EFF list* that initially contains entire set of functions. For classification, we consider basic block as vertex and branch as edge of graphs and run DFS algorithm to enumerate the entire code path of a function. The function address is added to the blacklist if we discover any violation of our rule. This blacklist-based approach speeds up the classification performance. For example, if we categorize a function as a blacklist, all other functions internally nesting such function can be automatically categorized as blacklist. Classification results into three types.

Figure 3 describes three result cases in FT-Framework. In `Case1` a function does not encounter any case we consider as blacklist. Such functions can execute its internal set of instructions without depending on external code/data source. These functions are interesting in our perspective and we mount them with fuzzers. `Case2` can run independently with minor effort. For example, some system calls are trivial in terms of fuzzing. In such cases, we can simply remove/patch/emulate the calls and continue the execution without affecting the fuzzing process. A representative example would be getting time information via system call for debug/logging purposes. `Case3` is a blacklist case which FT-Framework does not consider for fuzzing.

There are three main reasons we filter out a function from being *EFF*: virtual call, system call, and interleaved data.

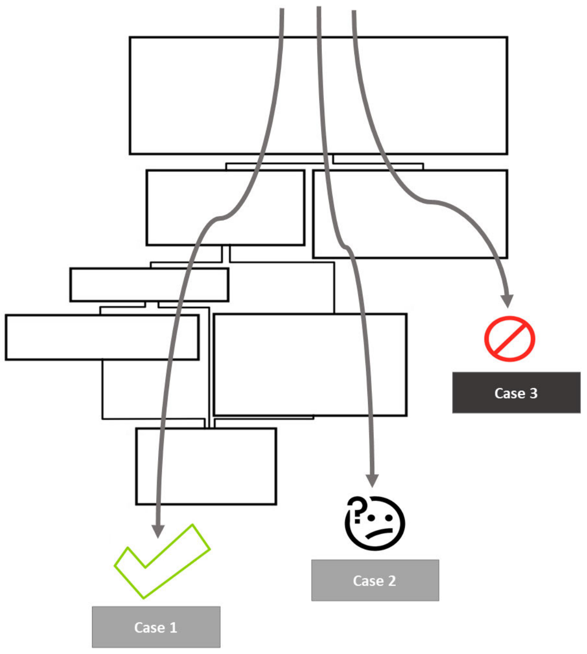
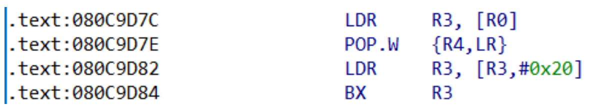


FIGURE 3. ***Fuzzability*** **test of FT-Framework. The figure shows three cases of analysis results. The first case does not encounter any blacklisted situation; thus, it is categorized as** ***EFF*****. The second case is problematic but can be solved using a minor patch or heuristic approach. The third case fails independent execution owing to blacklisted instructions.**



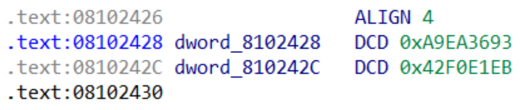
```
.text:080C9D7C                LDR     R3, [R0]
.text:080C9D7E                POP.W   {R4,LR}
.text:080C9D82                LDR     R3, [R3,#0x20]
.text:080C9D84                BX      R3
```

FIGURE 4. Virtual call case in ARM disassembly. Branch target in `R3` is calculated at runtime and it is difficult to estimate its value via static analysis.

1) VIRTUAL CALL

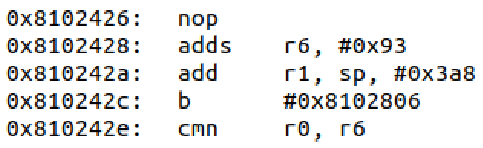
Virtual call is a representative case that cannot be determined as *EFF* through FT-Framework. In virtual call, the invoked function's address is dynamically calculated at runtime. For virtual calls, the target address of the invoked function is difficult to deduce via static analysis because the target branch address is dynamically calculated using memory and register operations at runtime.

Figure 4 is an example of virtual call in ARM architecture. In the figure, program branches to a target memory address stored inside `R3` register and its value is retrieved from the memory address indicated by `R3` with offset 0×20. Data inside such a memory address is unknown at the static analysis time and determined at runtime. However, it is difficult to assess if such virtual call will jump to the in-bound region of the given binary fragment. Therefore, in virtual call, we cannot determine if the function will normally execute with the given set of binary. Consequently, FT-Framework classifies it as a blacklist function.



```
.text:08102426                    ALIGN 4
.text:08102428 dword_8102428      DCD 0xA9EA3693
.text:0810242C dword_810242C      DCD 0x42F0E1EB
.text:08102430
```

FIGURE 5. Interleaved data case in IDA Pro. If the binary is intact, identifying such interleaved data is relatively easy because there is cross-referencing information.



```
0x8102426:   nop
0x8102428:   adds    r6, #0x93
0x810242a:   add     r1, sp, #0x3a8
0x810242c:   b       #0x8102806
0x810242e:   cmn     r0, r6
```

FIGURE 6. Misinterpreted interleaved data. Branch target in the figure (0 × 8102806) is invalid. However, disassembly based on incomplete binary is considered a valid branch.

