JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 39, 1037-1059 (2023)
DOI: 10.6688/JISE.202309_39(5).0003

Fuzz Testing Process Visualization

HAN-LIN LU[1,+], REN-JIE ZHUANG[1] AND SHIH-KUN HUANG[1,2]
[1]*Department of Computer Science*
[2]*Information Technology Service Center*
National Yang Ming Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: littleflyer2015@gmail.com[+]; {jackgrence.cs08; skhuang}@nycu.edu.tw

The conventional fuzz testing process consists of an input mutation, an execution to test the program, monitoring, and information collection to discover bugs and security vulnerabilities. However, practical programs have more features and complex logic, and legacy mutation strategies cannot reach a deeper path to find potential bugs. A solution to this problem is to analyze the input seeds and employ test harnesses for the testing flows. This study proposes an interactive visualization tool called FuzzInspector for fuzz testing. We implemented a visualizer mode on AFL++ to generate test data for a binary analysis tool (Qiling framework and Radare2). We then visualized the controlflow graph and execution path information. This method does not require the source code and reduces the performance overhead. We also implemented an interactive user interface for the user to set the breakpoint, seed, register, and memory address and send the request to the Qiling framework for dynamic analysis. Moreover, the seed constraint can assist the fuzzer in generating a formatted seed for exploring a specific execution path. We evaluated the search time using a known approach to common vulnerabilities and exposures (CVE) and found that the search for bugs with constraints is 15 to 20 times faster than that without constraints. Moreover, we introduced a dynamic analysis feature to find controllable data and assist the exploit development process.

***Keywords*:** big data, knowledge management, knowledge creation, knowledge application, technology, Cynefin framework

1. INTRODUCTION

In recent years, software testing has become increasingly important. As software vulnerabilities continue to be revealed, the resulting losses are growing. Software developers have been conducting software testing through a fuzzing process. Fuzz testing can be used to rapidly test a program through seed mutation, execution, monitoring, and information collection, allowing bugs and security vulnerabilities to be identified and thereby improving the quality of the software. However, in practice, fuzzing still requires numerous interactions between the fuzzer and engineers to more effectively find program vulnerabilities. For example, we may need to know which part of the program the fuzzer is exploring or how many paths are executed. We also want to know whether the fuzzer is stuck under a certain branch condition and, therefore, cannot pass. In addition, the run-time information provided by the fuzzer requires an appropriate user interface to help engineers debug or understand the fuzzing behavior. Conceptually, such an interface can provide visual information and data, allowing the different stages of fuzzing to be easily understood. However, the existing fuzzers can only provide limited information and lack a suitable display interface.

Received September 30, 2022; revised November 14, 2022; accepted December 21, 2022.
Communicated by Chu-Ti Lin.
[+] Corresponding author.

A coverage-guided fuzzer called American fuzzy lop (AFL) [1] is currently one of the most commonly used fuzzers. AFL uses an instrumentation technique to insert code at compile time during the fuzzing process to retrieve the path information and rapidly change the mutation strategy. The instrumentation stores the branch target with a bitmap hash to speed up the execution time and reduce memory usage. However, the hash value cannot be transformed into an execution path.

1.1 Execution Path of the Seed Inputs

The AFL can display the number of unique paths and crash seeds, as well as the execution speed per second during the fuzzing process. Based on the path counts and timestamps, we can only know the capability of exploring the execution paths between different fuzzers. Owing to the design restrictions of a bitmap hash of the code coverage, we cannot track the actual execution path of a seed or understand which condition is satisfiable or creates a bottleneck. To resolve this issue, we can accumulate the coverage result using *llvm-cov* to realize an unsatisfiable constraint. However, *llvm-cov* does not support a binary-only option, and it needs to compile from the source code. We, therefore, use a binary analysis tool to visualize the fuzz testing, display the path information using *vis.js* [2] during fuzzing, and provide a user interface for interacting with the fuzzer.

1.2 The Benefit of Visualization

Providing visualization information during the fuzzing process will help the user understand the fuzzer behavior and modify the test harness or fuzzer to detect more bugs. Previous studies [3, 4] have shown that visualizing software behaviors can effectively assist in software development and debugging. The visualization process can help in understanding the behavior of the executing software. For example, to determine how much performance overhead is incurred through a new method, researchers in this field typically use path-growing graphs to compare the capability of exploring the execution paths and the average execution time. Furthermore, binary analysis tools, such as interactive disassembler IDA Pro [5], or Ghidra [6], support control-flow graphs that help understand the program behavior. In addition, *llvm-cov* shows the execution counts for every code block. Hence, developers can write unit tests to examine the execution results and remove redundant code.

1.3 Interactive Visualization for Fuzz Testing

To avoid affecting the efficiency of fuzzing while generating visual information, we fed the seed to other processing procedures for analysis and then transferred the analysis results to a web page for display, allowing the different paths of various seeds to be displayed. When depending solely on the pure path information, it is difficult to understand the logic of the tested program. We, therefore, added a dynamic analysis function to allow the user to select the seed to be executed, set the breakpoint, and obtain the desired register or memory content using a specific syntax.

To understand the impact of seed input generation on the state of the program execution, we also modified the colorization method of REDQUEEN [7] to obtain seed data that

cannot be modified when a branch is reached. In addition, we used the colorization method to search for seed data affecting the current state of the CPU. Thus, we could quickly determine the seed data used to control the state of the program execution, find a new path, or assist in exploit development. Finally, according to the information obtained, we developed a seed-restricted operation interface that allows users to set the constraint such that the fuzzer can generate seeds that conform to the constraint and force the fuzzer to explore new execution paths or limit the exploration to specific paths.

1.4 Contributions

The contributions of our study can be summarized as follows:

- Visualization of the execution of the input seeds. This allows users to quickly understand the differences in the program execution paths of different seeds.
- Visualization of the execution path exploration capabilities of different fuzzers. Multiple fuzzing methods can be simultaneously evaluated to determine which conditional expressions in the methods can pass or fail.
- Dynamic analysis interface. We can obtain information regarding the execution stage of a program, choose different seeds as inputs, obtain register or memory content, and understand the logic of the program.
- Mark what seed data will affect a specified CPU state, understand the conditions for exploring new execution paths, and assist in developing exploits.
- Restrict seed generation formats and direct the fuzzer to explore new or specific execution paths and rapidly locate potential vulnerabilities.
- Use FuzzInspector as a standalone tool. FuzzInspector can directly connect different fuzzer strategies and display the visualized data without modifying the fuzzer architecture.

The source code can be accessed from https://github.com/JackGrence/FuzzInspector.

