

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

—*—

THESIS

SUBMITTED FOR PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

ENGINEER

IN

INFORMATION TECHNOLOGY

HERMES - A PROTOCOL FUZZER BASED ON AMERICAN FUZZY LOP

Author : **Do Minh Tuan**
Class ICT-59
Supervisor : **Dr. Tran Quang Duc**
Advisor : **Dr. Nguyen Anh Quynh**

HANOI, 12 – 2019

REQUIREMENTS FOR THE THESIS

1. Student information

Student name: Do Minh Tuan

Tel: 076 820 7969

Email: tuanit96@gmail.com

Class: ICT-59

Program: ICT

This thesis is performed at: School of Information and Communication Technology–
Hanoi University of Science and Technology

From: August, 2019 to Dec, 2019

2. Goal of the thesis: To develop a Fuzzing System which is used to find memory corruptions of network protocol's implementation in IoT devices.

3. Main tasks:

- Research about the mechanism of Linux Operating System interprocess communication and networking implementation
- Design a fuzzing system in order to find vulnerability in IoT devices
- Implement the proposed system
- Conduct the experiments, synthesize, analyze and compare the results

4. Declaration of student:

I - *Do Minh Tuan* - hereby warrants that the Work and Presentation in this thesis are performed by myself under the supervision of *Dr. Tran Quang Duc* and *Dr. Nguyen Anh Quynh*.

All results presented in this thesis are truthful and are not copied from any other work.

Hanoi, 27 Dec, 2019

Author

Do Minh Tuan

5. Attestation of the supervisor on the fulfillment of the requirements of the thesis

Hanoi, 27 Dec, 2019

Supervisor

Dr. Tran Quang Duc

TÓM TẮT NỘI DUNG ĐỒ ÁN TỐT NGHIỆP

Fuzzing, hay còn gọi là kiểm thử ngẫu nhiên, là một kỹ thuật kiểm thử tự động cho các phần mềm nhằm tìm ra các lỗ hổng bảo mật bằng cách sinh ra một chuỗi data ngẫu nhiên, coi chúng như một đầu vào và truyền cho chương trình. Sau đó nếu chương trình bị kết thúc đột ngột do lỗi bộ nhớ thì ta sẽ có được thông báo cụ thể. Những lỗi này có thể dẫn tới các tổn hại nghiêm trọng như thực thi code từ xa, nâng quyền của user hiện tại trên hệ thống hay tấn công từ chối dịch vụ. Hiện tại, American Fuzzy Lop là một trong những công cụ kiểm thử tốt nhất trên thế giới và nổi tiếng về các thuật toán làm tăng độ phủ của các trường hợp thử.

Tuy nhiên, đối với kiểm thử các giao thức mạng, American Fuzzy Lop không được hỗ trợ đầy đủ: thứ nhất, công cụ này chỉ làm việc với các chương trình đọc đầu vào từ file hoặc stdin, nên nó không có các gói hỗ trợ về mạng. Thứ hai, một máy chủ mở dịch vụ chạy một giao thức thường sẽ có một vòng lặp vô hạn, nhằm phục vụ cho nhiều người dùng, vậy nên American Fuzzy Lop không thể có được kết quả của việc thực thi một file đầu vào của chương trình.

Nghiên cứu này sẽ đưa ra một số giải pháp nhằm khắc phục những điểm yếu kể trên của American Fuzzy Lop và không những vậy còn đạt được những kết quả vượt trội so với các công cụ về kiểm thử giao thức mạng hiện hành. Bằng việc sử dụng công cụ fuzzing này, chúng tôi đã thành công trong việc khẳng định các lỗi đã tồn tại ở một số phần mềm và hơn thế nữa, trong vòng vài tháng tiến hành thử nghiệm, chúng tôi còn tìm ra được nhiều bug mới, một vài lỗi trong số đó được gán số CVE.

ABSTRACT OF THESIS

Fuzzing or fuzz testing is an automated software testing technique that generates a random data as inputs and gives to a computer program, then expects for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. These vulnerabilities can lead to serious damages: remote code executions, privilege escalation or denial of service. Currently, American Fuzzy Lop is one of the best fuzzing tool in the world. It employs genetic algorithms in order to efficiently increase code coverage of the test cases.

However, for fuzzing protocol, American Fuzzy Lop are seriously lacking: firstly, it works only with program that reads input from file or standard input, so there are no network modules for fuzzing network protocol. Secondly, a server that runs the protocol usually has an infinity loop in order to serve multiple clients, so American Fuzzy Lop cannot have a status code returned by the program.

This thesis introduces some solutions to cover those disadvantages of American Fuzzy Lop. Moreover, the results from some benchmarks with other kinds of network fuzzing indicate that this approach yields a result which is a multitude of times better than the current state-of-the-art. This fuzzer not only confirms/reproduces known vulnerabilities but also discovers some new bugs during just few months of experiments, and few of them got recognized with CVE numbers.

ACKNOWLEDGEMENTS

Foremost, I would like to express my cordial appreciation and gratitude to my supervisor, Dr. Tran Quang Duc and my Dr. Nguyen Anh Quynh for all the support and encouragement during the time I have been working on this thesis. Without their guidance and constant feedback, my graduation thesis would not have been completed.

Secondly, I am grateful to my friends, Do Quang Thanh who has given me many interesting ideas. He also provided me extensive personal and professional guidance and taught me very well about scientific research, coding and life. I would especially like to thank all the fellow friends on Github, Stackoverflow, Medium for all the tutorials, answers and supports during the time of this work.

Additionally, my sincere thanks also goes to all the lecturer, professors of HUST for the intensive knowledge and sincere advice and experiences, which assemble strong foundation for our - my classmate's and my - future career.

Lastly, I would like to dedicate this work to my family who always loves and supports me unconditionally for every single step in my life. Their encouragements give me the strength and motivations to finish this work.

Hanoi, 27th, December 2019

Do Minh Tuan

Acronyms

AFL American Fuzzy Lop. 2, 7, 9, 10, 12, 14, 15, 17, 18, 23–26, 33, 34, 39, 45, 46

AMF All message fuzzing. 26, 39

FFW Fuzzing For Worm. 2, 18–21, 23, 40, 46

FTP File Transfer Protocol. 26

GCC The GNU Compiler Collection. 32, 33

ICMP Internet Control Message Protocol. 31

IoT Internet of Things. 1–3, 17

MQTT MQ Telemetry Transport. 42, 43

PF Protocol Fuzzer. 2, 29, 39

SHM share memory. 10

SMF Single message fuzzing. 26, 39

TCP Transmission Control Protocol. 31, 38

UDP User Datagram Protocol. 31

Contents

Requirements for The Thesis	i
Tóm Tắt Nội Dung Đề Án Tốt Nghiệp	ii
Abstract of Thesis	iii
Acknowledgements	iv
Acronyms	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	2
1.3 Organization	2
2 Background	3
2.1 Security Vulnerability	3
2.2 An overview of fuzzing technique	4
2.3 Grey box fuzzing technique	7
2.3.1 Motivation	7
2.3.2 American Fuzzy Lop [1]	7
3 Problems and Solutions	17
3.1 Previous work	17
3.1.1 American Fuzzy Lop and Preeny	17

3.1.2	Fuzzing For Worm	18
3.1.3	PULSAR	21
3.2	Problems and solutions	22
3.2.1	Pause / Resume a process in Linux	23
3.2.2	Hooking system APIs	25
3.2.3	Context knowledge	26
3.2.4	Inter-process communication	26
4	Hermes - The next generation of protocol fuzzer	29
4.1	Basic concepts and structures	29
4.2	Interceptor	30
4.3	Compilation and Instrumentation	32
4.4	Interprocess Communications and Hooking System APIs	34
4.4.1	Architecture of fuzzer	34
4.4.2	Hooking System APIs	35
4.5	Workflow in details	39
4.6	Evaluation	40
4.6.1	Echo server	40
4.6.2	Chat room	41
4.6.3	MQTT broker of Mongoose Cesanta	42
5	Conclusions and Future work	45
5.1	Conclusions	45
5.2	Thesis contributions	46
5.3	Limitations	46
5.4	Future Work	47
	References	48

List of Figures

1	An example of Buffer overflow	5
2	American Fuzzy Lop screen	7
3	Basic blocks	8
4	Feedback driven fuzzing	8
5	Different coverages	10
6	New coverages	11
7	Normal design in fuzzing	13
8	Design of AFL in fuzzing	15
9	AFL's workflow	16
10	Internal architecture of FFW	19
11	Interceptor's model	19
12	Honggfuzz	20
13	Protocol between FFW and Honggfuzz	21
14	(1): Model interference; (2): Testcase generation; (3): Model coverage	22
15	Linux process life cycle	24
16	An example of stateful protocol	29
17	a) Original input b) Normal mutation c) Stateful mutation	30
18	Protocol Fuzzer's interceptor	31
19	GCC's workflow	32
20	Instrumentation	33
21	PF's components	35
22	Hermes's workflow	44

List of Tables

1	Pros and cons of protocol fuzzers	23
2	EchoServer fuzzinng performances	40
3	Server’s command	41
4	Chatroom fuzzing performances	42
5	mqtt_broker fuzzing performances	43

1.1 Motivation

Nowadays, people are talking about IoT over the places, from the newspaper to blogs, so it becomes one of the most common buzzwords in technology circles. This technical term is all about connecting machines and systems together via sensors and actuators, then that meaningful information from these systems can be collected and actions taken to enhance human productivity and efficiency. IoT is driving the proliferation of connected devices from around a billion plus today to over 50 billion in the next decade [2]. By doing so, IoT takes the meaning of inter-connectivity to a whole new level, but it also increases the risk in security.

Internet is the foundation and core supporting IoT hence almost all the security threats that lie within Internet propagate to IoT as well. Furthermore, the fast development and wider adoption of IoT devices in our lives signifies the urgency of addressing these security threats before deployment. Although a lot of companies state that their technologies are secured and protected, they are still prone to various types of attacks. Since the interconnected devices have a direct impact on the lives of users, there is a need for a well-defined security threat classification and a proper security infrastructure with new systems and protocols that can mitigate the security challenges regarding privacy, data integrity, and availability in IoT .

There are many kinds of security vulnerability in IoT device. From simple flaws like weak, guessable, or hardcoded passwords to more complicated bugs in communication among devices. In this work, the thesis focuses on memory corruption, which is an error occurred when behaviors exceeds the intention of original program/language constructs. In order to find this type of vulnerability, we design a Fuzzing System to generates an input to make a process being crashed over the network.

1.2 Contributions

The principal contributions of this thesis are[3]:

- Give an overview of security vulnerability and fuzzer.
- Provide a detail explanation of the architecture and algorithms of AFL and its pros and cons in network fuzzing.
- Address the problem of current network fuzzer and propose some solutions to cover those disadvantages.
- Conduct a set of experiments to evaluate the performance among "Fuzzing For Worm"[4], Pulsar[5] and our fuzzers. Based on such experiments, we explicitly address the improvements and the limitations of our proposals when compares with FFW [4].

Furthermore, this work also connects to Redback project, which was presented by us in some famous conferences:

- XCon - Beijing, China 2019
- T2 Conference - Helsinki, Finland 2019
- Insomni'hack - Geneva, Switzerland 2020
- BlackHat Asia - Singapore, 2020

1.3 Organization

The remains of thesis is organized as follow:

Chapter 2: Background provides a review of security vulnerability in IoT, fuzzing's concept and analyse about mechanism of American Fuzzy Lop

Chapter 3: Problems and Solutions indicates problems of current fuzzers in fuzzing network protocols and presents some proposed approaches for covering disadvantages of AFL

Chapter 4: Hermes - The next generation of protocol fuzzer describes in detail the architecture, design, mechanism, ... of our PF, represents experimental results and provides a comparison of the proposed approach's performance.

Chapter 5: Conclusions and Future work sums up the thesis with its improvements and limitations, then discusses the future work to develop our two proposals.

In this chapter, we provide a brief review of Security Vulnerability and Fuzzing

2.1 Security Vulnerability

In cyber security, a vulnerability is a weakness which can be exploited by a cyber attack to access anonymously to or perform unauthorized actions on a computer system.

However, a security risk is often incorrectly classified as a vulnerability, then the use of vulnerability with the same meaning of risk can lead to some confusions. The risk is the potential of a significant impact resulting from the exploit of a vulnerability, so there are vulnerabilities without risk: for example when the affected asset has no return value. A vulnerability with one or more known instances of working and fully implemented attacks is classified as an exploitable vulnerability. The window of vulnerability is the time from when the security hole was introduced, to when a security fix was available/deployed. The vulnerability is also categoried based on that time:

- 0-day vulnerability: No fixes yet on the system, very dangerous
- N-day vulnerability: Has fixed on some system, but maybe many more haven't updated yet, less dangerous

Security bug is a narrower concept: there are vulnerabilities that are not related to software such as hardware, site, personnel vulnerabilities.

There are many kinds of security vulnerability in the worldwide, but regarding about IoT, we can classified into 5 categories:

- Insufficient Authentication/Authorization: a kind of vulnerability occurs when attacker uses weak credentials or captures plain text credentials to access web interface. The impact results in data loss, denial of service and can lead to complete device take over. Many routers has been compromised because of weak password from administrator.

- **Insecure Web Interface:** this point concerns security related issues with the web interfaces built into IoT devices that allows a user to interact with the device, but at the same time could allow an attacker to gain unauthorised access to the device through some dangerous flaws such as SQL-injection, command-line injection, ...
- **Insecure Network Services:** Attackers use vulnerable network services to attack the device itself or bounce attacks off the device. Attackers can then use the compromised devices to facilitate attacks on other devices.
- **Lack of Transport Encryption:** This deals with data being exchanged with the IoT device in an unencrypted format. This could easily lead to an intruder sniffing the data and either capturing this data for later use or compromising the device itself.
- **Privacy Concerns:** these information are generated by the collection of personal data in addition to the lack of proper protection of that data. Privacy concerns are easy to discover by simply reviewing the data that is being collected as the user sets up and activates the device. Automated tools can also look for specific patterns of data that may indicate collection of personal data or other sensitive data.

The thesis will focus more on Insecure Network Services, specifically, memory corruption in the implementation of network protocol.

On the other hands, in order to recognize the efforts of security researchers in the world, a public system, called Common Vulnerabilities and Exposures (CVE) system, was developed to provide reference-methods for publicly known information-security vulnerabilities and exposures. The National Cybersecurity FFRDC, operated by the Mitre Corporation, maintains the system, with funding from the National Cyber Security Division of the United States Department of Homeland Security.