2) UNIDENTIFIED SYSTEM CALLS

System call falls into the blacklist category because its branch target is inside the operating system kernel. Generally, system calls allow user applications to execute the privileged commands by switching from user mode to kernel mode, and their code bases are separated from one to another. If the binary is based on a specific operating system, system call identification is non-trivial. For a typical POSIX-based operating system binary, some system calls might be easily interpreted. If system calls are identifiable, some of them can be trivially emulated (example, `getpid()`). However, in other environments, emulating/analyzing system call is non-trivial because they do not provide any documentation regarding their application binary interface (ABI). Therefore, FT-Framework treats system calls as blacklist cases. However, kernel code can be merged as a single statically linked code in embedded systems such as UAV environments as part of the firmware image. In that case, system calls can be treated same as ordinary function calls.

3) BASIC BLOCK CONTAINING INTERLEAVED DATA

Interleaved data is a data fragment inserted in the middle of a code. As the Figure 5 shows, this data/code layout is often observed in RISC-architecture-based compilers such as ARM. ARM compilers often locate data in the middle of the code segment because of the RISC CPU's memory-addressing feature.

Therefore, such interleaved data may be misinterpreted as a code in FT-Framework due to the lack of code boundary information. This issue is less problematic in generic binary-analysis tools because they have complete information regarding the entire binary layout and headers. However, for FT-Framework, this issue becomes difficult to handle because the binary can be incomplete and often lacks header information. Particularly, such a case becomes a problem if data is misinterpreted as branch instruction (i.e., data that coincidentally have the same value as `BL` instruction-code prefix in ARM architecture), as shown in Figure 6.

In such cases, FT-Framework explores a non-existent or wrong portion of the binary, and its analysis results become wrong. Therefore, accurately identifying interleaved data fragments inside a function code is a challenge in FT-Framework. We use a heuristic approach to classify instruction as interleaved data if (i) opcode is unknown and (ii) branch target address is outside the binary range while the opcode is a legitimate branch.

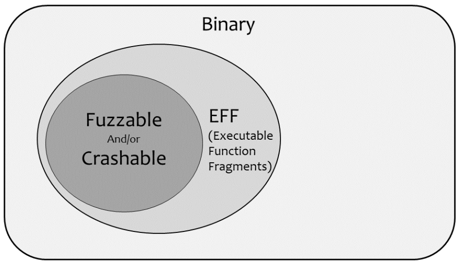


FIGURE 7. FT-Framework further classifies a function that can detect a crash with fuzzing technology in a function group previously classified as *EFF*, that is, crashable, through parameter profiling.

C. COMBINATION PHASE

After classifying *EFF*, crashable functions, FT-Framework continues to the next phase, which combines such functions to a coverage-guided fuzzer. FT-Framework provides a stub (wrapper) program similar to that in libFuzzer to connect the fuzzing interface to the target function. However, being *EFF*s is not enough to apply proper fuzzing because of additional dependencies such as referencing a global variable, referencing file. Also for proper fuzzing, a function must take some arbitrary input (e.g., via function parameter). Therefore, among *EFF*s, only the subset of them are additionally classified as *fuzzable*.

Figure 7 summarizes the fuzzable (or crashable) target categorization performed by FT-Framework. In our paper, a *fuzzable* function must take at least one pointer variable as its input argument. If the function satisfies such requirement, we pass fuzzer's input as a pointer via `R0` to `R3` using in-line assembly code which is a standard ARM parameter passing channel. FT-Framework uses *AFL-QEMU-ARM32* as fuzzing engine which can measure coverage based on binary without source code.

1) PARAMETER PROFILING

Parameter profiling is the process of inferring parameters through responses observed during the forced execution of the program. Profiling function parameters is required to narrow down *Fuzzable* functions among *EFF*s. With profiling, we additionally identify information regarding function parameter. For example, a function that does not receive input is meaningless for fuzzing because it does not take input.[1]

To profile a function parameter, we force executing a function with various parameters. For instance, if we force executing a function with read-only pointer as first argument

[1]We only assume inputs being passed via parameters, however, a function can take input in different ways such as global memory buffer. this is limitation of our paper.

and observe segmentation fault; whereas we do not observe such error with read/write pointer (pointing exact same memory contents), we can reasonably suspect the function is using first pointer argument as an *output pointer*.