2. RELATED WORK

The visualization of statistical data helps users understand their meaning. However, few studies on fuzzing have employed visualization technology. For example, VisFuzz [8] records basic blocks and functions during a compilation, but it cannot automatically analyze a bottleneck. The relationship function of FuzzInspector can assist in understanding the program logic through human operations. FuzzSplore [9] analyzed the fuzzing information over an extended period. The similarity among seeds produced by different fuzzers can be examined, and the seed mutation ability of different fuzzers can be determined. In contrast to FuzzInspector, we provide multiple fuzzers for simultaneous evaluation; in addition, we use the constraint function to share the seeds found by other fuzzers. Coverage visualizer [10] is a web-based JAVA code coverage visualization tool that provides visual code coverage information and displays which code is executed by the test program through color markers, allowing users to quickly understand the program behaviors. FuzzInspector can also draw stars in the executed code block and display the path according to the seed. The IJON [11] annotation mechanism uses human intervention to directly add comments that IJON can relate to the code and thereby guide the fuzzer. By contrast,

although FuzzIn-spector cannot provide a source-code-level guidance method, it can use the constraint function to achieve a certain level of guidance and does not require the program source code. In addition, FMViz [4] uses different color changes to represent the different mutation stages of the fuzzer. Some studies [12-14] have provided visualization reports that can help developers understand how program bugs occur and where they are located. FuzzInspector can help program developers conveniently identify program vulnerabilities, while allowing users to understand the relationship between the program input and program logic through information visualization. FuzzInspector implements a user interface, allowing the fuzzer to execute more efficiently by manually adjusting the strategy.

3. DESIGN AND IMPLEMENTATION

Initially, fuzzers cannot determine the execution paths they have not traveled along or why they cannot enter the branches. They rely solely on static and dynamic analyses to manually process large numbers of seeds. We must follow a required execution path to explore the path that a test harness has not traveled on and avoid entering branches that have been tested and are not of interest. To reduce manual intervention, we propose implementing an assistant tool called FuzzInspector, which employs an interactive visualization process for fuzz testing. It provides additional fuzzing information for viewing the differences in the execution paths of different seeds or fuzzers; moreover, it performs dynamic analysis to understand the impact of seed input on the state of a program and interacts with a fuzzer in real time to restrict it to exploring only a specific execution path.

We first introduce a fuzzer and discuss three aspects – cross-platform support, execution performance, and user interface support. AFL uses an instrumentation technique to insert code at the compile time during the fuzzing process to retrieve the path information and rapidly change the mutation strategy. Without the source code, AFL must rely on other tools to obtain running coverage information. For example, it can perform fuzz testing without the source code through QEMU mode. QEMU [15] is a generic open-source machine emulator that emulates a program on different CPU architectures. QEMU defines the minimal code unit, called a translation block, for translation. Therefore, the translation block can be considered a basic block and enables the AFL to insert the instrumentation code in the block and provide path coverage for the fuzzing process. However, QEMU does not provide a user-friendly interaction interface. Unicorn [16] provides an interface based on QEMU and implements the hook function to allow users to interact with the program (for example, when a program executes a system function, it records the relevant parameters and stops the execution if malicious commands are found.)

The changes proposed by Unicorn were eventually integrated into AFL-unicorn [17] and UnicornAFL [18], allowing AFL and AFL++ to fuzz any executable file simulated by the Unicorn Engine. Although Unicorn improves user interaction, it does not support system calls of different operating systems. Therefore, the Qiling framework [19] adds system call simulations of different architectures. In addition, QEMU's implementation of system calls to a local executable language is changed in Qiling using Python, resulting in a highly flexible user experience. In the next section, we introduce the FuzzInspector architecture, including real-time interactions with a fuzzer, the binary analysis queue, and the user interaction interface.

3.1 FuzzInspector Architecture

Our implementation is based on AFL++ [20], which integrates several fuzzing-related research methods, including power scheduling in AFLFast [21], mutation scheduling in MOpt [22], and transforming the input-to-state in REDQUEEN [7], in conjunction with MOpt. In addition, it supports LLVM, QEMU, GCC, Unicorn [16], and QBDI in the instrumentation aspect, allowing AFL++ to be applied to different target programs. Fig. 1 shows the architecture of FuzzInspector. To provide additional path information and interactions with the fuzzer, we divided FuzzInspector into several subsystems, including real-time interaction with the fuzzer, a binary analysis queue, and a user interaction interface. Using the FuzzInspector API, the system can exchange information, obtain the data required by each component, and interactively visualize the fuzz testing process.

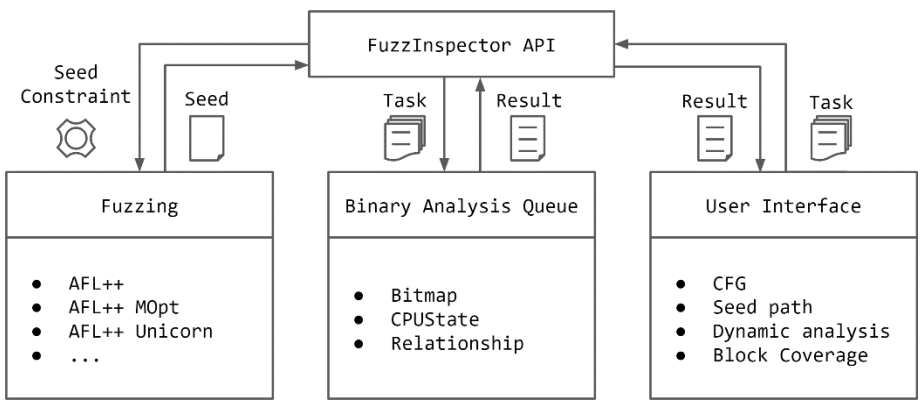


Fig. 1. FuzzInspector architecture.

• Functionality

First, real-time interactions with the fuzzer are responsible for collecting seeds, providing a follow-up analysis, and receiving user-defined seed formats. The fuzzer can only use received seeds to achieve a specific path exploration effect. The binary analysis queue provides and processes all binary-related information, including the figure generation process, conducts a dynamic analysis, returns the results of the seed path analysis, and provides statistical data. The user interface is responsible for displaying all data on the web page, such as a flow chart, log information, fuzzer path display, and related animation to distinguish the differences in the seed paths.

• Communication of Fuzzing Information and Status

The communication process can be divided into two types. The first is seed collection and analysis during the fuzzing phase. We allowed multiple fuzzers to run simultaneously. The fuzz section sends the seed to FuzzInspector and then conducts a binary analysis. The analysis results are displayed on a webpage to achieve real-time updates of the fuzzing information.

• Active Requests

The second type of communication is triggered by a user executing a required func-

tion. A function request is sent through the user interface. If it is binary, the results of a relevant information analysis are forwarded to the binary analysis queue for execution. The user interface receives the results and displays them until the execution is completed. We assume that the seed format of the fuzzer is set. In this case, the designated fuzzer on the left is notified to receive the restriction, the seed generation format of the fuzzer is controlled, and the path expected by the user is explored.

3.2 Interactions with the Fuzzer

The function of real-time interactions with the fuzzer is to collect seeds and receive the seed constraints. This allows FuzzInspector to analyze the path information provided by seeds, allowing users to understand the conditional current passing and failing formulas, compose constraints restricting the seed generation format, and control the fuzzer to explore specified paths.

- **Seed Transferring**

We added a visualization mode to AFL++. To analyze the binary information without affecting the fuzzing performance, a seed is only sent out when a new path is found. Therefore, if the fuzzer finds a new path, it will use the "Provide Seed Information" API to send the seed path and fuzzer PID to FuzzInspector, and then continue with the fuzzing process to reduce the burden on the fuzzer. To manage multiple fuzzers, we also collect the fuzzer PID for follow-up purposes.