2.2 An overview of fuzzing technique

In the evaluation of software for exploitable vulnerabilities, automated testing is heavily used. The most popular testing technique that has been proven effective is fuzz testing, or fuzzing, a technique which generates inputs to a program in order to exercise different parts of the program and attempt to induce undefined or undesired behavior.

Prevention of vulnerabilities may involve compile-time code modification utilities, which can mitigate some programmer errors. For example, a vulnerability, called "Buffer Overflow" occurs when the program receives data whose size is bigger than the place-holder, which lead to remote code execution. Modern compiler has tried to extenuate the impact of result by adding canary to buffer or randomize the position of place-holder.

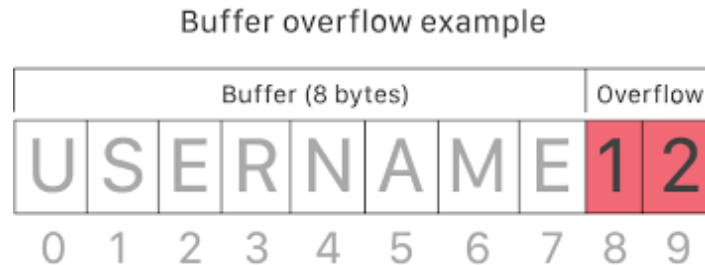


Figure 1: An example of Buffer overflow

However, these strategies can not consider vulnerabilities that will manifest themselves mainly during runtime of the program, or vulnerabilities in the logic of the program's execution. It is hard to enumerate and isolate in practice, as in a real-world software project, there are many paths of execution and only some of them, with particular permutations, may invoke undefined or unintended program behavior. Accordingly, approaches have been developed to attempt to reach and identify these execution paths.

A common way for identifying execution permutations which exhibit unintended program behavior is to provide a large number of inputs into the program, and observe the result. A number of techniques make use of this model so as to drive vulnerability detection, including fuzzing.

Generally, there are three types of technique, based on the approaches towards mutating and generating inputs in the context of fuzzing:

- **Black box fuzzing:** A fuzzer will input massive amounts of random or semi-random data into another program to see how it responds, then reports back with details on how the program responded to the "fuzz test"[6]
- **White box fuzzing:** make use of program inspection, or utilize knowledge of the format of the input for making more meaningful input mutations. In this type of fuzzing, source code must be visible to the fuzzer.
- **Grey box fuzzing:** instrument some mini-code inside the program during the compiling time, so that we can keep track of which program paths are covered, and how frequently these program paths are executed. Inputs are evaluated by their propensity for higher code coverage, and interesting inputs are continuously fed into the queue of input seeds.

In order to assess the efficacy of fuzzing techniques, a number of metrics must be taken into consideration:

- The amount of novel vulnerabilities discovered in the program under test.
- The efficiency with which vulnerabilities are discovered.
- Code coverage of the program under test.

The primary goal of each fuzzer is to find novel vulnerabilities in real-world programs. This metric can be evaluated from fuzzing runs on up-to-date, previously well-fuzzed software. The ability to discover novel vulnerabilities shows the effectiveness of an evaluated fuzzing technique.

Because fuzzing is not a finite process for any programs, the efficiency with which a fuzzing technique can discover vulnerabilities is an important evaluation. In scope of fuzzing, this metric will be affected by the choice of input mutation strategy, the details of which will differ between fuzzing approaches and techniques.

A higher code coverage indicates that a larger surface area of the program under test was actually executed and evaluated for vulnerabilities or errors. Code coverage is often used as an optimization criteria to indicate the effectiveness of a particular input.

The software under test will generally vary based on the fuzzing approach being assessed.

- Closed-source software is usually tested by black box fuzzing techniques, since these techniques are focused on detecting undesired behavior in programs which are not easily instrumentable to other code-level analysis techniques.
- Open-source software will typically be evaluated by white box and grey box fuzzing techniques.

In scope of the thesis, we will mainly discuss about grey box fuzzing technique for open-source program.

2.3 Grey box fuzzing technique

2.3.1 Motivation

Grey box fuzzing, as the name implies, finds a middle space between black and white box fuzzing approaches. Black box fuzzers do not utilize program structure to make more informed decisions on input generation and the coverage is not really measured. Conversely, white box fuzzers explicitly solve program constraints in order to traverse the possible execution paths of the program. This method is a thorough and guided approach, but it may be typically much less efficient due to its heavy reliance on constraint solving algorithms. Grey box fuzzers make use of the availability of program structure or code in order to explicitly evaluate what kind of coverage a fuzzed input can deliver, and uses this coverage information to guide input mutation or generation. Accordingly, grey box fuzzing techniques find a middle ground between program inspection and efficient input generation.

2.3.2 American Fuzzy Lop [1]

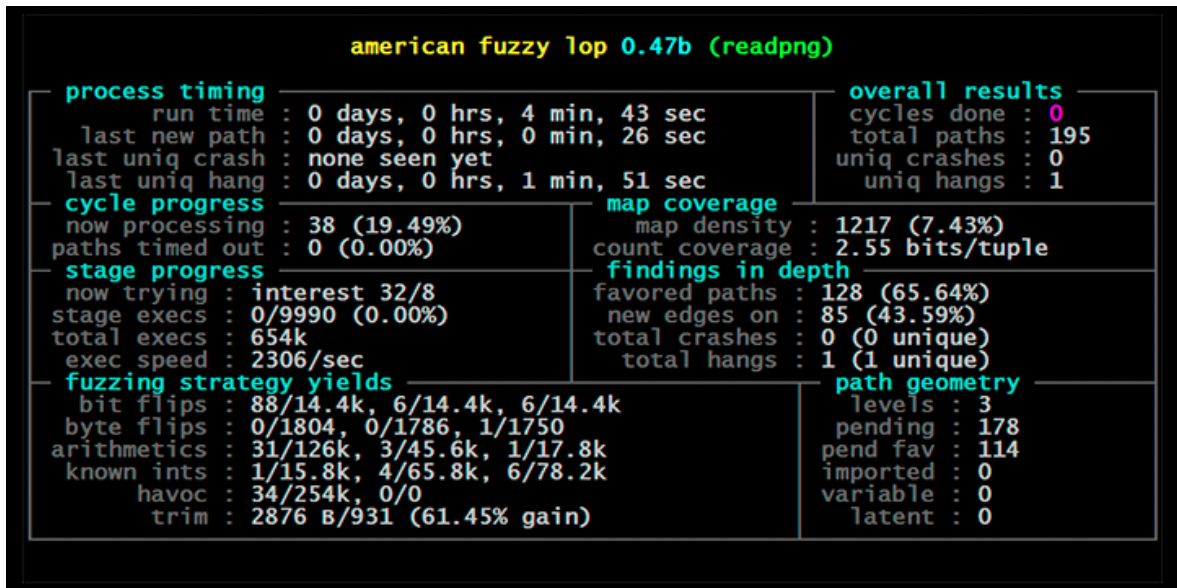


Figure 2: American Fuzzy Lop screen

AFL is a grey box fuzzer which is considered state-of-the art, and is accordingly extended upon in many grey box fuzzing techniques. AFL leverages coverage-feedback to learn how to reach deeper into the program. It is not entirely blackbox because AFL leverages at least some program analysis. It is not entirely whitebox either because AFL does not build on heavyweight program analysis or constraint solving. Instead, AFL uses lightweight program instrumentation to collect some information about the (branch) coverage of a generated input. If a generated input increases coverage, it is added to the seed corpus for further fuzzing.

BASIC BLOCKS

In compiler construction, a basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. This restricted form makes a basic block highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks form the vertices or nodes in a control flow graph.

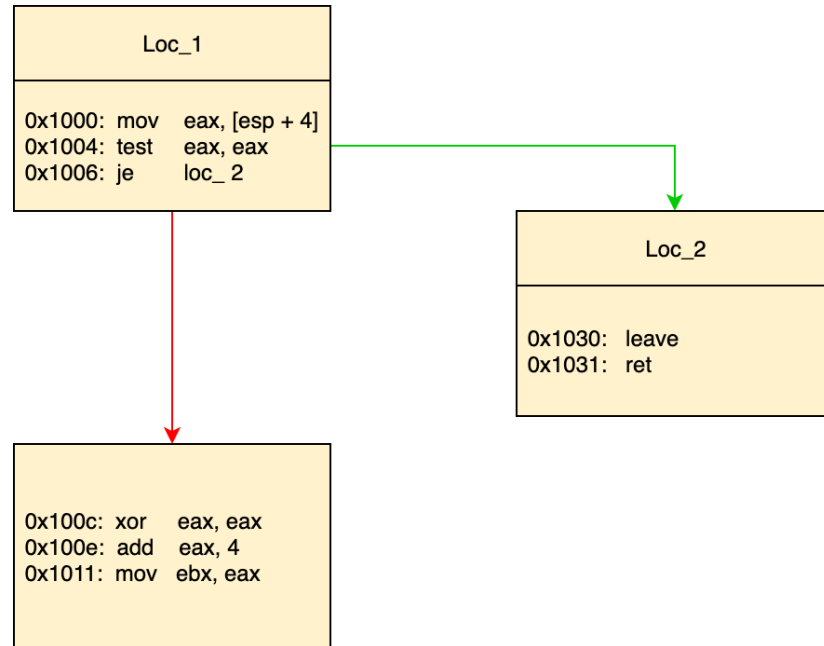


Figure 3: Basic blocks

FEEDBACK DRIVEN FUZZING

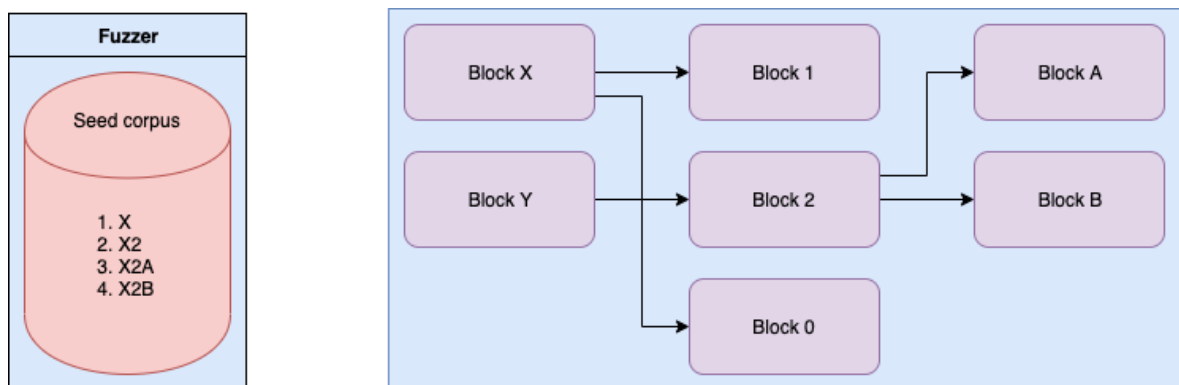


Figure 4: Feedback driven fuzzing

Feedback driven fuzzing is a smart technique which has a capability of measuring the coverage of execution. Regarding of the basic block level, when we change some values of input, we observe the list of blocks that program has met, then if we reach to a new block, we consider the mutated input as a new corpus and continue the fuzzing. For example, in Figure 3, the initial input is X, then we have a list of blocks that the

process has called, which now includes only Block X. Next, we try to random the input and make it into X2, we see in the list there is a new element, Block 2, so we store X2 as a new corpus (Figure 4).

With the coverage guiding, fuzzer will achieve a better performance than using normal fuzzing. In details, consider that we have the following block of code:

```

if (input[0] == 'A') {
    if (input[1] == 'B') {
        if (input[2] == 'C') {
            if (input[3] == 'D') {
                // crash
            }
        }
    }
}

```

In order to reach to crash-place, the normal fuzzer needs a total 2^{32} times brute-forcing, assume this is a 32-bit application. This is really huge number, however, if we use coverage-guidance fuzzer, the corpus will update itself after each iteration:

1. {}
2. {"A"}
3. {"A", "AB"}
4. {"A", "AB", "ABC"}
5. {"A", "AB", "ABC", "ABCD"}
- ...

Then the result should be 2^{10} .

COVERAGE MEASUREMENT

To instrument a program, AFL injects a piece of assembly code right after every conditional jump instructions. When executed, this so-called trampoline assigns the exercised branch an unique identifier and increases a counter that is associated with this branch. For efficiency, only a coarse branch hit count is maintained. In other words, for each input the fuzzer knows which branches and roughly how often they are exercised. The instrumentation is usually done at compile-time. The code injected at branch points is equivalent to Algorithm 1.

The *cur_location* value is generated randomly at compile-time to simplify process of linking complex projects and keep the XOR output distributed uniformly.

Algorithm 1 Update bitmap

```

1:  $cur\_location \leftarrow nonce\_compile\_time$ 
2:  $shared\_mem[cur\_location \oplus prev\_location]$  increment
3:  $prev\_location \leftarrow cur\_location \gg 1$ 

```

The `shared_mem[]` array is a 64 kB SHM region passed to the instrumented binary by the fuzzer. Every byte set in the output map can be thought of as a hit for a particular (branch_src, branch_dst) tuple in the instrumented code. This can be done by calling functions `shmget()` and `shmat()` from system. In details, first `shmget()` returns to AFL the identifier of the System V shared memory segment, in other words, `shared_mem[]`. Then both AFL and the instrumented assembly code call `shmat()` to obtain the address of SHM. Before that, AFL also passed the identifier of SHM to target binary through environment variable.

This form of coverage provides considerably more insight into the execution path of the program than simple block coverage. In particular, it trivially distinguishes between the following execution traces:

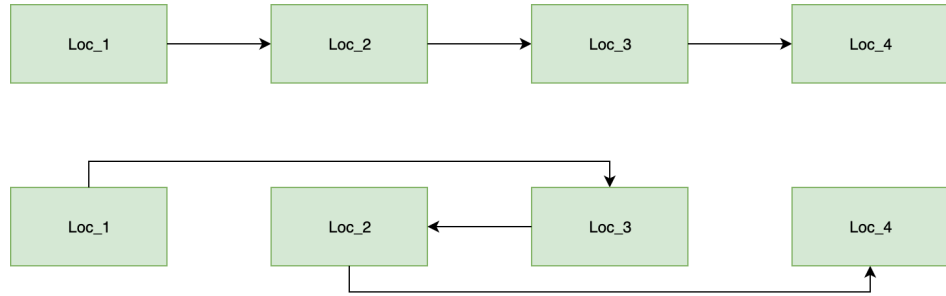


Figure 5: Different coverages

This aids the discovery of subtle fault conditions in the underlying code, because security vulnerabilities are more often rely on unexpected or incorrect state transitions than with merely reaching a new basic block.