IV. IMPLEMENTATION

Our implementation is based on Python3; Pwntool and Capstone library to disassemble the binary and obtain function information. After basic binary analysis, DFS algorithm is used to visit all basic blocks of the given binary. When a basic block contains a function call instruction, DFS recursively trace all the child functions.

A. EFF CLASSIFICATION

In our implementation, blacklists are considered into three categories.

The first category is a virtual call. If the disassemble result includes `bl`, `blx`, and `bx` instructions corresponding to a function call, the argument is also examined. If the argument is neither a link register (LR) nor a constant (address value), the instruction is considered as a virtual call.

Algorithm 1 Virtual Call

```
branches = ["bl", "blx", "bx"]
if $instruction \in branches$ then
    $operand = get\_operand(instruction)$
    if $operand \neq$ "$linkregister$" | $operand \neq constant$ then
        blacklist.append(operand)
    end if
end if
```

The second category is a system call. If the disassemble result includes `swi` and `svc` instructions, they are classified as the system call.

Algorithm 2 System Call

```
branches = ["swi", "svc"]
if $instruction \in branches$ then
    systemcall.append(address)
end if
```

To determine if an instruction is interleaved data, we use aforementioned heuristics: (i) opcode is unknown and (ii) branch target address cannot be referenced. Additionally, every upper functions of the three categories are also included in the blacklist.

B. PARAMETER PROFILING AND FUZZER PORTING

Profiling is conducted to improve the efficiency of the FT-Framework. Profiling classifies the function by gathering the results of its execution (forced) upon various parameter types. If it delivers the parameter form (data type) that the function does not expect to process, there is a high probability of errors such as crashes. FT-Framework can save time compared to fuzzing an entire binary by pre-screening functions that are not worth fuzzing through profiling. Functions other than *EFF* are not included in the profiling candidate. To force the execution of a target function, we write a wrapper code, which is responsible to map the target function as a dynamically loaded library (or, similarly to shellcode in computer virus) into memory and invoke a function pointer pointing the loaded memory.

Algorithm 3 DFS

```
function DFS($address, visited, blacklist$)
    if $address \notin visited$ & $address \notin blacklist$ then
        run DFS
    else if $address \in visited$ & $address \notin blacklist$ then
        pass one branch line
    else
        blacklist.append(visited function)    ▷ Add the address of the function and parent function
    end if
end function
```

The wrapper code is automatically compiled and executed with python script for profiling evaluation. Candidate parameter types for profiling are: (i) null, (ii) number, (iii) pointer, and (iv) double pointer. The pointer reference an arbitrary string, and the double pointer reference another pointer. The execution result for each parameter of the function can be inferred through the `python subprocess` return code. The code is zero for normal termination, `-11` for segmentation fault, and `-4` for incorrect CPU state, which is a characteristic of ARM binary. We gather these information in .csv file to confirm the results of each functions. In some cases, execution results could not be identified owing to infinite loops, and these cases are handled using a timeout. All functions are executed for 30 secondes for profiling, and functions that do not end within that time window are marked as `infinite`. Finally, we use a small wrapper code which articulates the target binary to AFL++ fuzzer.

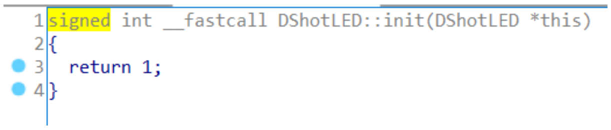
There are two ways in which AFL++ takes input: `stdin` and file. In the proposed methodology, input is received and processed in the form of a file. These processes proceed with binary mapping within the wrapper code. To transfer the input value received by the file as a function parameter, a memory buffer is created, file data is copied, and the pointer of the memory buffer is handed over to the target function to be fuzzed as a parameter. Finally, the fuzzing is initiated by delivering the address of the target function.

V. EXPERIMENTATION & EVALUATION

In this section, we evaluate the functionality of FT-Framework that can determine the *fuzzability* of *EFF*. We applied FT-Framework for UAV firmware binaries such as PX4, ArduPilot, based on the proposed methodology. In addition to firmware, we applied FT-Framework to Objdump binary in Binutils, a typical Linux binary. Table 1 is a result of *Analysis Phase* for PX4, ArduPilot, and Objdump binaries. C1, C2,

TABLE 1. Summary for PX4, Ardupilot, Objdump. This table is a result of *Analysis Phase* for PX4, Ardupilot, Objdump binary. C1, C2, and C3 in the column indicates virtual call, system call, and Interleaved data, respectively. In PX4 and ArduPilot, about half of the total number of functions were *fuzzable*. However, in case of Binutils, one third of functions are classified to be *fuzzable*.