- **Constraints**

In the constraint-handling part, we implement the signal handler of the signal, *i.e.*, user-defined signal 2 (SIGUSR2). When the fuzzer receives a signal, it uses api:Obtain Seed Constraint to obtain the seed constraint, as indicated in Table 1, and stores it in the fuzzer for further fuzzing.

Table 1. FuzzInspector API.

Function	Path	Argument 1	Argument 2	Argument 3
Obtain Seed Constraint	/fuzzer			
Unexplored List	/funccov	Binary keywords		
Refresh Screen	/bitmap/get	Basic Blocks		
Disassemble	/disassemble	Address to disassemble		
Obtain Seed Path	/path/get	Seed File path	Basic Blocks	
Obtain Seed Info	/seed	Seed File path	Fuzzer PID	
Setup Constraint	/constraint	CPU Status	Fuzzer PID	
Obtain CPUState	/cpustate	breakpoint	CPU status	Seed path
Obtain Relationship	/relationship	breakpoint	CPU status	Seed path

Table 2 lists the structure of each constraint type stored in the fuzzer. The constraint type is divided into whitelists and value ranges. Whitelist data are provided by the user for seed data replacement. Moreover, users can make detailed adjustments to the seeds through the API, such as the starting position and length of the data in the seed to be replaced.

Table 2. Constraint structures.

Offset	Name	Description
0×00	constraint_type	Numeric type or Whitelist
0×04	endian	Little or Big endian
0×08	offset	Seed Offset
0×0c	overwrite_len	Overwritten Data Length
0×10	data_cnt	Constraint Size
0×14	data	Constraint 1 size and type
0×??	data	Constraint 2, 3, ..., data_cnt

A constraint is used after a fuzzer generates a new seed and the target program executes it. The pseudocode converting the seed is shown in Algorithm 1. It reads all constraint expressions on the execution path. Modifiable seed data are converted into numerical index values of the candidate data. However, to avoid data that already conform to a constraint formula from becoming index values, we add an external constraint formula to ensure that the seed is not converted if the data on the offset satisfy the constraint requirements. This will enable the fuzzer to explore a specific path through seed format conversion.

Algorithm 1: Seed Format Transform

Require: *seed*: The Seed generated by the fuzzer; *constraints*: The user-defined constraints;
Ensure: The *seed* follows the constraint;
1: For each *constraint* $\in$ *constraints*
2: Read *candidates* from *constraint*;
3: Read *off set* from *constraint*;
4: Read *randint* from *seed*[*off set*];
5: Calculate *index* = *randint* mod Size of *candidates*;
6: Read *value* from *seed*[*off set*]
7: If *value* $\notin$ *candidates*
8: *value* = *candidates*[*index*]
9: Replace data at *seed*[*off set*] to *value*;
10: return *seed*;

3.3 Binary Analysis Queue

Fuzzinspector is based on AFL++, which supports Unicorn mode and allows programs simulated by the Unicorn Engine to perform fuzz testing. However, because Unicorn does not provide the functions of the system calls, it is necessary to use the Unicorn hooks to implement the system calls required by a target program. The pre-operation process in fuzz testing is time-consuming; therefore, we use the Qiling framework as the binary analysis tool. The Qiling framework implements the system calls of other major systems and does not need to be written by the user. Because it is developed based on the Unicorn Engine, it can be connected to UnicornAFL and uses AFL++ Unicorn mode for fuzzing.

The binary analysis queue stores all execution data on the binary-related functions, which the fuzzer then uses to obtain dynamic and static analysis information through the Qiling framework and r2pipe API. We define different execution logic roles according to

the required function and store the work to be executed in a queue, allowing r2pipe to execute in sequence and maintain consistency in the binary analyses. Radare2 and r2pipe provide a binary analysis library [23]. FuzzInspector uses the r2pipe API for a binary analysis and provides the data required for visualization. It supports various operating systems, CPU architectures, and file formats and provides r2pipe, allowing language bindings to use the Radare2 functions.

• **Seed Analysis**

First, the bitmap function analyzes the received seed and displays data on the interactive user interface. To obtain the path information in the execution stage, FuzzInspector creates a sub-process program to execute a pre-written Qiling framework script, which can provide parameters to change the execution behavior. The bitmap function uses parameters to create a sub-process of the Qiling framework with a basic hook block ability and prints the current address and path information in each basic block without the program source code. In this way, the fuzzer can count the unexplored paths, allowing users to understand the path differences and which functions currently have low program coverage; this will help identify the functions that are difficult for the fuzzer to explore.

In addition, the function address of r2pipe is defined as the execution path because the released program typically uses compiler optimization options to make the program smaller and faster, which hinders the ability of the static analysis tool to locate the function accurately. Therefore, if we find an address that r2pipe cannot resolve while obtaining the running path, we define it as the beginning of the function to provide a DOT graphic description file to help the user develop a flowchart in the interactive interface.

• **Dynamic Analysis Information**

The CPUState function is used to dynamically analyze the target program and receive a breakpoint address, the CPU state information, and the seed path to be used. With these three parameters, we can create a sub-process sequence to set the parameters, and the Qiling framework script runs in CPUState mode. The script stops at the breakpoint, analyzes the CPU status information, and prints the result to provide the execution stage information to the user.

The CPU status information format must comply with the format defined by (*type_length_address*). The supported types are numeric, string, and hexadecimal values. The stack, default, and register names are for a particular usage, and the formats do not need to provide the length or address fields. In addition, the address field can either be given a value or the register's name. The register value will be automatically read as the address during parsing, and the data at the address will be obtained.

• **Obtaining Seed Information**

The relationship function determines the relationship between the seed data and the CPU status at the execution stage. The user can quickly understand which of the seed data are necessary for a branch and which data will affect the specified register or memory content. The relationship function can help with the exploit development to control the CPU state. To achieve this, we implemented the relationship mode in the Qiling framework script. The receiving parameters are the same as the CPUState function parameters – the breakpoint address, CPU state information, and seed path. In this mode, the modified colorization method is performed twice.

Table 3. CPU status types.

Type	Description
u16	16 bit value
u32	32 bit value
u64	64 bit value
str	Null terminated string
byte	hexadecimal format and printable ASCII
hex	hexadecimal format
map	filename and offset
stack	first 10 data in stack
default	all registers and stack information
register	register values

The original colorization algorithm continuously replaces the seed data with random characters and determines whether the bitmap is tainted. If the bitmap is not tainted, the algorithm restores the data, cuts the seed range into two parts, and runs the colorization algorithm again. Finding the relationship between a bitmap and seed enables the fuzzer to be more efficient because the fuzzer only needs to replace a specific part of the seed. We believe that such an algorithm can be used to determine the seed data necessary to reach the target address and determine the seed data that can be affected by the specified CPU status information, both of which can be satisfied by modifying the bitmap-check process of the colorization algorithm. After the dynamic library is loaded, we create a sub-process to run the CPUState mode, collect the return data of the sub-process, select the scope of the colorization method to be used, and identify the data that affect the state of the execution.