The reason for the shift operation in the last line of the pseudocode shown earlier in this section is to maintain the directionality of tuples (without this, $A \oplus B$ would be no differences to $B \oplus A$) and to retain the identity of tight loops (otherwise, $A \oplus A$ would be obviously equal to $B \oplus B$).

DETECTING NEW BEHAVIORS

The fuzzer preserves a global map of tuples seen in previous executions; this data can be quickly compared with individual traces and updated in just a couple of dword- or qword-wide instructions and a simple loop.

When a mutated input produces an execution trace containing new tuples, the corresponding input file is kept and routed for additional processing later on. Inputs that do not trigger new local-scale state transitions in the execution trace (i.e., produce no

new tuples) are discarded, even if their overall control flow sequence is unique.

This approach allows for a very fine-grained and long-term exploration of program state while not having to perform any computationally intensive and fragile global comparisons of complex execution traces, also avoiding path explosion.

To illustrate the properties of the algorithm, consider that the second trace shown in Figure 6 would be considered substantially new because of the presence of new tuples (31, 15).

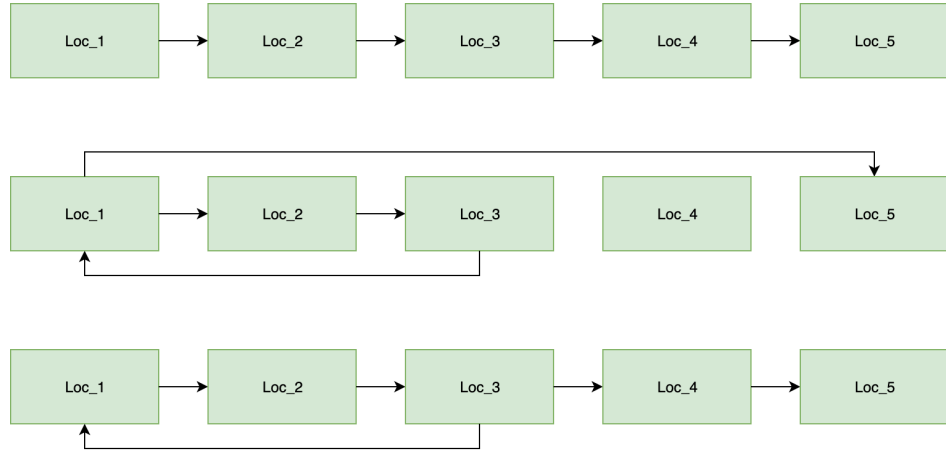


Figure 6: New coverages

At the same time, with the second sequence processed, the third pattern in Figure 6 will not be seen as unique, in spite of having an obviously different overall execution path.

In addition to detecting new tuples, the fuzzer also considers coarse tuple hit counts. These are divided into several buckets: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+. In other words, the number of buckets is an implementation artifact: it allows an in-place mapping of an 8-bit counter generated by the instrumentation to an 8-position bitmap relied on by the fuzzer executable to keep track of the already-seen execution counts for each tuple.

Changes within the range of a single bucket are ignored; transition from one bucket to another is flagged as an interesting change in program control flow, and is routed to the evolutionary process.

The hit count behavior provides a way to distinguish between potentially interesting control flow changes, such as a block of code being executed twice when it was normally hit only once. At the same time, it is fairly insensitive to empirically less notable changes, such as a loop going from 47 cycles to 48. The counters also provide some degree of "accidental" immunity against tuple collisions in dense trace maps.

The execution is handled fairly heavily through memory and execution time limits; by default, the timeout is set at 5x the initially-calibrated execution speed, rounded up to 20 ms. The aggressive timeouts are meant to prevent dramatic fuzzer performance

degradation by descending into tarpits that improve coverage by 1% while being 100x slower; AFL pragmatically rejects them and hope that the fuzzer will find a less expensive way to reach the same code. Empirical testing strongly suggests that more generous time limits are not worth the cost.

FUZZING STRATEGIES

One of the interesting side effects of the design of AFL is that it provides a rare feedback loop to carefully measure what types of changes to the input file actually result in the discovery of new branches in the code. This data is particularly easy to read because the fuzzer also approaches every new input file by going through a series of progressively more complex, but exhaustive and deterministic fuzzing strategies before diving into purely random behaviors. The reason for this is the desire to generate the simplest and most elegant test cases first.

- *Walking bit flips*: the first and most rudimentary strategy employed by AFL involves performing sequential, ordered bit flips. The stepover is always one bit; the number of bits flipped in a row varies from one to four.
- *Walking byte flips*: a natural extension of walking bit flip approach, this method relies on 8-, 16-, or 32-bit wide bitflips with a constant stepover of one byte.
- *Simple arithmetics*: to trigger more complex conditions in a deterministic fashion, the third stage employed by AFL attempts to subtly increment or decrement existing integer values in the input file; this is done with a stepover of one byte. When it comes to the implementation, the stage consists of three separate operations. First, the fuzzer attempts to perform subtraction and addition on individual bytes. With this out of the way, the second pass involves looking at 16-bit values, using both endians - but incrementing or decrementing them only if the operation would have also affected the most significant byte (otherwise, the operation would simply duplicate the results of the 8-bit pass). The final stage follows the same logic, but for 32-bit integers.
- *Known integers*: the last deterministic approach employed by AFL relies on a hardcoded set of integers chosen for their demonstrably elevated likelihood of triggering edge conditions in typical code (e.g., -1, 256, 1024, MAX_INT-1, MAX_INT). The fuzzer uses a stepover of one byte to sequentially overwrite existing data in the input file with one of the approximately two dozen "interesting" values, using both endians (the writes are 8-, 16-, and 32-bit wide).
- *Stacked tweaks*: with deterministic strategies exhausted for a particular input file, the fuzzer continues with a never-ending loop of randomized operations that consist of a stacked sequence of:

- Single-bit flips,
 - Attempts to set "interesting" bytes, words, or dwords (both endians),
 - Addition or subtraction of small integers to bytes, words, or dwords (both endians),
 - Completely random single-byte sets,
 - Block deletion,
 - Block duplication via overwrite or insertion,
 - Block memset.
- *Test case splicing*: this is a last-resort strategy that involves taking two distinct input files from the queue that differ in at least two locations; and splicing them at a random location in the middle before sending this transient input file through a short run of the "stacked tweaks" algorithm.

THE FORK SERVER

The most common way to fuzz data parsing libraries is to find a simple binary that exercises the interesting functionality, and then simply keep executing it over and over again with slightly different, randomly mutated inputs in each run. In such a setup, testing for evident memory corruption bugs in the library can be as simple as doing *waitpid()* on the child process and checking if it ever dies with *SIGSEGV*, *SIGABRT*, or something equivalent.

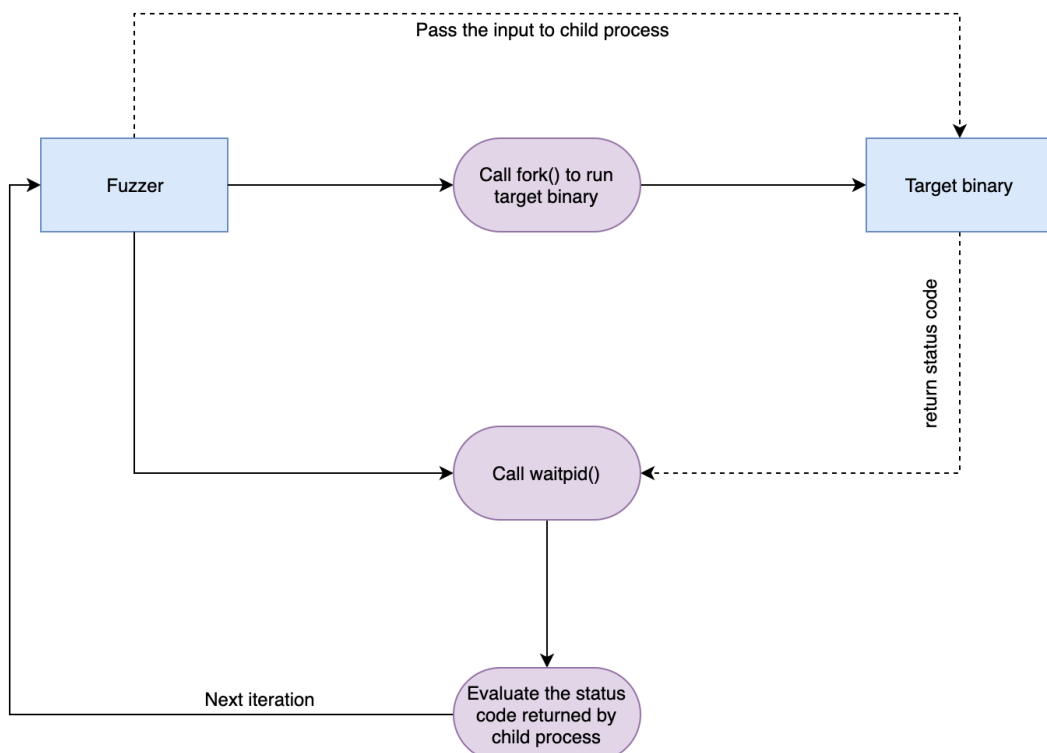


Figure 7: Normal design in fuzzing

This approach is favored by security researchers for two reasons. Firstly, it eliminates the need to dig into the documentation, understand the API offered by the underlying library, and then write custom code to stress-test the parser in a more direct way. Secondly, it makes the fuzzing process repeatable and robust: the program is running in a separate process and is restarted with every input file, so you do not have to worry about a random memory corruption bug in the library clobbering the state of the fuzzer itself, or having weird side effects on subsequent runs of the tested tool.

However, there is also a problem: especially for simple libraries, you may end up spending most of the time waiting for *execve()*, the linker, and all the library initialization routines to do their job. So the AFL's solution is that it injects a small piece of code into the fuzzed binary - a feat that can be achieved via *LD_PRELOAD*, via *PTRACE_POKE TEXT*, via compile-time instrumentation, or simply by rewriting the ELF binary ahead of the time. The purpose of the injected shim is to let *execve()* happen, get past the linker, and then stop early on in the actual program, before it gets to processing any inputs generated by the fuzzer or doing anything else of interest.

Then, the shim simply waits for commands from the fuzzer; when it receives a "go" message, it calls *fork()* to create an identical clone of the already-loaded program; due to the powers of copy-on-write, the clone is created very quickly yet enjoys a robust level of isolation from its older twin. Within the child process, the injected code returns control to the original binary, letting it process the fuzzer-supplied input data (and suffer any consequences of doing so). Within the parent, the shim relays the PID of the newly-created process to the fuzzer and goes back to the command-wait loop. (Figure 8)

AFL also creates an interprocess communication between Fuzzer and Fork Server. By calling *pipe()* and *dup()* to clone some more file descriptors from parent process to child process, AFL and Fork Server can easily talk to each other with functions *read()* and *write()*.

WORKFLOW

An overview of AFL's architecture is shown in Figure 9. However, before using AFL, we have to compile the target with a custom compiler, which is a built-in tool of AFL project, so that the target will be injected with some instrumentation code. Basically, we can divide the process into three main phases.

At the very beginning, AFL sets up the environment, initializes the shared memories and also creates a forklserver, which is the container includes target binary and injected shim. These things are done only once, before going deeper. Then, AFL reads the seed corpus, a set of valid and interesting inputs that serve as starting points

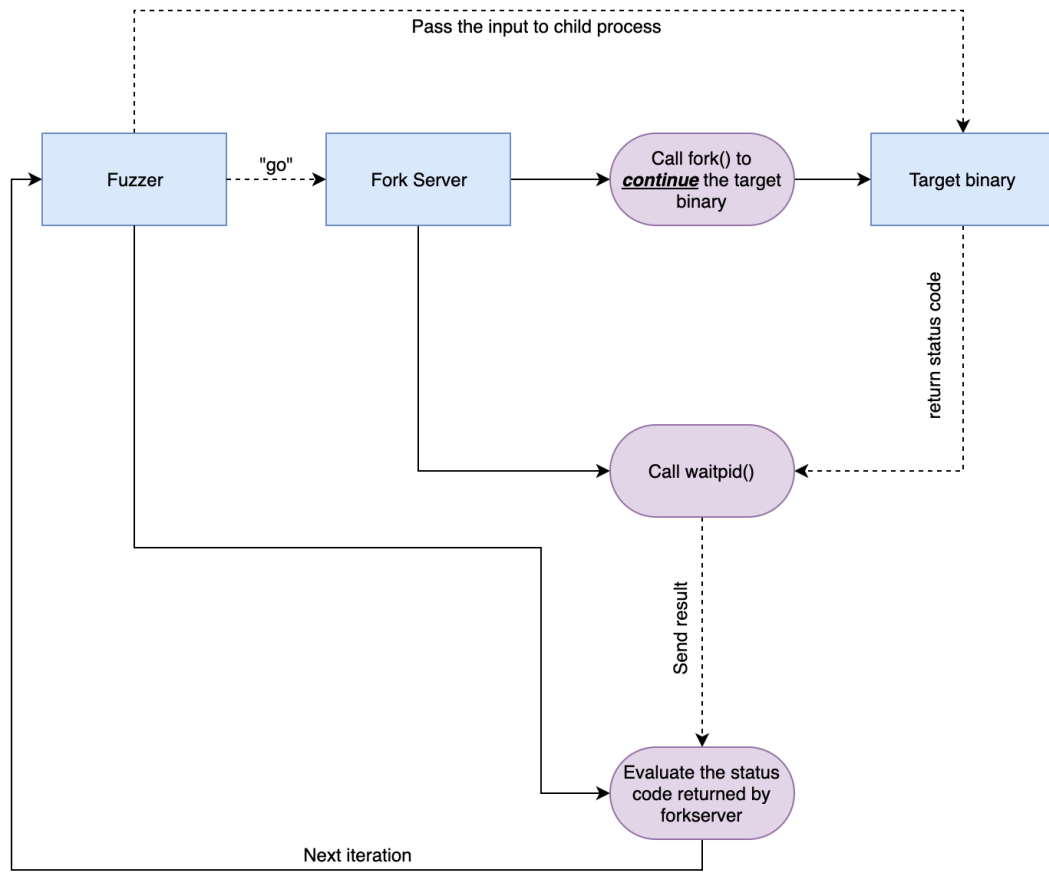


Figure 8: Design of AFL in fuzzing

for a fuzz target, and pushes them to a queue. Moreover, AFL pretreats the queue in order to optimize performance of further processes.