	Total	EFF	Blacklist	C1	C2	C3
PX4	10,093	5,583	4,510	552	7	37
ArduPilot	9,400	4,879	4,521	1,393	11	19
Objdump	2,280	688	1,592	223	1	60



```
signed int __fastcall DShotLED::init(DShotLED *this)
{
  return 1;
}
```

FIGURE 8. Parameter profiling automatically filters out functions that do not interact with any parameters.

and C3 in the column indicates virtual call, system call, and interleaved data, respectively.

In PX4 and ArduPilot, approximately half of the total number of functions are *EFF*s. However, in the case of Binutils, one third of functions are classified to be *EFF*s. When we applied FT-Framework to PX4 binary, 552 out of 10,093 functions were classified as virtual call, seven as system call, and 37 as interleaved data cases, as shown in table1. The number of final blacklists including upper functions in the previous three cases was 4,510, the total number of the *EFF* was 5,583. The results of *EFF* functions through code-targeting ArduPilot was 4,879 of 9,400 functions. They were classified into 1,393 virtual call, 11 system call, and 19 interleaved data cases. Result for classifying *EFF* functions for Objdump binary shows 668 out of 2,280 functions. Among them, there were 223 virtual call, one system call, and 60 interleaved data cases. Notably, the sum of C1, C2, and C3 does not match with the blacklist number because a function nesting any of such cases is blacklisted without being counted.

A. PROFILING EFF

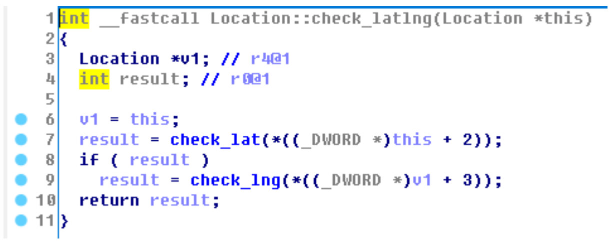
To profile *EFF*s, we select several types of input values, using them as function parameters in forced execution, and inferring parameters and functions with the execution results during the execution. Candidate parameters were given as null, number, pointer, and double-pointer types. The profiling results for *EFF* were classified into five categories based on the presence or absence of memory access violation as summarized in Table 2.

Case-1 are functions normally returned for all candidate parameter inputs, such as Figure 8.

Case-2, 3, 4 and 5 is summarized in the table respectively as well based on the function's response to the parameter type. O indicates the function returned normally upon the given parameter type, and X indicates function raised an error upon the given parameter.

TABLE 2. There are five cases classified in the function-profiling phase of the FT-Framework, where RW ptr is readable and writable pointer, O indicates function normally returns upon the given parameter type while X raises error.

Profile Type	null	number	R-only ptr	RW ptr	double ptr
Case-1	O	O	O	O	O
Case-2	X	X	X	O	X
Case-3	X	X	X	X	O
Case-4	X	X	O	O	O
Case-5	X	X	X	X	X



```
int __fastcall Location::check_latlng(Location *this)
{
  Location *v1; // r4@1
  int result; // r0@1

  v1 = this;
  result = check_lat(*((_DWORD *)this + 2));
  if ( result )
    result = check_lng(*((_DWORD *)v1 + 3));
  return result;
}
```

FIGURE 9. Function that returned normally upon all parameter types.

Case-2 is a function that shows normal results only for one-dimensional (e.g., byte buffer) pointer inputs capable of both read and write. The function that belongs to this case takes a one-dimensional array input. Based on the profiling results with a one-dimensional array that can only read, the case with both read and write pointers did not crash and showed 40 more normal responses based on the ArduPilot firmware. Therefore, it was possible to distinguish functions including write operations through these criteria.

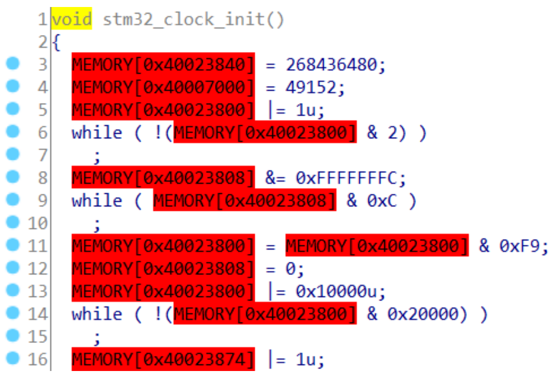
Case-3 is a function that shows normal results only for double-pointer inputs. It can be assumed that the corresponding function is a function that uses complex objects utilizing double pointers (e.g., C++ this pointer).

Case-4 is a function that returns normal results for all three inputs: a double-pointer, one-dimensional pointer that can read and write, and one-dimensional pointer that can only read. In this case, it shows the pattern shown in Figure 9, and it can be estimated that the function is likely to receive the pointer format.