• **Forcing a Fuzzer on a Pre-specified Path**

The constraint function can restrict the seed-generation format to explore a specific path. The constraint information in the queue parses the constraint format returned by the user and then converts it into API data for the fuzzer to read.

The constraint function can restrict the seed generation format to explore a specific path. The constraint information in the queue parses the constraint format returned by the user and then converts it into API data for the fuzzer to read. Multiple constraints can be provided simultaneously and separated by custom symbols. As shown in Fig. 2, the | symbol is used to separate different constraints. Custom symbols can separate relevant constraint data. The constraints include type, endianness, offset, copy length, and data. There are four field formats, namely str, hex, int, and range. Except for the range format, all other types are written in a whitelist file, which is stored in list form. Each time the seed is converted, one of the whitelist data points is selected as the seed data for that time. By contrast, the range type accepts only two pieces of data – the beginning and end of the value range. The seed data is obtained by selecting a numerical range.

The endian order (byte order) is primarily used for the int and range types. The "<" and ">" symbols indicate whether a little endian or big endian is used to fill in the value when modifying the seed. The offset field specifies the offset addresses in the seed to fill in the data. Considering Fig. 2 as an example, which indicates that two bytes are written at

address offset 10 of the seed, the data are in big-endian order, and a value of between 0×1 and 0×5000 is selected according to the initial data of the seed number.

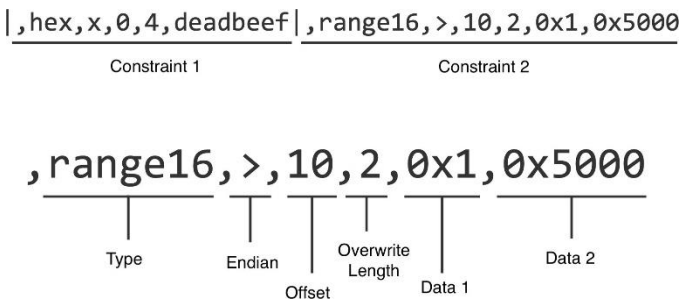


Fig. 2. Constraint example.

After successfully parsing the constraint data provided by the user, the constraint API will be updated, and the Fuzzinspector will obtain the data content of the seed constraint API and send a SIGUSR2 signal to the fuzzer. The fuzzer invokes the API upon receiving the signal. It receives and parses the constraints provided by the user, enables the user to interact with the fuzzer in real-time, and restricts the seed format to reach the function of a specific path for exploration.

Table 4. Constraint type.

Type	Description
range8, range16, range32, range64	8, 16, 32, 64 bit range type
int8, int16, int32, int64	8, 16, 32, 64 bit value type
hex	hexadecimal type
str	string type with null termination

3.4 User Interface

The user interface provides information for users to track the current running status of the fuzzer, including the different paths of seeds, the CPUState information, the relationship search process, and the current running log information. To achieve this, we used *vis.js* to obtain the program flowchart and API provided by the flask. *Vis.js* [2] is a browser-based visualization library that can handle large amounts of dynamic data for drawings, including network diagrams, timelines, line graphs, and bar graphs. Charts can interact with data to achieve drag-and-drop operations. In addition, many events and methods are provided to customize the required visual content. Users can view the specified address of the function flow chart or other advanced information and functions provided by FuzzInspector.

• Path Information

In the execution path interface, the “real-time screen update” API is called once per second to update the current status of the fuzzer in real-time. The API returns the address displayed on the current process by which the fuzzer is executed with stars of different colors. Using this information, we can identify the differences in the ability of the fuzzer

to explore new paths. In addition, we use the "get seed path" API to obtain the execution path of the specified seed and add flickering and jitter effects on the stars belonging to the path, allowing users to understand the path differences between different seeds.

• **Dynamic Analysis and Constraint Interface**

For the CPUState, relationship, and constraint functions, we must provide the input boxes required for each function, such as the disassembled command, available seed path, and fuzzer list. FuzzInspector places the required information into the binary analysis queue and waits for the execution. After the user presses the send button, the "get cpustate," "get relationship," and "set constraint" APIs are called. We also included loading animation and real-time log information for users to know which feature is currently running during the waiting process. In addition, we introduced a data search animation in the relationship function to show the details of each execution step. Finally, we marked the unchangeable and variable data with different colors to enable users to understand the relationship between seed data and the execution stage.

4. RESULTS AND EVALUATION

A 3.2-GHz Intel i7-8700 processor and 16 GB of memory were used for the experiment.

4.1 FuzzInspector User Interface

In the first experiment, we used a WF2419 router as the target and ran the AFL++ MOpt and default modes. This section introduces various user interfaces and functions in FuzzInspector.

• **Seed Path Difference**

The interface of FuzzInspector is shown in Fig. 3. We can select the seed and address we want to observe and create the program flowchart in the middle. To observe and understand the differences in the seed paths, we can select the seed according to the top menu shown in Figs. 4 and 5 and add brackets to the stars according to the selected seed to indicate the seed path.

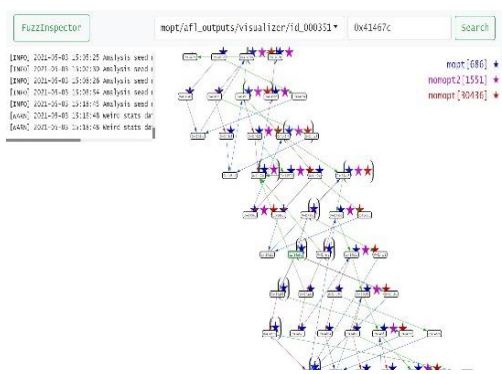


Fig. 3. FuzzInspector main interface.

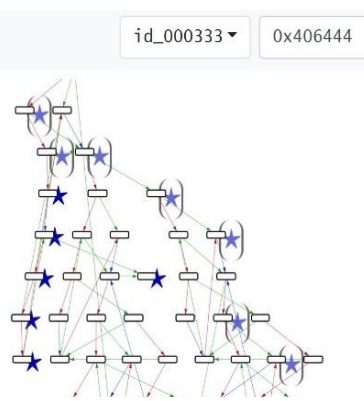


Fig. 4. The path difference flowchat.

• Fuzzer Path Exploration

FuzzInspector can be simultaneously connected to multiple fuzzers. It can show which fuzzer has a greater path exploration ability among the fuzzers. As shown in Fig. 19, different-colored stars represent different fuzzer execution results. The presence of the blue star on a code block indicates that the fuzzer has a better exploration ability.

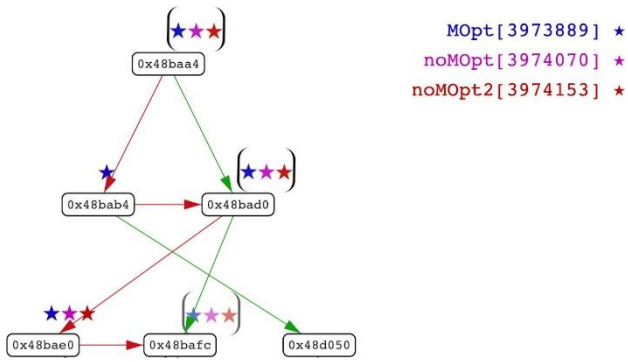


Fig. 5. Functions with large path differences.