For each fuzzing loop, AFL pops out the first test case in queue, then mutates it according to some algorithms, which is mentioned in the previous section and saves it into a file. After making the input, AFL sends a signal to forkserver through file descriptor and wait for the result. Meanwhile, forkserver has received the message so it starts forking the target and call *waitpid()* to retrieve the status code of child process. The target binary runs as usual, and due to the injected instrumentation code, it also updates the local bitmap, which is shared memory between AFL and the current process.

Forkserver gets the status code returned by child process, and sends the result back to AFL through file descriptor. Then AFL evaluate the status code, if target has been crashed or timeout, AFL puts the current input to corresponding placeholder. Otherwise, if target has exited normally, AFL compares the local bitmap with a global bitmap, if a new bit has been set, then the input leads to a new coverage. Therefore, AFL stores the input as a new corpus and push it to the queue and start a new fuzzing loop. AFL also observes the bitmap and the way how inputs are changed to determine next mutation.

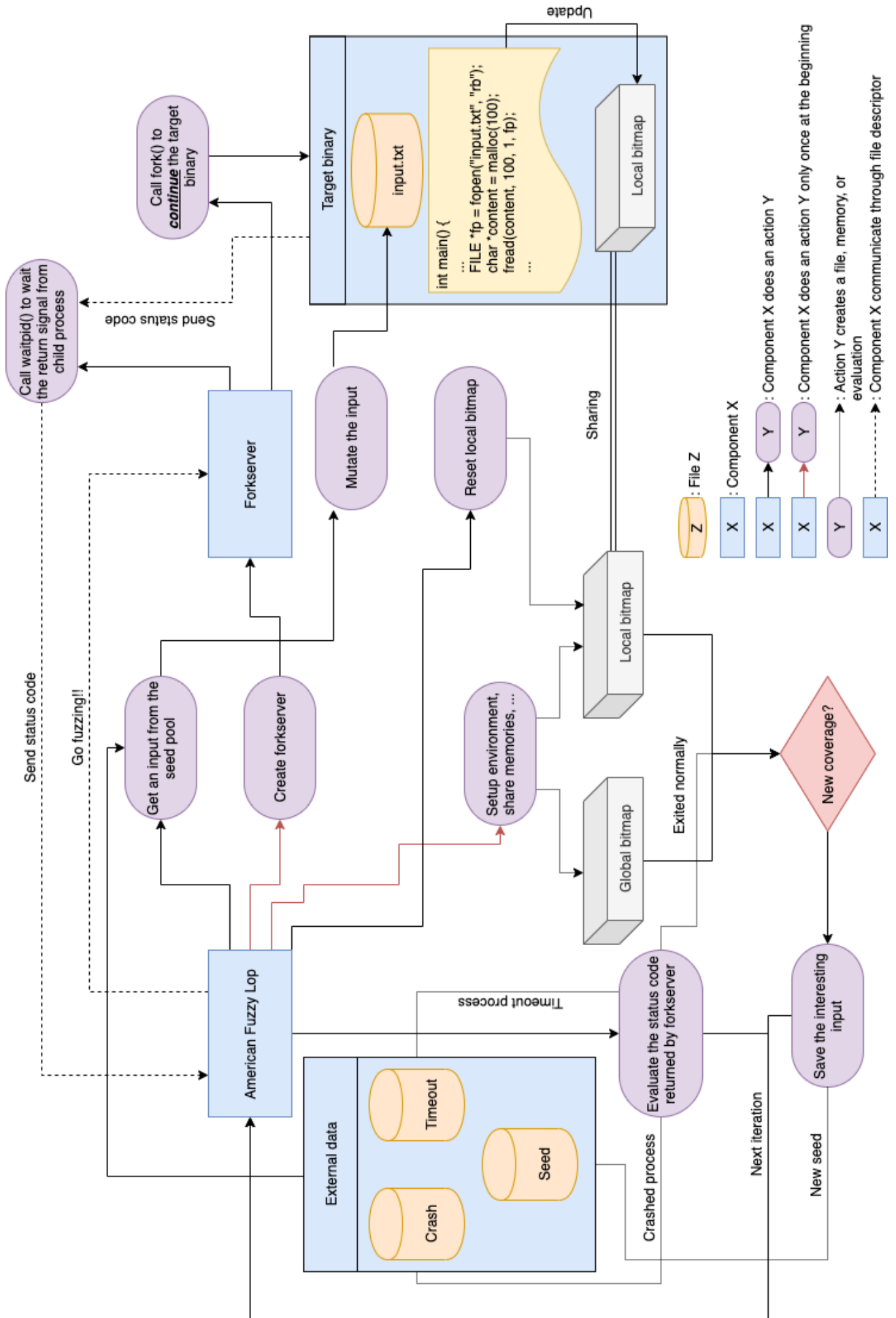


Figure 9: AFL's workflow

In the IoT era, every components are connected in networks and they communicate to each other by protocols. The complexity of many network protocols combined with time-to-deliver constraints or insecure coding practices make errors inevitable. As a result, new vulnerabilities in network-based applications are found and advertised on a daily basis. A single security vulnerability in the implementation of a protocol can compromise a network service and expose sensitive data to an attacker. For example, a flaw in the implementation of the universal plug-and-play protocol rendered roughly 23 million routers vulnerable to attacks from the Internet [7]. Therefore, we need effective methods and tools to identify bugs in network-based applications before they are deployed on live networks.

Fuzzing has a number of advantages over other testing techniques, such as manual code review, static analysis, and model checking. Fuzzing can be applied to programs whose source code is not available (black-box fuzzing technique). Second, fuzzing is largely independent of the internal complexity of the examined system, overcoming practical limits that prevent other testing methods (e.g., static analysis) from being able to operate on large applications (grey-box fuzzing technique). Being completely independent of the tested program's internals, the same fuzzing tool can be reused to test similar programs regardless of the language in which they are implemented. Finally, bugs found with fuzzing are reachable through user input, and, as a consequence, are exploitable.

3.1 Previous work

3.1.1 American Fuzzy Lop and Preeny

American Fuzzy Lop[1] is a file based fuzzer which feeds input to program via file descriptors (standard input, file on disk, ...). Using it with network program like server's or clients is not possible in the original state. More on that, there is an AFL patch to fuzz network app, but this one is not merged upstream and it is older

compared to currently released versions.

A temporary solution has been introduced, which uses another project, called *preeny*[8], to support for fuzzing network. The project provides library which when used with *LD-PRELOAD* can desocket the network program and make it read from standard input. The general idea can be summarized as follows:

- *desock.so* is a shared library provided by *preeny* works only with read and write (or rather other system call does not work with standard input) system calls and then we have to replace any reference to *send*, *sendto*, *recv* and *recvfrom* with *read* and *write* system calls respectively.
- If the network program is using forking or threading model, remove all those and make it plain simple program which receives request and sends out response.
- Run AFL under experimental persistent mode, which uses *__AFL_LOOP* for loop iteration. Via some minor modifications to the source code, it lets the tester control when AFL forks the target application and when AFL feeds new input to the application

This mode solves the problem of waiting for a program to start; the tester can now feed fuzzed inputs to the target program without restarting it. The major caveat here is that the tester must be careful to reset the state of the application between each iteration of fuzzing. If the state is not reset carefully, the bugs will be found in test harness instead of the target.

3.1.2 Fuzzing For Worm

Fuzzing For Worm[4] is a project which provides capabilities to fuzz network servers/services by intercepting valid network communication data, then replay it with some fuzzing. FFW can fuzz open source applications and supports feedback driven fuzzing by instrumenting honggfuzz, for both open and closed source apps. In comparison with the current alternatives, FFW is the most advanced, feature-complete and tested network fuzzer.

This project includes some features:

- Fuzzes all kind of network protocol (HTTP, MQTT, SMTP, ...)
- No modification of the fuzzing target needed (at all)
- Has feedback-driven fuzzing (with compiler support, or hardware based)
- Can fuzz network clients too (wip)
- Fast fuzzing setup (no source code changes or protocol reversing needed)
- Reasonable fuzzing performance

INTERNAL ARCHITECTURE

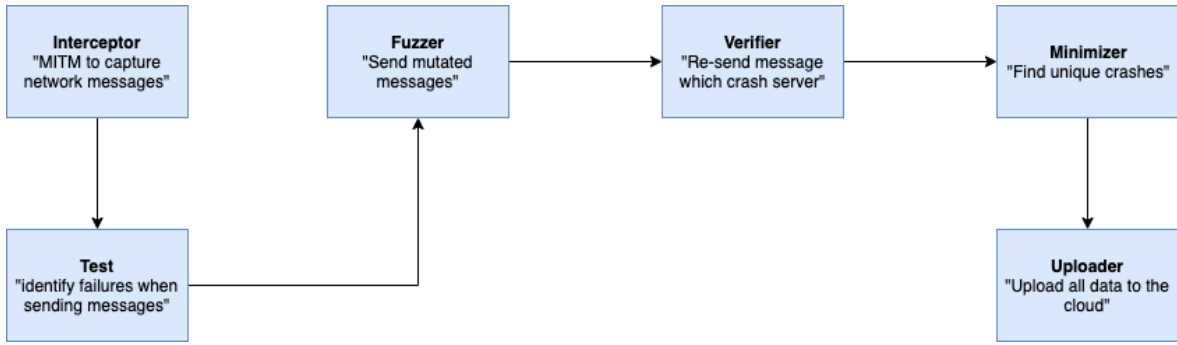


Figure 10: Internal architecture of FFW

Figure 10 shows a general architecture of FFW. The fuzzer first records a transferring data from client to server. It is possible to capture it by using tcpdump and then convert the pcap to pickle file, however we can intercept it on the fly. FFW tests it if the intercepted things really work before sending some mutated messages. There is also a verifying stage, which help to avoid false positive cases. Finally, FFW minimizes it because there may be a lot of duplicated crashes.

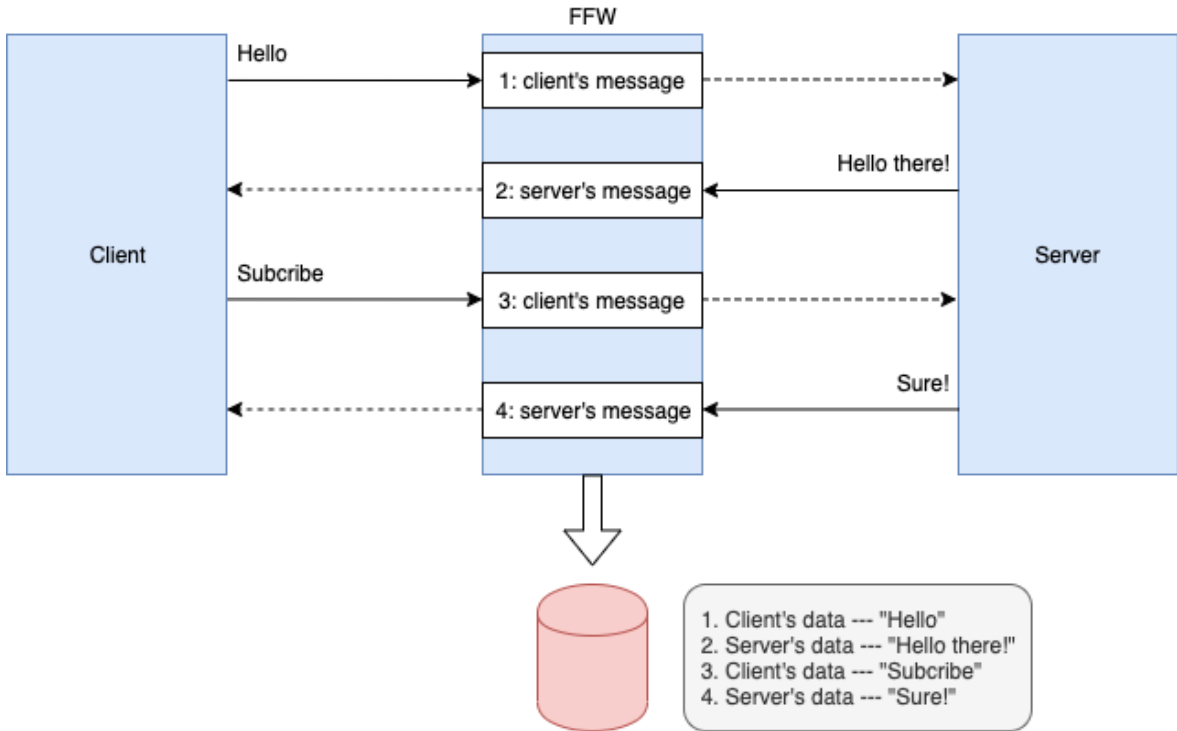


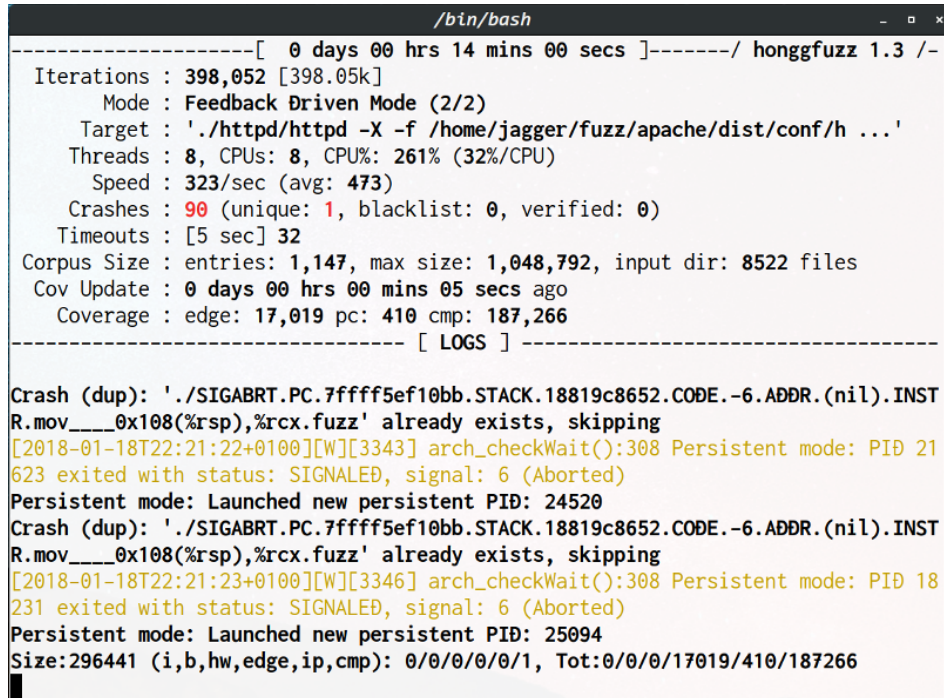
Figure 11: Interceptor's model

Figure 11 shows a specific case for studying. This is considered as a communication of Mongoose server, particularly there are four messages. FFW writes all recorded data to a file, and after finishing the protocol, FFW reads the file, selects a random message then mutates it and imitate the client to send data to server. The mutation is done by using external tools such as Radamsa[9].