Case-5 is a function in which a segmentation fault error occurs for all candidate parameters, likes Figure 10. In this case, it was suspected that the function depends on a global variable or other external data source.

Through profiling, we can select a set that is worth fuzzing. For example, in Case-1, the structure of the function is too simple thus it is unnecessary to apply fuzzing. In Case-5, additional analysis is required because the function is complicated and might depend on global variables or results from other data source thus excludes from *fuzzable* target.

The profiling experimental results are summarized in Table 3. It was possible to reduce the crashable candidates by 26% by excluding Case-1 (no parameters) from the total *EFF* list. As a result of profiling evaluation, functions in the firmware are classified as follows.



```
void stm32_clock_init()
{
  MEMORY[0x40023840] = 268436480;
  MEMORY[0x40007000] = 49152;
  MEMORY[0x40023800] |= 1u;
  while ( !(MEMORY[0x40023800] & 2) )
    ;
  MEMORY[0x40023808] &= 0xFFFFFFFC;
  while ( MEMORY[0x40023808] & 0xC )
    ;
  MEMORY[0x40023800] = MEMORY[0x40023800] & 0xF9;
  MEMORY[0x40023808] = 0;
  MEMORY[0x40023800] |= 0x10000u;
  while ( !(MEMORY[0x40023800] & 0x20000) )
    ;
  MEMORY[0x40023874] |= 1u;
```

FIGURE 10. Red blocks indicates global variable reference, In this case, the function is classified as Case-5.

TABLE 3. Fuzzable-function profiling result.

Profile Type	PX4	ArduPilot	Objdump
Case-1	692 (20%)	1,162 (33%)	133 (27%)
Case-2	74 (2%)	54 (2%)	0 (0%)
Case-3	148 (4%)	252 (7%)	94 (19%)
Case-4	407 (11%)	676 (19%)	6 (1%)
Case-5	2,225 (63%)	1,378 (39%)	252 (52%)
Total (Profiled as Fuzzable)	3,546	3,522	485

B. FUZZABILITY TESTING THROUGH FORCED EXECUTION

To evaluate the compatibility of FT-Framework, we conduct a forced execution against the profiled *fuzzable* functions with the state-of-the-art coverage-guided fuzzer, AFL++. Note that as we develop our stub program based on libFuzzer mechanism to handle *fuzzable* functions, coverage-guided fuzzers that utilize the library are also compatible with FT-Framework. In this experiment, *fuzzability* was thoroughly investigated for 3,522, 3,546, and 485 *EFF*s in the PX4, ArduPilot and objdump binaries, respectively.

Once we applied fuzzer, we largely observe four different types of responses: (i) normal fuzzing, (ii) fuzzer refuse to run, (iii) crash is discovered too shortly, (iv) fuzzer hangs. In our evaluation, we spent approximately 10 seconds for each function to apply fuzzing. In the third response case, we analyzed that the instant crash is mostly due to the fuzzer mangling the function's specifically expected parameter format (e.g., passing object/structure). Such a crash should not be considered as a bug. We also note that distinguishing such a crash from a real bug is outside the scope of this paper (We discuss this issue in §VI).

As an evaluation result, we show three tables (Table 4, Table 5, Table 6) that summarize the fuzzability testing for each functions in our target binaries. *Fuzzable* means successful porting to fuzzer, and *crashed* indicates fuzzer quickly crashed upon testing nevertheless successfully porting. And *not fuzzable* indicates fuzzer refused to run, the

TABLE 4. Experiment results of the PX4 functions.

	Fuzzable		Not Fuzzable	Total
	Normal	Crashed		
Case-1	692	0	0 (0%)	692
Case-2	68	1	5 (7%)	74
Case-3	45	40	63 (43%)	148
Case-4	336	52	19 (5%)	407
Case-5	76	211	1938 (87%)	2225

TABLE 5. Experiment results of the ArduPilot functions.

	Fuzzable		Not Fuzzable	Total
	Normal	Crashed		
Case-1	1162	0	0 (0%)	1162
Case-2	29	9	16 (30%)	54
Case-3	35	49	168 (67%)	252
Case-4	578	83	15 (2%)	676
Case-5	36	80	1262 (92%)	1378

TABLE 6. Experiment results of the Objdump functions.

	Fuzzable		Not Fuzzable	Total
	Normal	Crashed		
Case-1	133	0	0 (0%)	133
Case-2	0	0	0 (0%)	0
Case-3	51	12	31 (33%)	94
Case-4	6	0	0 (0%)	6
Case-5	4	14	234 (93%)	252

percentage shown in the table refers to the ratio of the *not fuzzable* function within the corresponding case. Lastly, *total* is the total number of functions inside the target binary.