• Dynamic Analysis Interface

The breakpoint address, seed, register, or memory can be specified in the dynamic analysis interface. It is executed by Qiling to obtain the required data and display it in the user window. Fig. 6 shows the interface for selecting the breakpoint. We can view the disassembly result of the address and select it for observation. We also provide a seed selection interface, as shown in Fig. 7, which shows all seeds running on the basic block that users can choose from.

Fig. 6. The break point selection interface.

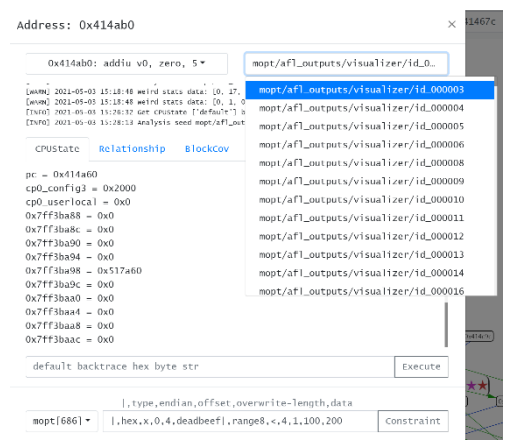


Fig. 7. The seed selection interface.

By observing the contents of the register or memory, as shown in Fig. 8, we can enter register a1 to be observed in the input box and then use u32 to observe the memory and display it as a 32-bit value. Fig. 9 shows the string and byte types used to observe the

specified memory address. We also provide the memory address field for the register's name, and the current value of the register is used as the memory address for reading dynamic data.

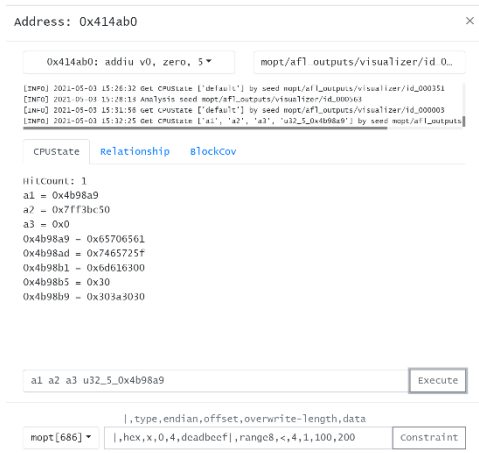


Fig. 8. The address selection interface.

Fig. 9. The value in the memory address.

- **Seed and CPU Status**

The relationship function can obtain the relationship between the seed and specified CPU state. It allows us to determine which seed data are necessary to reach the branch and which seed data affect the CPU state. Thus, we can determine which input should be used to find new paths and assist with the exploit development. Fig. 10 shows an example of a program flowchart. Next, we use the relationship function to understand the arrival conditions of the 0×414adc address and the controllable data.

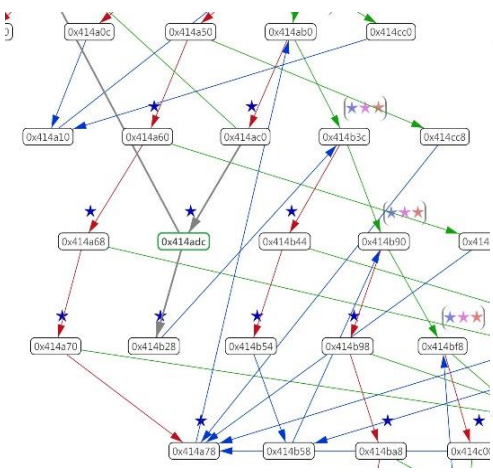


Fig. 10. The flowchat of program.

Fig. 11. The exploit content.

First, the 0×414adc address is clicked to enter the analysis interface, as shown in Fig. 11, and then the CPUState function is used to obtain the memory content, which is the data we want to control. Next, the switch to the relationship function interface to be searched, as shown in Fig. 12, is an animation of the execution process of the relationship, which allows users to understand what bytes are found at this stage and are marked with different colors. The final result is shown in Fig. 13, where the pink block represents the seed data that must exist to reach the address, and the blue block represents the data that can affect the memory content, allowing users to quickly understand what control information is used to assist in the exploit development.

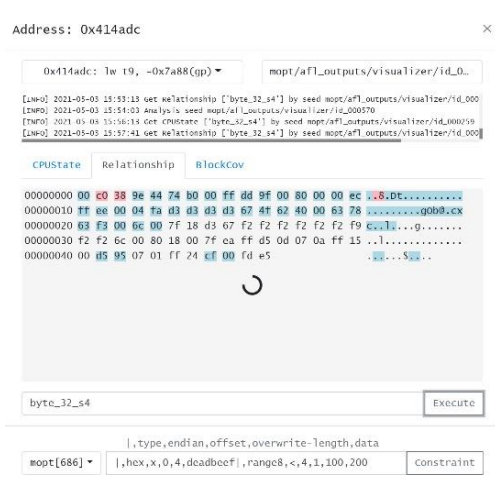


Fig. 12. The relationship loading flow.

Fig. 13. The example of showing the controllable seed.

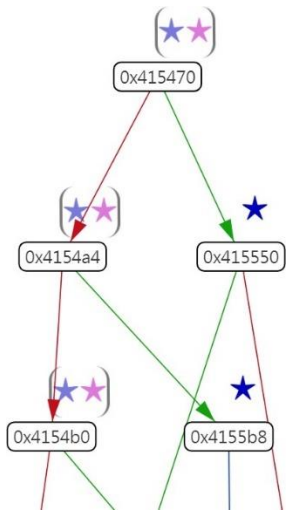


Fig. 14. Before constraint setting.

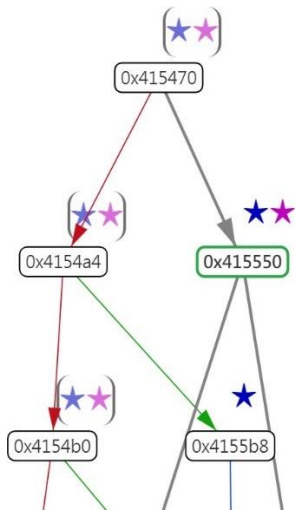


Fig. 15. After constraint setting.

• **Constraint Interface**

The constraint function receives the constraint formula provided by the user and allows the fuzzer to convert the mutated seed according to the constraint formula before running the target program to obtain a seed that conforms to the constraint formula. It then uses the seed to run the target program. Using this method, users can set up whitelist data, explore different paths, or restrict the fuzzer to explore specific paths.

As shown in Fig. 14, only the fuzzer of the blue star can move to the right branch in this function. To let the pink star also go to this path, we first use the relationship function to analyze how to reach this branch condition. Fig. 16 shows the conditional search process. After the search is completed, as shown in Fig. 17, we set the constraint for the fuzzer of the pink star according to the results. After a period of time, we obtain the result shown in Fig. 16. As shown, a conditional setting allows the fuzzer to go to the branch that is not considered initially. This function explores a specific path according to the user's requirement.