In fact, it is difficult to detect crashes. Normally, we send the first packet to server which makes it crashes, and then ask the process whether it is still alive or not. However, due to some lag or latency, it may take time to identify the process as being crashed. FFW chooses a different way to get the knowledge of status of server, by checking the connection at TCP handshake from the fake-client to server.

HONGGFUZZ INTEGRATION

Honggfuzz[10] is a security oriented, feedback driven, evolutionary fuzzer. It's multi-process and multi-threaded, which means no need to run multiple copies of your fuzzer, as honggfuzz can unlock potential of all your available CPU cores with a single supervising process. The file corpus is automatically shared and improved between the fuzzing threads and fuzzed processes.



```

/bin/bash
-----[ 0 days 00 hrs 14 mins 00 secs ]-----/ honggfuzz 1.3 /-
Iterations : 398,052 [398.05k]
Mode : Feedback Driven Mode (2/2)
Target : './httpd/httpd -X -f /home/jagger/fuzz/apache/dist/conf/h ...'
Threads : 8, CPUs: 8, CPU%: 261% (32%/CPU)
Speed : 323/sec (avg: 473)
Crashes : 90 (unique: 1, blacklist: 0, verified: 0)
Timeouts : [5 sec] 32
Corpus Size : entries: 1,147, max size: 1,048,792, input dir: 8522 files
Cov Update : 0 days 00 hrs 00 mins 05 secs ago
Coverage : edge: 17,019 pc: 410 cmp: 187,266
----- [ LOGS ] -----

Crash (dup): './SIGABRT.PC.7ffff5ef10bb.STACK.18819c8652.CODE.-6.ADDR.(nil).INST
R.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2018-01-18T22:21:22+0100][W][3343] arch_checkWait():308 Persistent mode: PID 21
623 exited with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent PID: 24520
Crash (dup): './SIGABRT.PC.7ffff5ef10bb.STACK.18819c8652.CODE.-6.ADDR.(nil).INST
R.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2018-01-18T22:21:23+0100][W][3346] arch_checkWait():308 Persistent mode: PID 18
231 exited with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent PID: 25094
Size:296441 (i,b,hw,edge,ip,cmp): 0/0/0/0/0/1, Tot:0/0/0/17019/410/187266

```

Figure 12: Honggfuzz

Basically we can instrumentalize honggfuzz as code-coverage oracle by integrating it into FFW. Honggfuzz plays a role as a middle-ware, it handles the binary server and observes the whole transferring process of protocol. If there are new behaviors, honggfuzz will notify to FFW by using an unique protocol. The detail messages are mentioned in Figure 13.

At the beginning, FFW waits a signal from Honggfuzz for starting fuzzing. Next, FFW reads the corpus, selects randomly a message then mutates it and sends data to target server from the first message to the end. Each time it finishes sending a message to target, FFW also sends a signal to Honggfuzz telling that Honggfuzz should check coverage now. After that, Honggfuzz will inform to FFW he has discovered a new

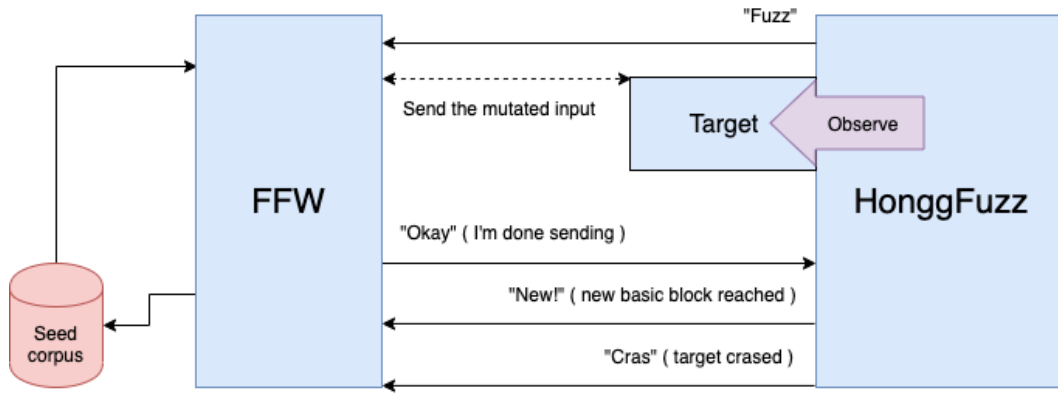


Figure 13: Protocol between FFW and Honggfuzz

region or the target was crashed. From there, FFW will guide the fuzzing process to the final goal.

3.1.3 PULSAR

Pulsar[5] is a method for stateful black-box fuzzing of proprietary network protocols. The method combines concepts from fuzz testing with techniques for automatic protocol reverse engineering and simulation. It proceeds by observing the network traffic of an unknown protocol and inferring a generative model for message formats and protocol states that can not only analyze but also simulate communication. In contrast to other approaches, this model enables effectively exploring the protocol state space during fuzzing and directing the analysis to states which are particularly suitable for fuzz testing. This guided fuzzing allows for uncovering vulnerabilities deep inside the protocol implementation. Moreover, by being part of the communication, Pulsar can increase the coverage of the state space, resulting in less but more effective testing iterations.

The goal of Pulsar is to be able to effectively fuzz the implementation of proprietary protocols for which no specification exists and the underlying code is hard to analyze. In order to achieve this, the method starts by inferring a model of the protocol including its state machine and the format of the messages. The combination of both elements allow to actively control the communication in order to guide the fuzzing process and to build faulty inputs that are sent to the network service. As explained in Figure 14, Pulsar proceeds in the following steps:

- *Model inference*: A sample of network traces from the protocol under test is captured and a model is inferred from its messages. This includes a Markov model representing the state machine of the protocol, templates that identify the format of the messages and rules that track the data flow between messages during communication.

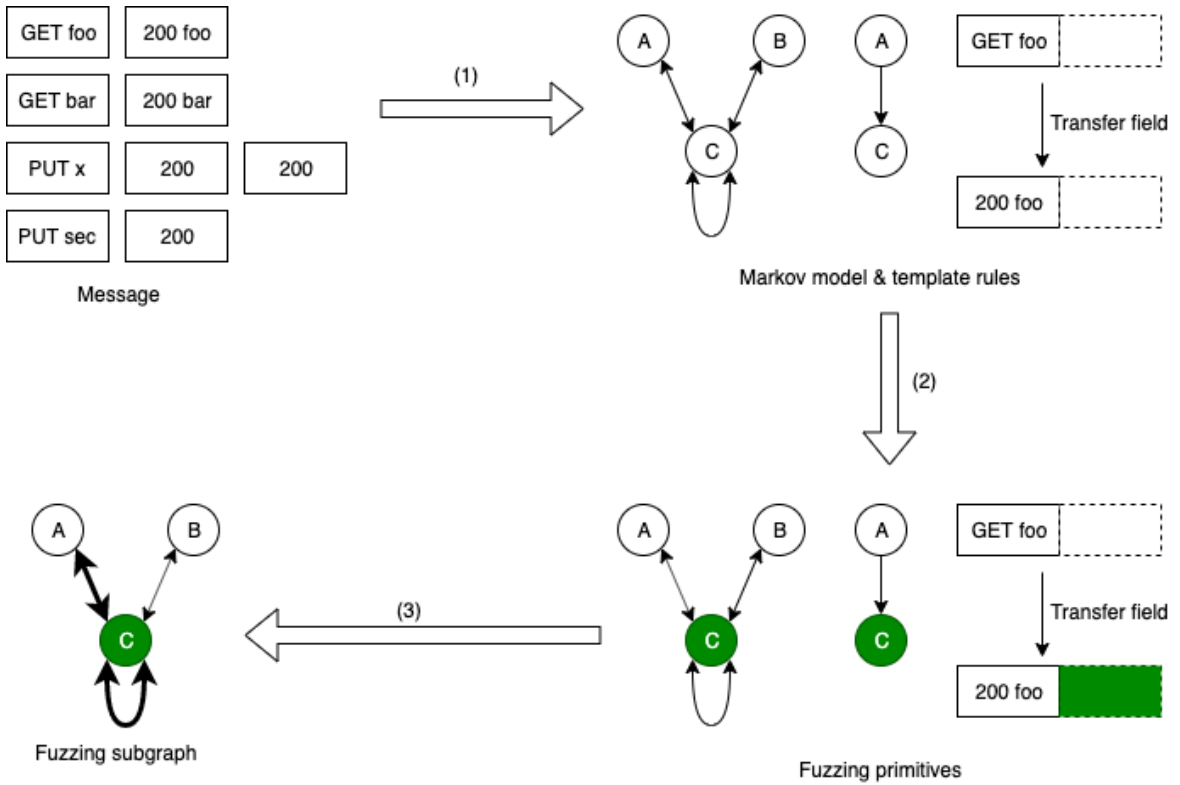


Figure 14: (1): Model interference; (2): Testcase generation; (3): Model coverage

- *Test case generation*: The extracted templates and rules enable defining a set of fuzzing primitives that can be applied to message fields at specific stages of the communication. Using these primitives, test cases for black-box fuzzing are automatically generated.
- *Model Coverage*: To increase the coverage of the security analysis, protocol states that are particularly suitable for fuzzing are selected. To this end, the fuzzer is guided to subgraphs in the state machine that are rarely visited and contain the largest number of messages with variable input fields.

3.2 Problems and solutions

Several methods for locating and eliminating vulnerabilities in protocol implementations have been proposed, each addressing different aspects of the problem. For example, the combination of American Fuzzy Lop and Preeny project brings to the world a new fuzzing tool with super fast speed due to low-level interferences. On the other hands, Fuzzing For Worm shows another comfortable way to test protocols without getting much knowledge of scenario (but truly, it still knows a little information of context because it has mutated single message per iteration), which uses high-level programming language and concept to create links among all components. Pulsar is totally not the same, since it constructs an intelligent model to predict and increase coverage of program by extracting necessary information from the commu-

nication between client and server.

However, current state-of-the-art fuzzers still have some limitation in fuzzing network. For instance, firstly, the original AFL does not support network connection. Besides, normally, a server running protocol has a infinite loop, which helps to handle multiple client's connections, so if we run original AFL, the forkserver will never retrieves results from child process. On the other hands, the mixed project AFL and Preeny requires user to understand deeply how service is running. Then user decides to modify a piece of code to insert AFL's macro and also link to *desock.so* library.

Regarding about FFW, the fuzzer does not force you to do anything, but its speed is very slow because FFW does not have any speedup mechanisms and it is written in Python. Moreover, in fact, elements in this system seem not to be relevant to each others. More specifically, FFW calls Radamsa[9] to randomize inputs, then it feeds to target which was spawned from Honggfuzz and finally Honggfuzz reports the result back to FFW.

For Pulsar, as a drawback of using Markov Chain, or a disadvantage when we compare deterministic and probabilistic method, the result may be not accurate, which affect to the next step for reaching new coverage. The summary is shown in table.

	American Fuzzy Lop and Preeny	Fuzzy For Worm	Pulsar
Speed performance	✓	—	✗
Context knowledge	✗	—	✓
Deterministic coverage	✓	✓	✗
Easy-to-use	✗	✓	—

Table 1: Pros and cons of protocol fuzzers

In this thesis, we choose American Fuzzy Lop as a basement and we present some solutions to eliminate negative side-effect of pure American Fuzzy Lop and Fuzzing For Worm tools in network fuzzing. Our method exploits the feature of Linux operating system and provide a suitable protocol for interprocess communication and synchronizing all tasks together.

3.2.1 Pause / Resume a process in Linux

Processes in Linux follow patterns similar to that of humans. Just like people, processes are born and do carry out regular tasks; taking rest, sleeping in between and finally, being killed or dying. Processes, being the most fundamental aspects in Linux, are necessary to carry out tasks in the system. A process is created by running binary program executable. The binary program executable comes from a piece of code, transitions to a process, while acquiring states during its lifetime and demise.

In a Linux system, the first process to start is the 'init' and it has a PID of 1.

All subsequent processes are `init`'s children, grandchildren and so on. The Linux kernel uses a scheduler and it controls the execution sequence of all processes. Linux processes can have one of four states at any given time: running, waiting or sleeping, stopped and zombie (Figure 15).

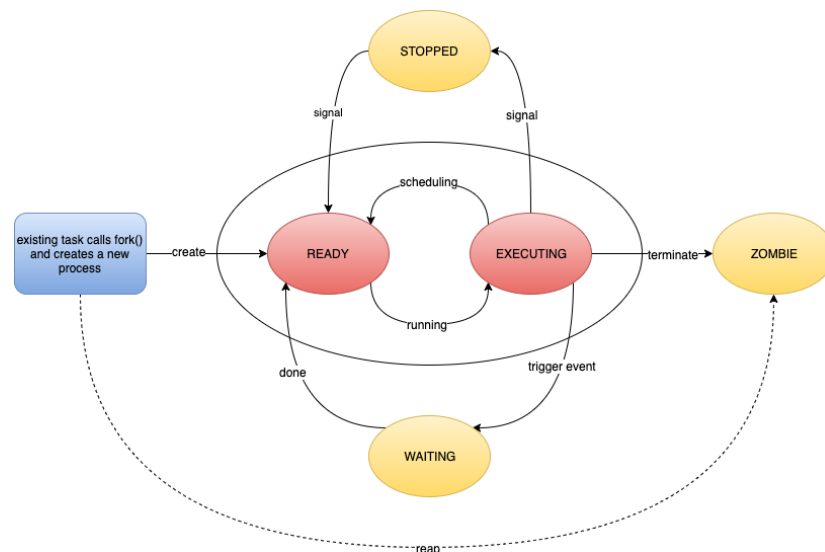


Figure 15: Linux process life cycle

Every processes begin in ready state. In the ready state, a process is awaiting processor time and when the system is not busy, processes don't spend much time in the ready state, they go into the executing state right away. In the executing state, the process is on CPU. There are four ways out of the execute state:

- If a process is interrupted to make way for another ready process it goes back into the ready state.
- If a process is interrupted by a special signal it's halted and goes into the stopped state where it must wait for a user to resume it, which puts it back into the ready state.
- If a process requests I/O operations, like reading data from disk or waiting for a keystroke, it goes into the waiting state. Then it will stay until whatever it's waiting for becomes available
- If a program exits the process goes into the zombie state. In the zombie state the process can no longer execute and waits for it's parent process to acknowledge it's exit.