In summary, 42.9% (PX4), 58.5% (ArduPilot), and 45.4% (Objdump) number of fuzzable functions were applicable to the vanilla fuzzer among the candidate EFFs. Successfully porting the *EFF*s to a fuzzer does not imply that such functions are worth applying fuzzing techniques and fully addressing the probabilistic coverage of finding bugs. Nevertheless, it is worth noting that FT-Framework enables automatically extracting information from unknown binary fragments to classify candidate functions that are potentially vulnerable based on forced execution.

In order to accurately evaluate the precision of our heuristic classification methods, ground truth is required. However, obtaining such ground truth data from incomplete binary fragment in a systematic way is extremely challenging; thus require additional research. In our work, although it is not systematic, we have verified the precision of our classification heuristics based on manual reverse-engineering effort with sample cases (e.g., Figure 8, Figure 9, and Figure 10).

VI. DISCUSSION

In this section, we discuss the applicability scope of FT-Framework for bug detection and a couple of limitations in our work with the suggestion for further improvement.

A. AIM AND SCOPE OF FT-Framework

This work aims to design and implement a framework that enables coverage-based fuzzing even with partial binary fragments rather than focusing on improving the fuzzing performance and precision. The point here is to facilitate *fuzzability* for broken or incomplete binaries incompatible with existing fuzzers. To achieve this goal, we attempt to scrap potential candidates for the further fuzzing process from binary fractions with static analysis only, and subsequently, we first need to discuss the *fuzzable*. Our paper divided them into *EFF*(Executable Function Fragments) and functions that are portable to fuzzers, which means *fuzzable*, and utilized a blacklist approach to identify them. By demonstrating the throughout fuzzing execution for real-world examples even without a full-system or running environment, we showed that FT-Framework empowers fuzzability by diagnosing and pre-processing target functions.

We also note that FT-Framework can be utilized to enhance the efficiency of existing coverage-guided fuzzers. For example, FT-Framework allows investigators to concentrate more on analysis and fuzzing partial path, not the entire code coverage, from the given complete binary. This feature has long been presented as an effective bug detection methodology because it reduces relatively more search space [44], [45], [46], [47]. In particular, FT-Framework can be used to prioritize various types of function cases included within a binary for bug detection. Specifically, in Case-1, which does not require parameters, users cannot trigger a bug even if it exists. Therefore, filtering-out such cases and rather focusing on the rest of the function types can increase efficiency, as discussed in [44].

A previous study [48] presented a methodology that scores to select *fuzzable* functions based on several criteria by performing static analysis, but it differs from FT-Framework in several perspectives. Likewise to our study, they analyzed the inside of a specific function using depth-search algorithms. In contrast to the goal of our paper, which attempted to determine the *fuzzable* of a particular function, they focused on identifying functions with a high probability of the occurrence of a crash depending on the complexity of the deep function. FT-Framework identified and classified low-level instructions that cause issues in independent execution by assuming that the binary may be incomplete. However, the related study does not assume such a restricted condition.

Second, the intended definition of the terminology *fuzzable* is slightly different with our proposal. The meaning of *fuzzable* in the previous study is also an ability to fuzz with a function unit, but rather, it focuses on the fact that it is more likely to cause an occurrence of crashes when fuzzing is performed. However, it does not guarantee that such functions are actually compatible with the existing fuzzers even if there may be actual crashes. On the other hand, FT-Framework supports the entire steps starting from extracting potential candidates to porting them to fuzzers as an actual independent unit, which means that it is fully compatible with the existing coverage-guided fuzzing procedure. We believe leveraging ideas of [48] would have a complementary effect on improving the limitations of FT-Framework on excavating true crashes (See §VI-B).

B. LIMITATIONS AND LESSON LEARNED

The first is the necessity for additional work for function identification in profiling Case-5. FT-Framework delivers arbitrary data by forcibly calling a function in the program rather than through the normal data-transfer path of the program when delivering the input value to the target function (*EFF*) to perform fuzzing. Therefore, Case-5 functions that refer to global variables or require upper-level logic cannot be executed normally. Thus, coverage-guided fuzzers such as AFL++, which adds input values while executing the program, produces inaccurate results. Consequently, the verification process at the previous step, such as the parent function, disappeared, resulting in errors that cannot occur through normal data delivery. An additional validation step for the data delivered to the function should be addressed to solve this problem. For example, additional information can be added on top of FT-Framework such as control-flow-graph, taint-analysis and increase the accuracy of function profiling.