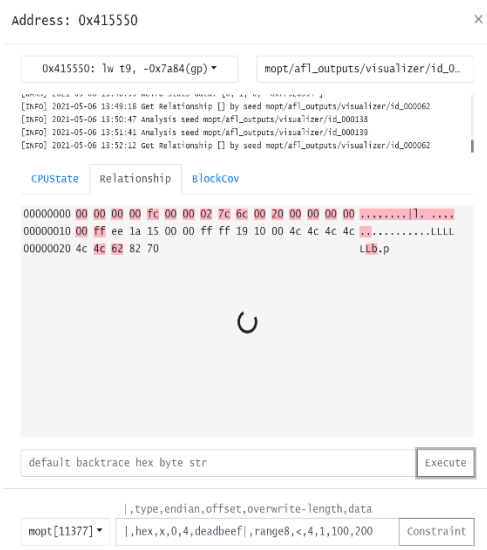


Fig. 16. The condition searching flow.

Fig. 17. The condition search result.

• **Block Coverage**

While analyzing the seed, the block coverage is also calculated, allowing users to discover the function with low coverage. In addition, we also calculate the degree of difference between the execution paths. The information on the degree of difference can help the developer find bottlenecks encountered by the fuzzer and use this information as a basis for improvement.

Fig. 20 shows the block coverage screen, where the user can input the keywords of the executable file or function library to filter the function. The function of filtering out the CGI program is shown in Fig. 20. The interface is divided into left and right columns. The left side depicts the function with the least and highest coverage. If the first link is clicked, we will be directed to the page shown in Fig. 18. As this function runs only two basic

blocks, the coverage of this function is low. On the interface shown on the right, the function of the fuzzer from a large path difference to a small path difference is represented from top to bottom. The user clicks on the first link to enter the page shown in Fig. 19. Even when the basic block of this function is executed, not all fuzzers will be executed, allowing users to find the fuzzer bottleneck quickly.

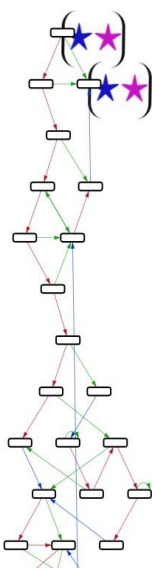


Fig. 18. Functions with low coverage.

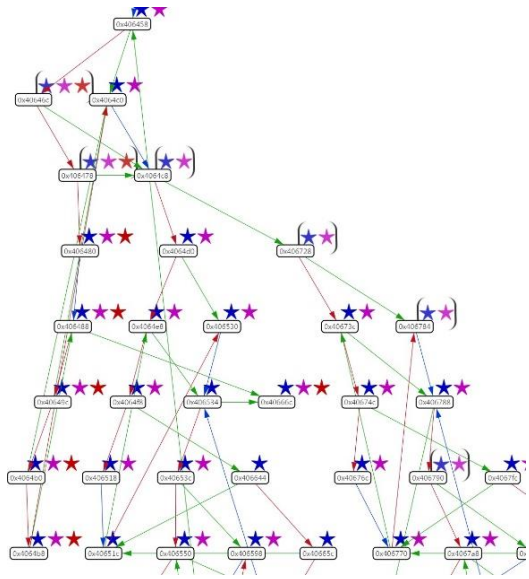


Fig. 19. The comparison of fuzzer coverage age capabilities.

Fig. 20. Block coverage interface.

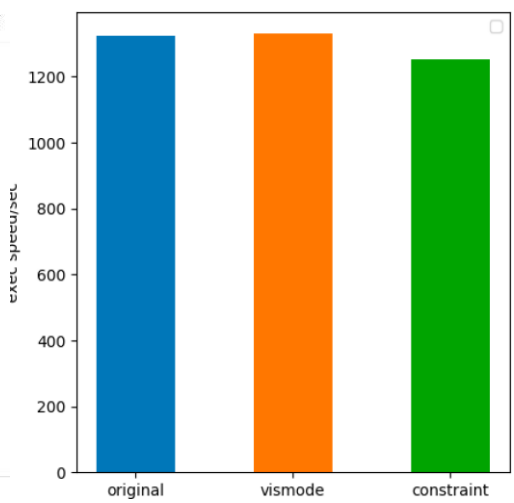


Fig. 21. The efficiency comparison of different fuzzing strategies.

• **Log Information**

The log message records all related functions and actions and then provides users with an understanding of the current running status of FuzzInspector, such as the message class, time, and detailed information. The log display box is displayed in the upper-left corner of the main screen, as shown in Fig. 3. In addition, a log message display is also provided in the function interface, allowing the user to check which task FuzzInspector is currently performing.

• **Alert on Fuzzer Bottleneck**

To notify users when a bottleneck is encountered, we developed a simple algorithm that averages the number of paths at regular intervals to determine whether the fuzzer has encountered a bottleneck. As shown in Fig. 22, in addition to the fuzzer name and color displayed in the upper-right corner, when we determine that the fuzzer has encountered a bottleneck, an exclamation point is added to the front of the name, prompting the user to check the performance analysis of the fuzzer and modify the constraint to find a new path.

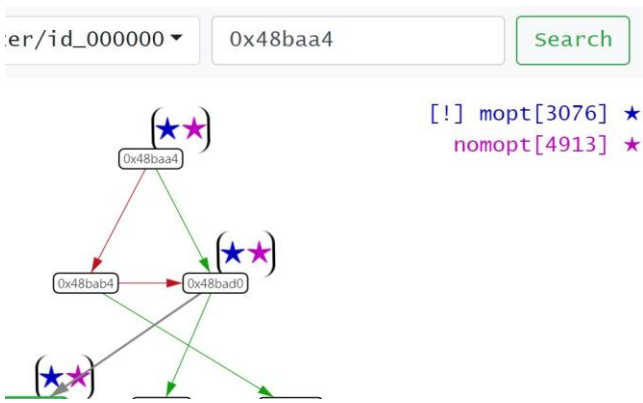


Fig. 22. The bottleneck tips.

• **Semi-Auto Filling of Constraint**

To allow the execution results of the relationship to be readily used in the constraint function, we provide a semi-automatic method for users to select the constraints to be applied. The data marked by the relationship function can be converted into a constraint format by clicking and dragging it into the input box to facilitate user operations.

• **Constraint Efficiency**

The execution speed is analyzed, and the experimental results are shown in Fig. 21. Here, “original” stands for the original AFL++, “vismode” indicates the enabling of visualization mode, and “constraint” represents the use of the constraint function. It can be observed that FuzzInspector incurs almost no performance penalty for fuzzing.

4.2 Known CVE

In this experiment, we use CVE-2021-3156 as the experimental target. Vulnerability

is a heap-based buffer overflow, which allows an attacker to use the vulnerability to escalate the privileges on the local terminal and gain root privileges. We used FuzzInspector with AFL++ to test whether the constraint setting function can find vulnerabilities more quickly, applied FuzzInspector to analyze the causes of the crash, and utilized the conditions to assist in the exploit development.