Therefore, we can take an advantage of "cycle" and the infinite loop to reuse previous process of an existing client's session. By sending special signals (*SIGSTOP* and *SIGCONT*) from parent process to child process, we manipulate AFL to fuzz with current instance of the server again, without initializing anything from the scratch.

This method helps to speedup dramatically number of executions per second, since a server often setups many things before it operates a protocol. However, we must choose wisely where the process should be paused / resumed and when AFL need to clear bitmaps, otherwise, previous results may affect to current coverage. They are the places where a session ends and start, respectively. By doing so, we decide to put those points into functions *close()* and *accept()*, as every network programs call *accept()* for listening a new connection and ends up it with *close()*.

3.2.2 Hooking system APIs

In computer programming, the term hooking covers a range of techniques used to alter or augment the behavior of an operating system, of applications, or of other software components by intercepting function calls or messages or events passed between software components. Code that handles such intercepted function calls, events or messages is called a hook. This thesis will focus on function hooking in linux using the dynamic loader API, which allows us to dynamically load and execute calls from shared libraries on the system at runtime, and allows us to wrap around existing functions by making use of the *LD_PRELOAD* environment variable.

The *LD_PRELOAD* environment variable is used to specify a shared library that is to be loaded first by the loader. Loading our shared library first enables us to intercept function calls and using the dynamic loader API we can bind the originally intended function to a function pointer and pass the original arguments through it, effectively wrapping the function call. In other words, the loader will search the address of function in the library we have passed to *LD_PRELOAD* before finding in system's library, so as soon as loader sees that function *accept()* has been defined, it will no longer look for another *accept()* symbol. However, because we do not want to change the behaviors, we need to keep an original version of those functions. This can be done with a global function pointer and *dlsym()* API.

```
typedef int (*orig_close_f)(int fd);
static orig_close_f orig_close = NULL;
static __attribute__((constructor))
void init_method(void) {
    orig_close = dlsym(RTLD_NEXT, "close");
}

int close(int fd) {
    // Notify to AFL that a session has been ended
    return orig_close(fd);
}
```

However, we can see that we are modifying the default conduct of APIs from system, so we can not guarantee that our target will run normally. In fact, some software wants to build its own network stack, then they usually enable some options for I/O network interface.

3.2.3 Context knowledge

In order to understand partly scenarios of protocols, we provide two modes of running: single message fuzzing and all messages fuzzing. All messages fuzzing means that AFL considers a data in seed pool as an input and mutates the whole input for each round of fuzzing. On the other hands, Single messages fuzzing means that AFL choose randomly a message in the protocol and mutate it, then replace the old message with the new one and then send all of them to target. Both modes have their owns advantages.

There are some protocols which have delimiters to separate messages, for this kind of protocol, we have better to fuzz one message per round. Then we can discover new features of protocol without knowing protocol specification. For example, File Transfer Protocol (FTP) use newlines (`\r\n`) as delimiters to split messages. If we do not know what commands FTP supports, we can use SMF to find new branches of commands automatically. Moreover, this method prevents to mutate overlapping two consecutive messages: by some chances, AFL may modify *HELLO\r\nLOGIN* to *HELxxxxIN*, so obviously it is an invalid command.

In some cases, many protocols, such as MQTT, use an identifier (at the beginning of message) to know which type of command the message is. Then based on the specification, the protocol continues reading remain data. For this situation, we run AMF mode so that it increases the chance to discover vulnerabilities.

3.2.4 Inter-process communication

In order to link all processes together, we use pipes in linux system, which are channels that connect processes for communication. A channel has a write end for writing bytes, and a read end for reading these bytes in FIFO (first in, first out) order. In typical use, one process writes to the channel, and another reads from this same channel. Here is an example:

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ReadEnd 0
#define WriteEnd 1

int main() {
    int pipeFDs[2];
    char buf;
    const char* msg = "Test_communication\n";

    pipe(pipeFDs);
    pid_t cpid = fork();

    if (0 == cpid) { /* ** child ** */
        close(pipeFDs[WriteEnd]);

        while (read(pipeFDs[ReadEnd], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, sizeof(buf));

        close(pipeFDs[ReadEnd]);
        _exit(0);
    } else { /* ** parent ** */
        close(pipeFDs[ReadEnd]);

        write(pipeFDs[WriteEnd], msg, strlen(msg));
        close(pipeFDs[WriteEnd]);

        wait(NULL);
        exit(0);
    }
    return 0;
}
```

The program above uses a system function *fork()* to create a process. Although this program has a single source file, but multi-processing occurs during (successful) execution. Take a deep look into the program:

- If the *fork()* function, called in the parent process, succeeds, it thereby spawns a new child process, returning one value to the parent but a different value to the child. Both the parent and the child process execute the same code that follows the call to *fork*. In particular, a successful call to *fork* returns:
 - Zero to the child process
 - The child's process ID to the parent
- An if/else or equivalent construct typically is used after a successful *fork* call to segregate code meant for the parent from code meant for the child.

If forking a child succeeds, the program proceeds as follows. There is an integer array *pipeFDs*[2] to hold two file descriptors, one for writing to the pipe and another for reading from the pipe. A successful call to the system pipe function, made immediately before the call to *fork*, populates the array with the two file descriptors. The parent and the child now have copies of both file descriptors, but the separation of concerns pattern means that each process requires exactly one of the descriptors.

The first statement in the child if-clause code, therefore, closes the pipe's write end and the first statement in the parent else-clause code closes the pipe's read end. The parent then writes some bytes to the pipe, and the child reads these and echoes them to the standard output.

An aspect of the program needs clarification: the call to the *wait()* function in the parent code. Once spawned, a child process is largely independent of its parent. The child can execute arbitrary code that may have nothing to do with the parent. However, the system does notify the parent through a signal—if and when the child terminates.

Hermes - The next generation of protocol fuzzer

Based on the current state-of-the-art protocol fuzzers, we introduce a new kind of network fuzzing, called **Hermes**. Our PF focuses on testing and finding vulnerabilities on a server which is running protocols.

4.1 Basic concepts and structures

We try to create a stateful fuzzer so we should consider some fundamental concepts. A stateful fuzzer is a testing system which mutates seed corpus based on the state of target. Generally, a stateful fuzzer is not totally different from stateless fuzzer, however, it may have a gap when we look into how fast the fuzzer discovers a new feature of target. Imagine that we have a following protocol (Figure 16), an employee logs in to system to check in his daily work. Nevertheless, this feature is only available to employee, not employer or admin. So if we login as an admin, it can lead to a type confusion vulnerabilities and server will probably be crashed.

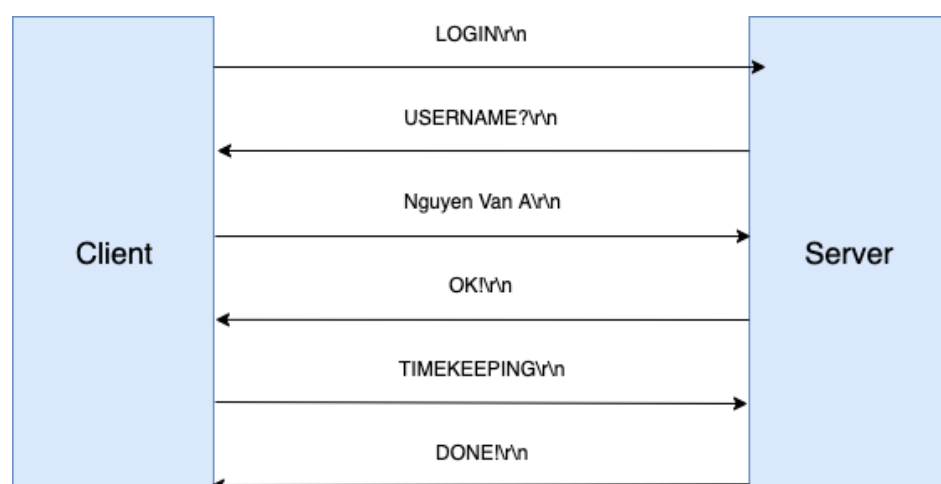


Figure 16: An example of stateful protocol

For a stateless fuzzer, this diagram will conduct to a input as Figure 17a for testing server. As using stateless fuzzers, the whole input will be randomized (Figure 17b) and it takes much much time to reach the meaningful desired input (Figure 17c).

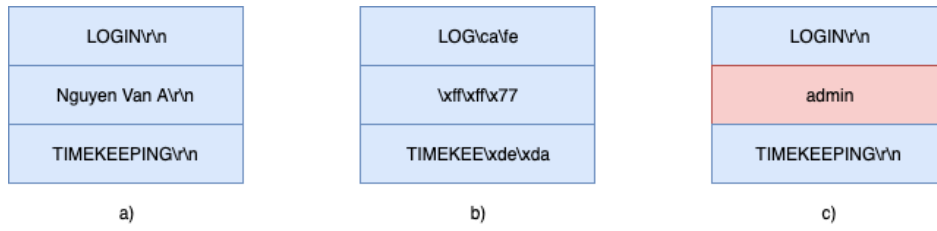


Figure 17: a) Original input b) Normal mutation c) Stateful mutation

Therefore, we address that a protocol is a linked list of some messages, and each message has its own length and next message pointer to control mutation process. A message individually changes something in the context of protocol, which means we do not consider the case of dependent sequence messages. Note that we only perform deterministic algorithm of mutation and trimming input only once, since it is not necessary to duplicate a known process. Moreover, we need to store those information in structure of protocol to manage fuzzing works. This can be done by treating each message as a bit, then after finishing deterministic algorithm or trimming case, we simply flip that bit. We also save information about the previous fuzzed message in order to avoid fuzzing one message two times in a row. In details, two following structures are created:

```

struct message {
    int size;
    int done_det; // Finish deterministic mutation
    int done_trim; // Finish trimming mutation
    char *data;
    struct message *next_msg;
};