The second limitation is the lack implementation for emulating system calls, thus, the coverage is slightly low. As mentioned in the previous section, functions containing system calls are excluded from *EFF*. However, some system calls can be altered/patched to be suited for fuzzing through emulation. For example, data-logging features, making a visual signal (e.g., LED device) could be considered irrelevant to data processing. System calls relevant to such features could be simply erased or ignored. Additionally, emulating a system call with fake results is another method for addressing this issue (i.e., return random value for `time` related system calls). Modifying/patching such system calls will result in achieving more *EFFs*, and finally, expand the coverage. In a previous study, external elements that are not necessarily required for testing were disabled or simply access-patterned to implement appropriate responses [49]. In system call emulation, there are limitations due to unnecessary parts, but by leveraging techniques that can be improved together, potential candidates (e.g. EFF) can be further obtained, resulting in an increase in coverage. We leave this for future work.

Another key challenge to be addressed is to detect actual bugs from the crashed *EFFs*. For example, unexpected false alarms (crashed but not vulnerable) might occur when utilizing FT-Framework, as it operates the partial fragments of binaries by design.

To address such issue, there are orthogonal researches from other literature. For example, FOCAL proposed an idea so-called *context stitching* to gradually expand the range of codes and test the feasibility of triggering specific crash. The main idea is based on the observation that relevant codes shows special sequence patterns in their invocation. Such approaches can be applied to our framework and reduce the false-alarm rate. We leave this issue as a limitation and open problem for future research.

VII. CONCLUSION

In this paper, we introduce a framework that assess *fuzzability* to functions inside incomplete binaries such as recovered firmware. Our implementation extracts *fuzzable* functions from a given binary fragment and merges with executable fuzzer stub to apply coverage-guided fuzzing based on *AFL-QEMU-ARM32*. We evaluated our system using PX4, ArduPilot firmware and GNU Binutils and successfully applied coverage-guided fuzzing. Our study shows that, at this point, additional efforts for accurately identifying a function's parameter format is required for practicality. FT-Framework introduced a first line of work towards fuzzing incomplete binary via forced execution and we hope FT-Framework can inspire other researchers to this end.

ACKNOWLEDGMENT

(Jiwon Jang and Gyeongjin Son are co-first authors.)

REFERENCES

[1] M. Zalewski. (2014). *American Fuzzy Lop*. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[2] LLVM Project. (2015). *libFuzzer—A Library for Coverage-Guided Fuzz Testing*. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[3] K. Serebryany, "Sanitize, fuzz, and harden your C++ code," USENIX Assoc., San Francisco, CA, USA, Jan. 2016.

[4] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing Javascript engines with aspect-preserving mutation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1629–1642.

[5] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in Javascript engines," in *Proc. NDSS*, 2019, pp. 1–15.

[6] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 445–458.

[7] Google. (2015). *Syzkaller—Kernel Fuzzer*. [Online]. Available: https://github.com/google/syzkaller

[8] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 167–182.

[9] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 729–743.

[10] S. Willassen, "Forensic analysis of mobile phone internal memory," in *Proc. IFIP Int. Conf. Digit. Forensics*. Cham, Switzerland: Springer, 2005, pp. 191–204.

[11] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer, "UEFI firmware fuzzing with Simics virtual platform," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[12] Z. Gao, W. Dong, R. Chang, and Y. Wang, "Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware," *Concurrency Comput., Pract. Exper.*, vol. 34, no. 16, pp. 1–15, Jul. 2022.

[13] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *Proc. 13th USENIX Workshop Offensive Technol.*, 2019, pp. 1–11.

[14] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 321–338.

[15] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 360–365.

[16] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1683–1700.

[17] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "InsTrim: Lightweight instrumentation for coverage-guided fuzzing," in *Proc. Symp. Netw. Distrib. Syst. Secur. (NDSS), Workshop Binary Anal. Res.*, 2018, p. 40.

[18] C. Zhang, W. Y. Dong, and Y. Zhu Ren, "INSTRCR: Lightweight instrumentation optimization based on coverage-guided fuzz testing," in *Proc. IEEE 2nd Int. Conf. Comput. Commun. Eng. Technol. (CCET)*, Aug. 2019, pp. 74–78.

[19] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: Fuzzing with a multi-task neural network," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 737–749.

[20] 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 *Proc. NDSS*, 2018.

[21] 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 *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1099–1114.

[22] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proc. 2nd Int. ACM Workshop Secur. Privacy Internet-Things*, 2019, pp. 15–21.

[23] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2007–2024.

[24] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.

[25] A. 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," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1201–1218.

[26] ArduPilot. (2007). *Ardupilot Main Community*. [Online]. Available: https://ardupilot.org/

[27] Drone Foundation. (2021). *Px4-Open Source Autopilot*. [Online]. Available: https://px4.io/

[28] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. NDSS*, vol. 1, 2016, p. 1.

[29] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE: Fuzzing windows applications with harness synthesis and fast cloning," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2021.

[30] C. Zhang, X. Lin, Y. Li, Y. Xue, and Y. Liu, "APICraft: Fuzz driver generation for closed-source SDK libraries," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2811–2828.