• **CVE Searching Time**

We used the AFL++ MOpt and default modes to compare the time cost under different constraints, including no constraints, parameters known to trigger vulnerabilities, and a list of known parameters. The experimental results are listed in Table 5. As shown, MOpt mode is nearly three times faster than the default mode when the constraint function is not used; however, the speed is approximately the same as the constraint; however, it is 30-times faster than that of MOpt and 130times faster than that of the default mode. In Table 5, constraint* represents a list of known parameters. The fuzzer with constraint* is approximately 15 to 20 times faster than that with no constraint, and the default mode with constraint* is even faster. The MOpt mode is approximately five times faster. Based on a certain degree of understanding of the target program, program vulnerabilities can be quickly identified by customizing data in a specific format for fuzz testing.

Table 5. CVE-2021-3156 searching time (in second).

Mode	No-Constraint	Constraint	Constraint*	Speedup	Speedup*
MOpt	843.33	22.83	39	36.94	21.62
Default	2428.17	18.50	161.67	131.25	15.02

• **Crash Analysis**

Next, we analyzed the crash and set the breakpoint in the parameter analysis loop, as shown in Fig. 24. The red box contains the parameter content that is typically read; however, if we look carefully, we will find that there is a duplication of data, and the memory address to be read is a chunk in the heap section. If the seed is used directly to execute the program, a heap-related error will be displayed. We, therefore, know that this crash belongs to a heap-based buffer overflow.

• **Controllable Data Analysis**

We can use the relationship function to confirm whether the overflow address is controllable; therefore, we set the relationship parameter to the memory section that causes the overflow. Fig. 23 shows the part that reaches this position (shown in pink). The necessary data for the address can be seen as parameter *i*. The data marked in blue represent the controllable range. We can determine the controllable data-position using such a display interface, which helps construct appropriate chunk data to complete the heap exploitation.

4.3 User Experience

In this experiment, we determine whether FuzzInspector can effectively assist test engineers in understanding the bottleneck of the fuzzer and improve the code coverage without understanding the target program. The target program was libexif-0.6.22, and AFL++ enabled the MOpt and CmpLog modes. The test engineers were two laboratory

students with experience in reverse engineering and fuzzing but unfamiliar with the target program libexif.

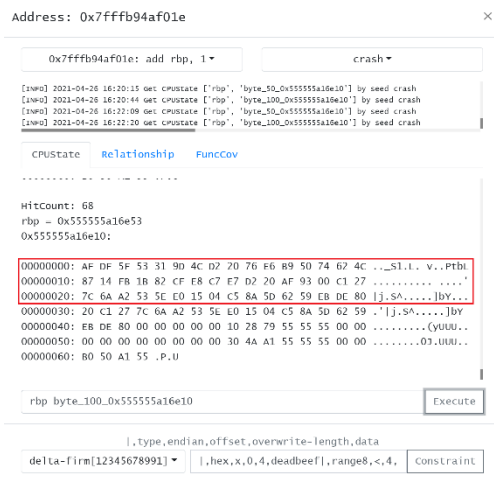


Fig. 23. Crash behavior.

Fig. 24. Crash exploit analysis.

Both test engineers intermittently used FuzzInspector. As shown in Fig. 29, the two test engineers increased the number of paths by 30% and 10%. They first executed the fuzzer for about 10 and 15 days, respectively. From Fig. 29, we can see that in the first 10 hours, the fuzzer was able to find new execution paths quickly. However, the fuzzer ran for about 24 to 50 hours, and the coverage slowed down rapidly. After 100 hours of execution, the fuzzer can find about 10 new paths every day, which shows that the fuzzer has tried most of the program branches that are easy to pass, but it is stuck in a difficult condition and cannot execute new paths. When the coverage can no longer increase, the engineer uses the FuzzInspector to find out the input that can pass the specific branch condition and then let the fuzzer continue to execute according to the new input.

From the positions of 260 and 370 hours on the horizontal axis of Fig. 29, the coverage drastically increases, showing that the fuzzer passed the stuck branch, thus discovering new program execution paths. For example, the function in Fig. 30 uses different strings for comparison. The test engineer uses FuzzInspector to find the conditional expression that has not passed “FUJIFILM”. After the conditional formula was set to the fuzzer, over 200 new paths were discovered.

We also noticed that libexif has several conditional expressions that the fuzzer cannot pass. Therefore, the curve in the figure is stepped, which means that every time an engineer uses FuzzInspector, the program coverage is significantly improved.

4.4 Comparing with Other Well-Known Tools

In this experiment, we also use libexif-0.6.22 as the benchmark program. The engineer uses FuzzInspector and other well-known software testing tools to help fuzz perform fuzz testing more effectively. The test engineer is currently an information security engineer.

We first define three steps to improve the fuzzer:

- Find the bottleneck of the fuzzer.
- Find the location of the bottleneck branch.
- Identify the relationship between inputs and branches.

In the first step, since afl++ cannot provide information on which branch conditions the fuzzer cannot pass. Therefore, the engineer uses GDB and r2 to obtain this information. As shown in Fig. 25, we first execute afl with gdb, then set a breakpoint at *_afl_maybe_log*, and obtain the execution path of the fuzzer each time. Then use the afb command of Radare2 to obtain the basic block address, as shown in Fig. 26. Finally, all the paths that the fuzzer has traveled are processed by the above method, and the memory address of the branch that the fuzzer cannot pass can be obtained. In the second step, when the branch memory location is found, the engineer must analyze the code through reverse engineering before they can understand the logic of the branch.

Fig. 27 shows that the engineer performed reverse engineering through the well-known software IDAPro [5], decompiled the source code, and found that the fuzzer could not generate a string like 0x4D4C4946494A5546LL, which is "FUJIFILM".

In the final step, the engineer must analyze the relationship between the fuzz seed and the string "FUJIFILM". As shown in Fig. 28, the seed is related to the register rdx. The engineer must gradually find the final relationship between the seed byte and the branch condition through the debugger and static analysis tools. The engineer might takes several hours or days to complete the above steps, depending on the complexity of the test program. However, during the execution of the fuzzer, there will be many bottlenecks. If the engineer uses traditional methods to deal with it, it will take a lot of time. Compared with our tool, FuzzInspector obtains the bottleneck location address through Qiling and directly finds the relationship between the input and memory through the relation function. It also provides a user-friendly interface for the engineer to create constraints directly, and FuzzInspector will automatically generate seeds that can pass the branch.

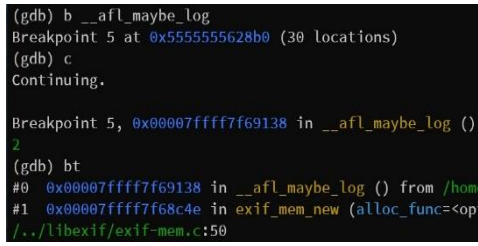


Fig. 25. Using breakpoint on GDB.

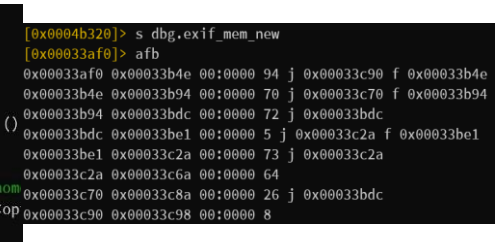


Fig. 26. Using afh command on Radare2.