struct protocol {
    int number_of_messages;
    int current_fuzzed_message;
    struct message *first_msg;
    unsigned long long int done_det_mask;
    unsigned long long int done_trim_mask;
};

```

4.2 Interceptor

Making seed pool is an important step before fuzzing because it will decide how effective our fuzzing process will be. The more variety of message our input has, the faster our fuzzer finds out new branches. Although our fuzzer's approach is greybox's

fuzzing without knowing protocol specifications, we recommend to have original inputs which are good in both quantity and quality.

In network fuzzing, initial seeds may be different because they were recorded from network. So we have to listen on network interfaces to capture those traffic. Moreover, we need analyse packets then we can serialize all communications to seed corpus. Luckily, *libpcap*[11] helps us to do that instead of writing a driver and running on kernel. *Libpcap* is a system-independent interface for user-level packet capture, and it provides a portable framework for low-level network monitoring.

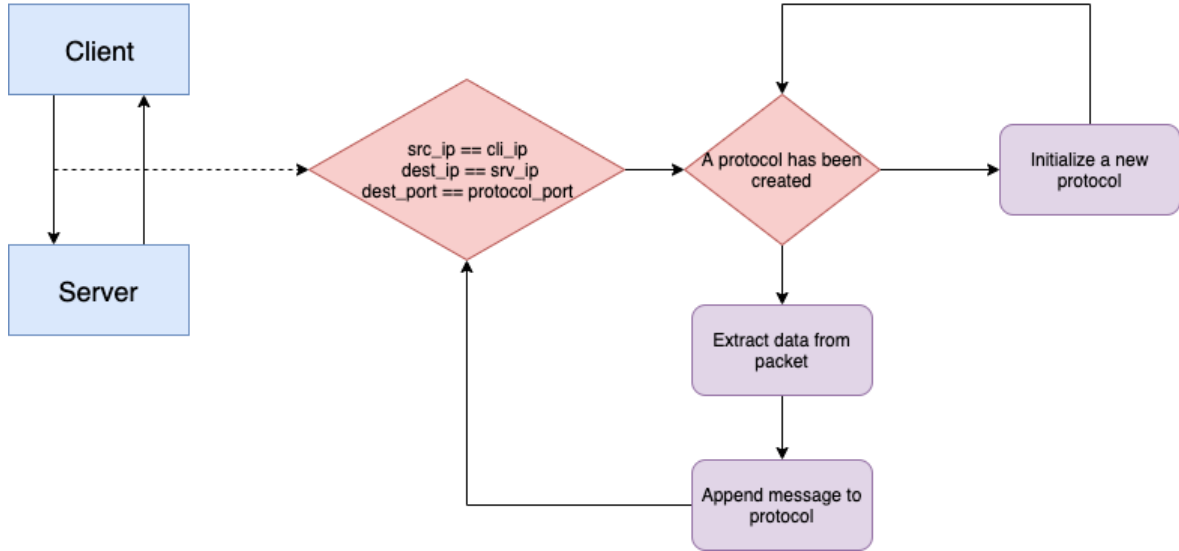


Figure 18: Protocol Fuzzer's interceptor

The model of our interceptor is shown in Figure 18. Assume that both client and server are running in the same local network. We specify the port that protocol is using, and put a listener on "lo" (loopback) network interface. When we capture a packet, we analyse it to see:

- If the source's ip is equal to ip of the client
- If the destination's ip is equal to ip of the server
- If the destination's port is equal to port of the protocol

Then we conclude that packet is a message of protocol, and we will append it to the existed protocol structure we have created. Because the listener event is an infinite loop, so we trap the interceptor to exit normally by making signal handler and hit *Ctrl-C*. In signal handler, we serialize the protocol we recorded into a file and save it as a corpus.

Noted that perhaps the protocol is sent through TCP, UDP or even ICMP, so we have to implement all of them in the handler to capture exactly.

4.3 Compilation and Instrumentation

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux, including the Linux kernel. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

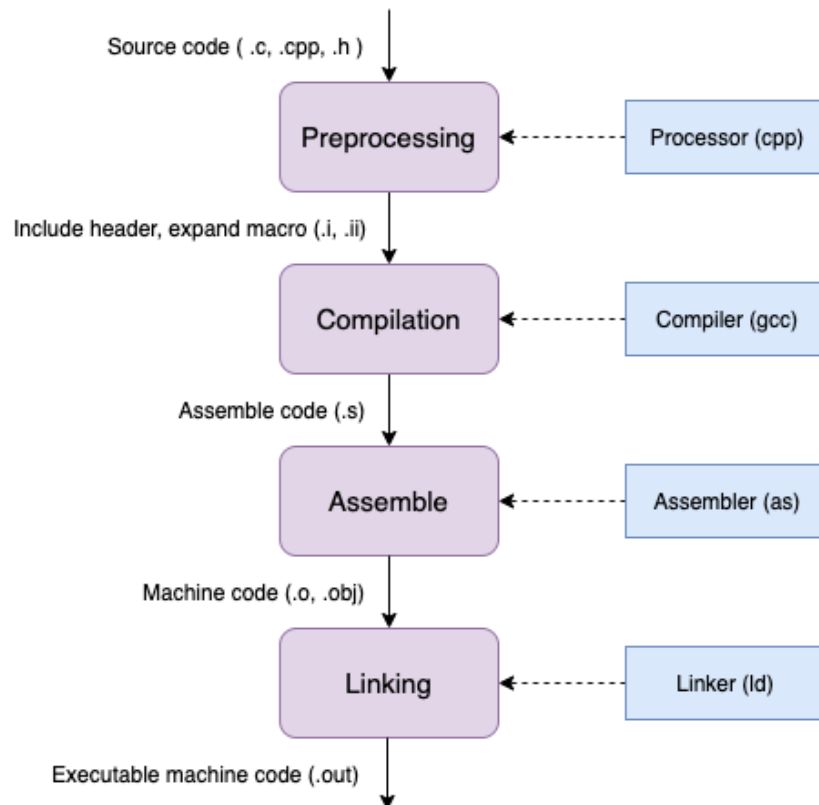


Figure 19: GCC's workflow

According to Figure 19, GCC's workflow has four main steps:

1. **Preprocessing:** This step is done via the GNU C Preprocessor (cpp), which includes the headers and expands the macros. The resultant intermediate file (.i) contains the expanded source code.
2. **Compilation:** The compiler compiles the pre-processed source code into assembly code (.s) for a specific processor.
3. **Assembly:** The assembler (as) converts the assembly code into machine code in the object file (.o).
4. **Linker:** Finally, the linker (ld) links the object code with the library code to produce an executable file (.out)

After second step, we have a meaningful and human-readable assembly code file with full logics of our program. Therefore, this is the best place to put our instrumentation code into. Fortunately, GCC provides an option (-B), which forces the process itself to use a different Assembler rather than using the default. We create a special tool playing a role as wrapper of default Assembler to add instrumentation codes first, and then run the commandline by calling *execvp()*.

As mentioned before, instrumented code is a piece of assembly code right after every conditional jump instructions or before a basic block. Its goal is about updating local bitmap, corresponding to report back to fuzzer the code path of execution.

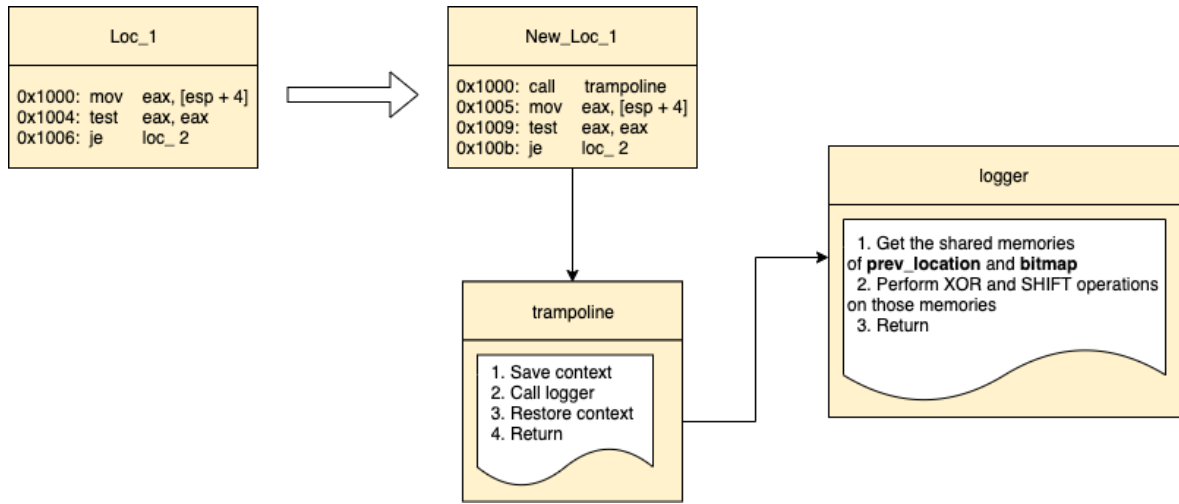


Figure 20: Instrumentation

In order not to change behaviors of program, we have to save context of CPU before running our injected code and restore it after that. CPU's context here includes memory registers and flags and depend on the architecture of CPU, we have methods to back-up state of machine. For example, in Intel-x86 architecture, there is a pair of assembly instruction (*PUSHAD* and *POPAD*) to store registers, flags into stack memory and retrieve those information respectively.

However, rather than using Forkserver mechanism like AFL does, we only use instrumentation to update the code path. After some experiments, we realize that the mechanism does not speed-up much time, because it only runs when the target is terminated. But normally, a server has an infinite loop to serve multiple clients, so it is not helpful to apply Forkserver here, and instead, we choose to implement Pause/Resume mechanism, which will be discussed later. Moreover, due to untermiated process, operator *prev_location* need to be re-initialized every round of fuzzing, so it is necessary to pass pointer's address of this operator through *shmat()* and *shmget()*.

4.4 Interprocess Communications and Hooking System APIs

4.4.1 Architecture of fuzzer

For file-format fuzzing, binary target takes the input from AFL and runs itself, which means there are no external factors interfere to the process. However, for network fuzzing, we need a client to send data to server, and we observe the behaviors of server. Therefore, in Hermes, we create three main components (Figure 21):

- **Main Fuzzer** (parent proccess): this component is our main loop fuzzing. It takes responsibility for mutation, initialization, observation, conclusion.
 - *Mutation*: Main Fuzzer reads corpus from seed pool and modify it. Noted that Hermes runs the same algorithms as AFL does: walking bit and byte flips, performing arithmetics, replacing with known integers, changing randomly and splicing testcase.
 - *Initialization*: Before running or resuming server target, we need to zero-out shared memories, specifically, *prev_location* and *bitmap*. Moreover, alarm signal should also be rewind for tracking timeout process. Then, Main Fuzzer treats the result from Mutation as an input and write to a file.
 - *Observation*: Main Fuzzer runs the server component and waits for returning status code from the server component.
 - *Conclusion*: Hermes evaluates the result to see if server has vulnerabilities or not.
- **Client** (senior child proccess): this component was forked from Main Fuzzer from the beginning. It has an infinite loop, which wait a signal from Server then start connecting and sending data to server.
- **Server** (cadet proccess): this component is the binary target, which was instrumented during compile-time. It is forked or resumed in each round of fuzzing.

Those components communicate to others through pipes and file descriptors. Generally, there are four pipes, each of them has a specific role. We also duplicate those file descriptors into some defined constant so as to use more easily. In particular, we clone eight file descriptors to eight constant number:

- File descriptor for Client to read data from Server: **997**
- File descriptor for Client to write data to Server: **996**
- File descriptor for Server to read data from Client: **995**
- File descriptor for Server to write data to Client: **994**

- File descriptor for Main Fuzzer to read data from Server: **993**
- File descriptor for Main Fuzzer to write data to Client: **992**
- File descriptor for Client to read data from Main Fuzzer: **991**
- File descriptor for Main Fuzzer to read data from Server: **990**

At the beginning, Main Fuzzer calls *fork()* to spawn Client, and from now, there are two parallel processes running together. For each fuzzing round, based on previous iteration, Main Fuzzer decides to fork new Server or resume it. In both cases, Server requires Main Fuzzer to rewind the alarm clock to re-calculate time-out. Then, Server asks Client to clean shared memories and wait its response. Next, Client clears the *bitmap* and set *prev_location* to zero and reply to Server that it has done the job. After that, Client starts reading input from file, which is result from mutation of Main Fuzzer, and sends to Server. Meanwhile, Main Fuzzer call *wait()* to expect returned status code from Server. Eventually, Main Fuzzer tells Client that it should be prepared for new connection.

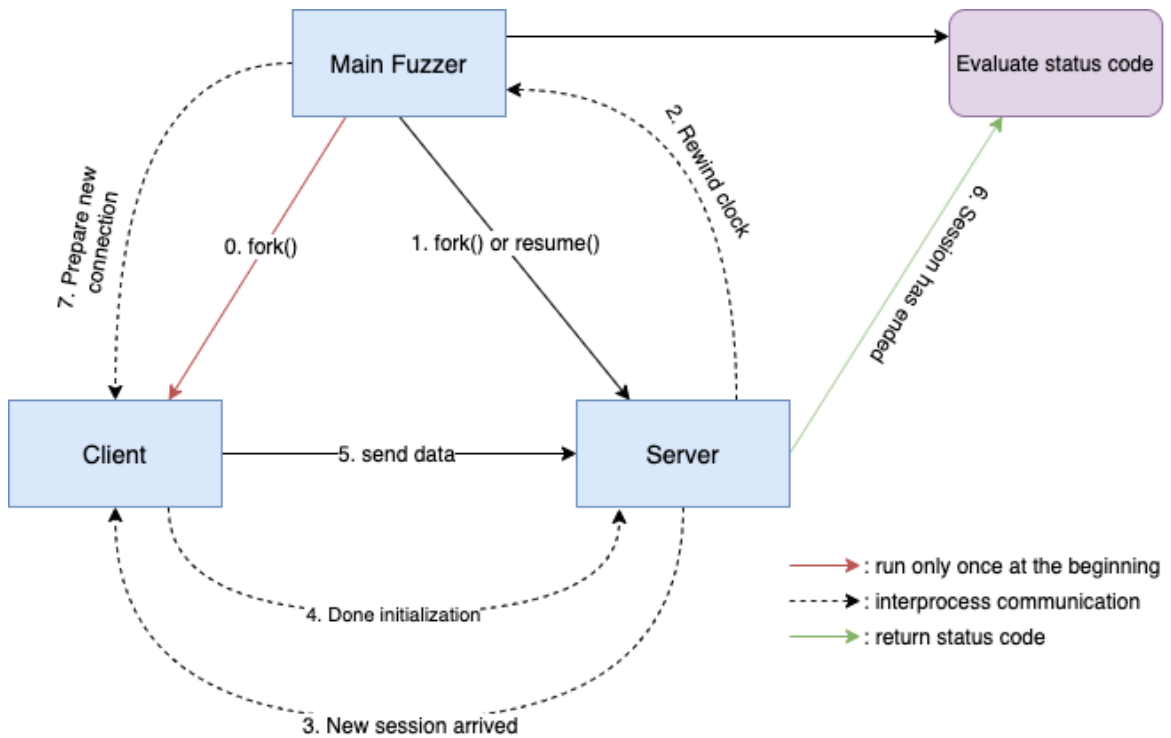


Figure 21: PF's components

4.4.2 Hooking System APIs

ACCEPT()

When Client connects to Server, it means that a new session has been arrived, we should observe the behaviors of Server now. New function *accept()* needs to tell to Client that it must reset the tracing map. This can be done by send something to

Client, because it is blocked now by *read()* function, here we choose to send *pid* of Server:

```
pid_t pid = getpid();
write(994, &pid, sizeof(pid_t))
```

accept() also have to wait a response from Client:

```
read(995, tmp_buf, sizeof(tmp_buf))
```

Main Fuzzer will rewind the clock as soon as it receives a signal from Server. This response actually is the socket file descriptor of the new connection. This helps us to distinguish other file descriptors:

```
write(991, "TIME", 4)
```

Generally, we can hook the original *accept()* function from system's library with this function:

```
int accept(int fd, struct sockaddr *addr, socklen_t *l)
{
    pid_t pid = getpid();
    char tmp_buf[10];

    int result = orig_accept(fd, addr, l);
    write(994, &pid, sizeof(pid_t));
    read(995, tmp_buf, sizeof(tmp_buf));
    write(991, "TIME", 4);

    sprintf(tmp_buf, "%d", result);
    setenv("CLIENT_FD", tmp_buf, 1);

    return result;
}
```

CLOSE()

We should pause Server after the *close()* call is made. However, some program may operate with file on disk, therefore, we only close the socket file descriptor corresponding to current connection. After original *close()* has finished its job, we pause Server by calling *kill()* with *SIGSTOP*. In other words, this is the new *close()* function:

```

int close(int fd)
{
    char *cfd_buf = getenv("CLIENT_FD");
    if (cfd_buf) {
        int client_fd = atoi(cfd_buf);
        if (client_fd == fd) {
            int result = orig_close(fd);
            kill(getpid(), SIGSTOP);
            return result;
        }
    }

    return orig_close(fd);
}

```

RECV()

When Client has sent all data it read from mutated file, it closes the socket on client side immediately. This leads to an error reading next time on server side, and function *recv()* will return -1 value. We can catch this exception and then pause Server (because no more messages would be passed to Server for processing).

```

ssize_t recv(int fd, void *buf, size_t len, int flags)
{
    ssize_t result = orig_recv(fd, buf, len, flags);
    if (result <= 0) {
        unsetenv("CLIENT_FD");
        orig_close(fd);
        kill(getpid(), SIGSTOP);
    }
    return result;
}

```

SEND()

Since our Client does not receive any data (it is unnecessary), we also hook *send()* function to disable this feature.