[31] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *Proc. IEEE Secure Develop. (SecDev)*, Sep. 2020, pp. 23–30.

[32] T. Ji, Z. Wang, Z. Tian, B. Fang, Q. Ruan, H. Wang, and W. Shi, "AFLPro: Direction sensitive fuzzing," *J. Inf. Secur. Appl.*, vol. 54, Oct. 2020, Art. no. 102497.

[33] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 1–13.

[34] D. Maier and L. Seidel, "JMPscare: Introspection for binary-only fuzzing," in *Proc. Workshop Binary Anal. Res.*, 2021, p. 21.

[35] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband sanitized fuzzing through emulation," in *Proc. 13th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, Jul. 2020, pp. 122–132.

[36] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "BaseSpec: Comparative analysis of baseband software and cellular specifications for L3 protocols," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[37] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Trans. Cybern.*, vol. 52, no. 5, pp. 3745–3756, May 2022.

[38] Y. Zhang, Z. Wang, W. Yu, and B. Fang, "Multi-level directed fuzzing for detecting use-after-free vulnerabilities," in *Proc. IEEE 20th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Oct. 2021, pp. 47–62.
[39] W. Wang, D. Tian, R. Ma, H. Wei, Q. Ying, X. Jia, and L. Zuo, "SHFuzz: A hybrid fuzzing method assisted by static analysis for binary programs," *China Commun.*, vol. 18, no. 8, pp. 1–16, Aug. 2021.
[40] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "StochFuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 659–676.
[41] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2271–2287.
[42] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "PMP: Cost-effective forced execution with probabilistic memory pre-planning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1121–1138.
[43] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.
[44] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, "One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Dec. 2022, pp. 388–399.
[45] Y. Kim, S. Hong, and M. Kim, "Target-driven compositional concolic testing with function summary refinement for effective bug detection," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 16–26.
[46] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2329–2344.
[47] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proc. 5th Workshop Hot Topics Syst. Dependability (HotDep)*, 2009.
[48] Alan. (Jan. 2023). *Framework for Automating Fuzzable Target Discovery With Static Analysis*. [Online]. Available: https://github.com/ex0dus-0x/fuzzable
[49] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, K. Butler, "FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2022.

JIWON JANG is a Researcher in future convergence technology engineering with Sungshin Women's University, Seoul, South Korea. Her research interests include system security, fuzzing, virtual and distributed systems, and penetration testing.

GYEONGJIN SON is a Researcher in convergence security engineering with Sungshin Women's University, Seoul, South Korea. Her research interests include system security, penetration testing, and incident response.

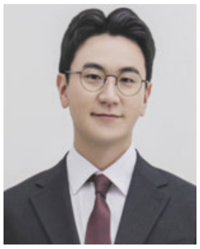
HYEONSU LEE is a Researcher in convergence security engineering with Sungshin Women's University, Seoul, South Korea. Her research interests include vulnerability analysis, system security, and fuzzing.

HEESUN YUN received the B.S. degree in convergence security engineering from Sungshin Women's University, Seoul, South Korea. Her research interests include penetration testing, autonomous vehicle security, cloud security, IoT security, and blockchain.

DEOKJIN KIM received the Ph.D. degree in information security from the Korea Advanced Institute of Science and Technology (KAIST), in 2019. He is a Principal Researcher with The Affiliated Institute of Electronics and Telecommunications Research Institute (ETRI). He has been conducting research in computer security for over ten years and has participated in projects that enhance the security of various systems and services.

SANGWOOK LEE is a Principal Researcher with The Affiliated Institute of Electronics and Telecommunications Research Institute (ETRI). He is responsible for leading the drone security research project with the institute. He has been conducting research in computer security for over ten years and has contributed to projects that improve system and service security. He has engaged in research and development planning and research of cyber security.

SEONGMIN KIM received the B.S. and M.S. degrees from KAIST, in 2012 and 2014, respectively, and the Ph.D. degree from the Graduate School of Information Security, KAIST, in 2019. He is an Assistant Professor with the Department of Convergence Security Engineering, Sungshin Women's University. His work has published in major computer science conferences, including USENIX NSDI, NDSS, and ACM WWW. More details about his research can be found at https://csesmkim.github.io.



DAEHEE JANG (Member, IEEE) received the Ph.D. degree in information security from KAIST, in 2019. He is currently an Assistant Professor with the Computer Science and Engineering Department, Kyunghee University. He was a Postdoctoral Researcher with Georgia Tech until 2020. He has participated in various global hacking competitions, including DEFCON CTF and has won several awards. In recognition of his exceptional skills, he received a special prize from the 2016 KISA Annual Event for Discovering 0-Day Security Vulnerabilities in Multiple Software Products. Additionally, he has founded pwnable.kr wargame, an educational platform aimed at enhancing hacking abilities.