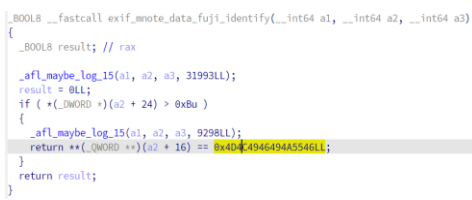


Fig. 27. The reverse code in IDAPro.

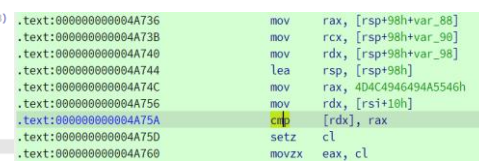


Fig. 28. The relationship between input and program register.

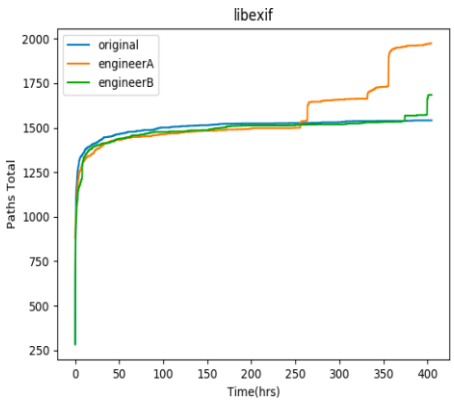


Fig. 29. Manually directed effects.

```
static void
interpret_maker_note(ExifData *data, const unsigned char *d, unsigned int ds)
{
...
    if ((mnoteid = exif_mnote_data_olympus_identify (data, e)) != 0) {
        ...
    } else if ((mnoteid = exif_mnote_data_canon_identify (data, e)) != 0) {
        ...
    } else if ((mnoteid = exif_mnote_data_fuji_identify (data, e)) != 0) {
        ...
    } else if ((mnoteid = exif_mnote_data_pentax_identify (data, e)) != 0) {
        ...
    }
...
}
...
int
exif_mnote_data_fuji_identify (const ExifData *ed, const ExifEntry *e)
{
        (void) ed;  /* unused */
        return ((e->size >= 12) && !memcmp (e->data, "FUJIFILM", 8));
}
```

Fig. 30. Difficult conditions.

5. CONCLUSION

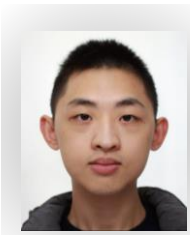
In this study, we developed FuzzInspector, which can analyze the behavior of a fuzzer without the source code and provides visual and interactive interface functions. FuzzInspector provides engineers with sufficient information to help them readily identify programming errors. The experimental results indicate that the visual interface enables engineers to clearly identify which branches the fuzzer cannot pass through and compare the path exploration capabilities of multiple fuzzers. Finally, the developed interactive interface can help engineers understand the program logic and mark controllable data to assist in the exploit development. The FuzzInspector enables the users to understand and guide the fuzzer using a visual interface. With this tool, the fuzzer is no longer just an individual running instance but can interact and communicate with humans.

REFERENCES

1. M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014.
2. B. Almende, "vis. js-a dynamic, browser based visualization library," http://visjs. org/. Acesso em, Vol. 1, 2016.
3. S. Diehl, "Past, present, and future of and in software visualization," in *Proceedings of International Joint Conference on Computer Vision, Imaging and Computer Graphics – Theory and Applications*, 2015, pp. 3-11.
4. A. Hussain and M. A. Alipour, "FMViz: Visualizing tests generated by AFL at the byte-level," *arXiv Preprint*, 2021, arXiv:2112.13207.
5. I. Guilfanov, "The best-of-breed binary code analysis tool," https://www.hex-rays.com /ida-pro/.
6. R. Rohleder, "Hands-on ghidra-a tutorial about the software reverse engineering framework," in *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, pp. 77-78.

7. C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proceedings of Network and Distributed Systems Security Symposium*, Vol. 19, 2019, pp. 1-15.
8. C. Zhou, M. Wang, J. Liang, Z. Liu, C. Sun, and Y. Jiang, "Visfuzz: understanding and intervening fuzzing with interactive visualization," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1078-1081.
9. A. Fioraldi and L. P. Pileggi, "Fuzzsplore: Visualizing feedback-driven fuzzing techniques," *arXiv Preprint*, 2021, arXiv:2102.02527.
10. M. C. Saputra and T. Katayama, "Code coverage visualization on web-based testing tool for java programs," *Journal of Robotics, Networking and Artificial Life*, Vol. 2, 2015, pp. 89-93.
11. C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proceedings of IEEE Symposium on Security and Privacy*, 2020, pp. 1597-1612.
12. P. Opmanis, R. Kikusts, and M. Opmanis, "Visualization of large-scale application testing results," *Journal of Modern Computing*, Vol. 4, 2016, p. 43.
13. H. Wang, X. Zhang and M. Zhou, "MaVis: Feature-based defects visualization in software testing," in *Proceedings of Spring Congress on Engineering and Technology*, 2012, pp. 1-4,
14. Z. Zuo, "Research on visual analysis method of software function testing process," in *Proceedings of International Conference on Computer Graphics, Artificial Intelligence, and Data Processing*, Vol. 12168. 2022, pp. 150-158.
15. F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41, 2005, p. 46.
16. N. A. Quynh and D. H. Vu, "Unicorn: Next generation CPU emulator framework," *BlackHat USA*, Vol. 476, 2015, pp. 63-72.
17. N. Voss, "afl-unicorn: Fuzzing arbitrary binary code," https://github.com/Battelle/afl-unicorn, 2017.
18. A. Fioraldi and D. Maier "unicornafl: Unicorn engine for aflplusplus," https://github.com/AFLplusplus/unicornafl, 2020.
19. L. Kaijern, "Qiling framework: Advanced binary emulation framework," https://github.com/qilingframework/qiling, 2019.
20. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies*, 2020, p. 10.
21. M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032-1043.
22. C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1949-1966.
23. Pancake, "Unix-like reverse engineering framework and command-line toolset," http://github.com/radareorg/radare2.

Han-Lin Lu received the BS degree in the Department of Transportation Technology and Management, and MS degree in Computer Science and Engineering from National Chiao Tung University, Taiwan in 2010, and 2012 respectively. He is currently pursuing the Ph.D. degree at the Institute of Science in Computer Science and Engineering of National Chiao Tung University. His research interests include software quality, software security, and fault localization.



Ren-Jie Zhuang is a senior vulnerability researcher in TeamT5. He received his BS degree in Computer Science from National Kaohsiung University of Science and Technology in 2019, MS degree in Cyber Security from National Yang Ming Chiao Tung University in 2021. His research interests include software quality, software security, and network security. Recently, his research focuses on IoT security and mobile security.

Shih-Kun Huang received his BS (1989), MS (1991), and Ph.D. (1996) in Computer Science and Information Engineering from the National Chiao Tung University. Currently, he is the Deputy Director of the Information Technology Service Center, and jointly with the Department of Computer Science, National Chiao Tung University. Dr. Huang's research integrates software engineering and programming languages to study cyber security and software attacks.