```

ssize_t send(int fd, const void *buf, size_t len, int f)
{ return len; }

```

SOCKET()

With super speed of fuzzing process, sometimes, we have a problem about sockets and network stack implementation. TCP's primary design goal is to allow reliable data communication in the face of packet loss, packet reordering, and packet duplication.

It's fairly obvious how a TCP/IP network stack deals with all this while the connection is up, but there's an edge case that happens just after the connection closes. What if a program with open sockets on a given IP address + TCP port combo closes its sockets, and then a brief time later, a program comes along and wants to listen on that same IP address and TCP port number?

SO_REUSEADDR is most commonly set in network server programs, since a common usage pattern is to make a configuration change, then be required to restart that program to make the change take effect. Without *SO_REUSEADDR*, the *bind()* call in the restarted program's new instance will fail if there were connections open to the previous instance when we killed it. Those connections will hold the TCP port in the *TIME_WAIT* state for 30-120 seconds.

A hooked *socket()* function should look like this:

```
int socket(int domain , int type , int protocol)
{
    int r;
    r = orig_socket(domain , type , protocol);

    if (r >= 0) {
        int x = 1;
        setsockopt(r , SOL_SOCKET , SO_REUSEADDR , &x , 4);
    }

    return r;
}
```


4.5 Workflow in details

Figure 22 shows the workflow of Protocol Fuzzer. Like AFL, before using Hermes, we need to compile the target with a the mentioned assembler, so that the target will be injected with some instrumentation code. By the way, we combine some hooked functions to a shared library.

Firstly, our PF sets up environment, initializes shared memories and also creates client component. These things are done only once, before going deeper. Client has an infinite loop which always tries to connect to a socket whenever it receives command. However, to reduce the overheat and isolate fuzzing process in each round, after finishing a session, Client waits signals from Main Fuzzer to continue the loop. In this step, both Client and Main Fuzzer create and duplicate some file descriptor, they also close any pipe that is not neccessary. Then Hermes reads data from seeds pool, deserializes and pushes them to queue.

For each fuzzing loop, our PF gets the first test case in the queue, mutates it based on current option (AMF or SMF), re-construct it to raw data and saves into a file (*.cur_input*). During this time, Client still manages to connect to Server until it succeeds. After having desired input, Main Fuzzer calls *fork()* to create an instance of Server and stops before calculating alarm. Noted that Main Fuzzer forks new process with an environment variable *LD_PRELOAD* which is set to the path of our shared library. Target server initializes network, context, ... (if it was forked rather than being resumed) and goes to new function *accept()* to wait a new connection. As a consequence, Client now has been connected successfully and expected notification from Server. Server carries on running the function *accept()* so it contacts to both Main Fuzzer and Client to finish pre-processing jobs.

Next, Client reads the file (*.cur_input*), sends it to Server and closes socket right after this task is done. This action makes Server's function *recv()* return error, which leads to closing socket on server side. Because function *close()* is called, Server is paused immediately due to *SIGSTOP*. This signal returns to *waitpid()* in Main Fuzzer to evaluate as whatever AFL does. Finally, Main Fuzzer tells to Client that it should be prepared for the next test case.

In fuzzing iteration later, Main Fuzzer checks that if Server is able to continue running, then resumes it. Client will connect completely and so on.

4.6 Evaluation

We evaluate our research by fuzzing three cases: simple echo server, complicated chat room and a real-world IoT software which running MQTT protocol. To test with the best performances, we use a powerful machine to run Hermes, FFW and Pulsar:

- CPU is AMD EPYC 7000 series processor that has 16 logical cores running at 2,5 GHz.
- Operating System is Ubuntu 18.04 (64 bit) and has access to 64GB of main memory.
- All instances have the same time budget (24 hours), the same computational resources, and are started with the same seed corpus.

In summary, using Hermes, we have confirmed known vulnerabilities and discovered a new one.

4.6.1 Echo server

An EchoServer is an application that allows a client connects to and then it can send a message to the server. The server receives that message and send, or echo, it back to the client.

Initially, we feed a same seed corpus for every fuzzers, which is "hello". Furthermore, in order to prove that Hermes works well, we put a bug in side the main handler of server: if the message server received is equal to string "exit", then server will be crashed.

```

if (buf[0] == 'e')
    if (buf[1] == 'x')
        if (buf[2] == 'i')
            if (buf[3] == 't')
                return 1 / 0;

```

Table 2 illustrates performances of each network fuzzer.

	Protocol Fuzzer	Fuzzy For Worm	Pulsar
Number of test cases	200M	10M	-
Number of paths	4	4	-
Unique crashes/hangs found	1	1	-

Table 2: EchoServer fuzzinng performances

Pulsar does not work in this case because EchoServer does not have any states, it looks like a mirror that reflects every message sent from client to server.

4.6.2 Chat room

This application is cloned from an arbitrary repository on github [12]. As its name indicates, this is a chat room based on client-server model. There are some commands used to interactive with server or other peers from client.

Command	Parameters	Meaning
/quit		Leave the chatroom
/ping		Test connection, responds with PONG
/topic	[message]	Set the chatroom topic
/nick	[nickname]	Change nickname
/msg	[reference] [message]	Send private message
/list		Show active clients
/help		Show this help
/crash		secret command

Table 3: Server's command

This example is more complicated than EchoServer because now server has a protocol which is built by client's command. We also put a deadcode in server which will be invoked whenever a client who has name "crash" and sends "crash" command to server.

It is interesting that after few minutes running, Hermes has not found the edge case yet, but it discovers another serious security vulnerability. A security bug was found when we create a new nickname:

```

command = strtok ( buff_in , "_" );
if ( ! strcmp ( command , "/nick" ) ) {
    param = strtok ( NULL , "_" );
    if ( param ) {
        char *old_name = _strdup ( cli -> name );
        if ( ! old_name ) {
            puts ( "Cannot _allocate _memory" );
            continue ;
        }
        strcpy ( cli -> name , param );
        free ( old_name );
        send_message_all ( buff_out );
    }
}

```

The program copies parameter which is attached to command by client to a property of an object ($cli \rightarrow name$) on server side. Meanwhile, the *name* field of object

cli is an array of char whose size is 32 bytes. Therefore, obviously, if we send a parameter with size greater than 32 bytes, server will be buffer-overflowed.

Meanwhile, Pulsar now provides better result in number of code paths due to its state discovery methods. Table 4 shows all fuzzer’s performances.

	Protocol Fuzzer	Fuzzy For Worm	Pulsar
Number of test cases	180M	8M	345k
Number of paths	83	20	50
Unique crashes/hangs found	2	1	1

Table 4: Chatroom fuzzing performances

4.6.3 MQTT broker of Mongoose Cesanta

Cesanta is a company based in Dublin, Ireland. Since 2013, they develop and distribute embedded software and hardware with focus on connected products and Internet of Things. Mongoose Embedded Web Server Library has been designed for embedded environments, connecting devices and bringing them online. On the market since 2004, used by vast number of open source and commercial products - it even runs on the International Space station. Mongoose makes embedded network programming fast, robust, and easy.

This library mainly implements MQTT protocol, which is a machine-to-machine (“Internet of Things”) connectivity protocol. It was built as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.

A software running on a computer (running on premise or in the cloud), could be self-built or hosted by a third party. The broker acts as a post office, MQTT does not use the address of the intended recipient but uses the subject line called “Topic”, and anyone who want a copy of that message will subscribe to that topic. Multiple clients can receive the message from a single broker (one to many capability). Similarly, multiple publishers can publish topics to a single subscriber (many to one).

Each client can both produce and receive data by both publishing and subscribing, i.e. the devices can publish sensor data and still be able to receive the configuration information or control commands (MQTT is a bi-directional communication protocol). This helps in both sharing data, managing and controlling devices.

Table 5 contains results of all fuzzers after 24 hours running on the Mongoose library. We found not only some old bugs in previous versions, but also a new bug, which was assigned as CVE-2019-19307.

	Protocol Fuzzer	Fuzzy For Worm	Pulsar
Number of test cases	140M	7M	200k
Number of paths	412	112	97
Unique crashes/hangs found	4	1	0

Table 5: mqtt_broker fuzzing performances

CVE-2019-19307

MQTT protocol uses a *command* and *command acknowledgement* format based on roles such as: Topic names, Client ID, User names and Passwords, ... Payloads excluding MQTT protocol information like Client ID etc is binary data and the content and format is application specific. Its format consists of a 2 byte fixed header (always present) + Variable-header (not always present) + payload (not always present).

The fixed header contains control header and packet length. The latter information's size is between 1 and 4 bytes, and each byte uses 7 bits for the length with the MSB used as a continuation flag. This is where our discovered vulnerability occurs: we can overflow value of remain length, which makes it turns into a negative number.

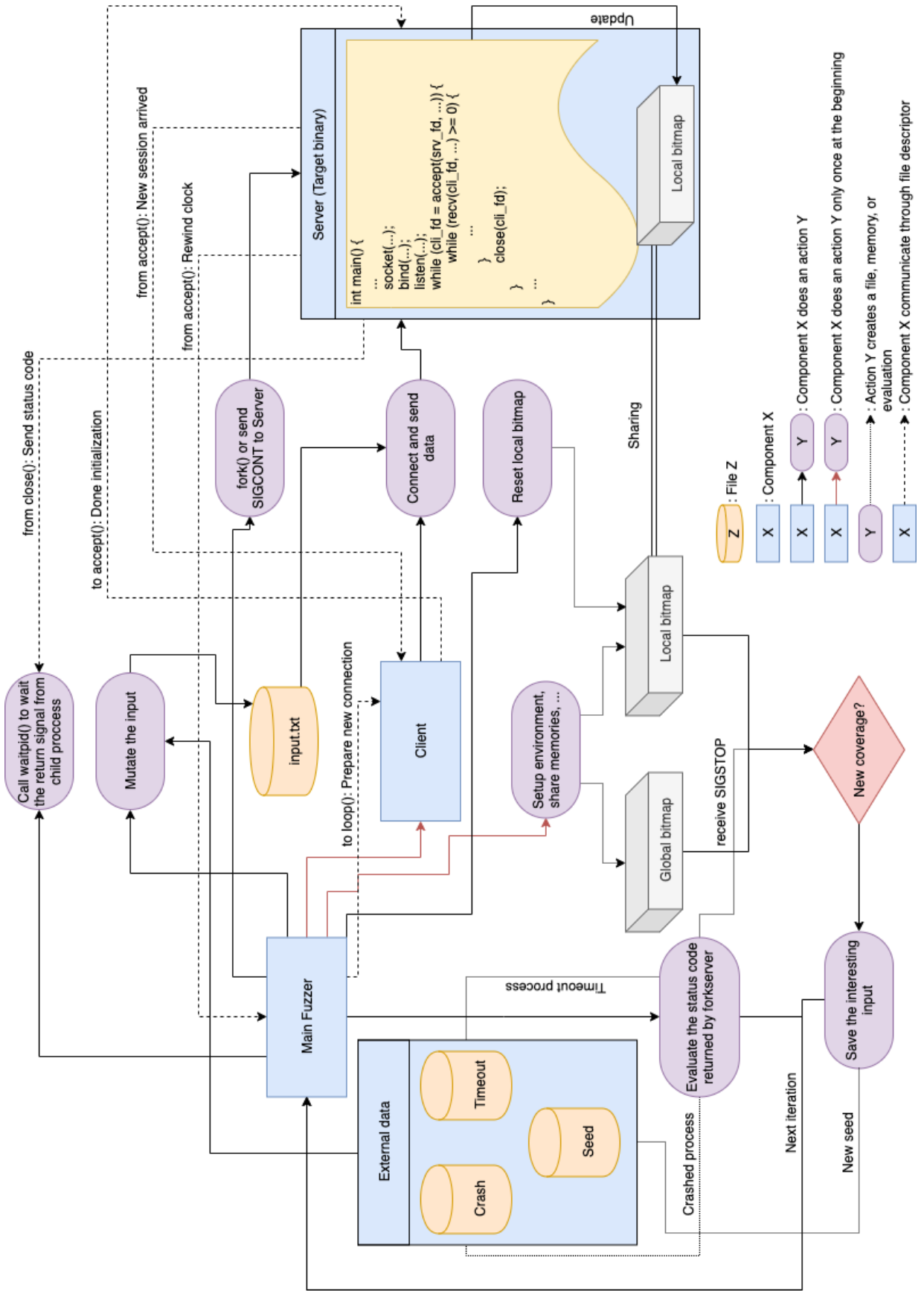


Figure 22: Hermes's workflow

Conclusions and Future work

This chapter consists the thesis contributions, limitations and discuss further development from the proposals presented in above chapters.

5.1 Conclusions

In this thesis, we propose 4 alterations in the architecture of AFL:

- Pause / Resume mechanism
- Hooking system APIs
- Getting context knowledge
- Inter-process communication

Then, we create experiments to evaluate the performance of two proposals and compare these results with current state-of-the-art network fuzzer.

The first proposal in the thesis is using Pause / Resume mechanism of Linux Operating System for reduce time which server consumes to initialize network components. This proposal is the main reason why Hermes give a much better speed test than other fuzzers.

The second proposal helps our fuzzer to determine when it should record and stop processing new fuzzing data. This method increases stability of Hermes, which is the percentage of bitmap bytes that behave consistently during a fuzzing round.

The third proposal supports Protocol Fuzzer to mutate input more effectively. In other words, our fuzzer reaches a new code path with context knowledge.

The last proposal is a way to build network model (client - server) and it is also used to construct a synchronized fuzzing system.

5.2 Thesis contributions

The contribution of this thesis consists numerous aspects listed below:

- Overview of security vulnerability and fuzzer.
- Detail explanation of the architecture and algorithm of AFL, its pros and cons inn network fuzzing.
- Problems of current network fuzzers and some solution to cover those disadvantages.
- The results comparisons and analyses among FFW, Pulsar and our fuzzer.

Our work is also related to Redback project, which was public in some famous conferences.

5.3 Limitations

There are some limitations in Hermes that make it quite difficult to apply our fuzzer in real-world software:

- **Custom network stack:** in some of applications, developers want to change normal behaviors of socket. For example, instead of closeing after a session finishes, they want to re-use that socket for next connections. Therefore, hooking mentioned apis may break the system.
- **Session awareness:** nowadays, developers usually design system that can handle multiple connections without multi-thread by using some system apis like *poll()* or *select()* (they wait for one of a set of file descriptors to become ready to perform I/O operations). So we must pick the place to put notifications carefully.
- **Complete stateful fuzzing:** Imagine that a system requires users to authenticate by username and password, and if user has failed three times in a row, the system would go to "forgot password?" feature, and there is security vulnerability in there. It is really hard to detect that something in context has been changed during processing protocol in this case. Particularly, if we decide to fuzz single message, AFL will never come to that code path because 3 times failed in authentication does not change the bitmap.
- **Encrypted protocol:** Fuzzing encrypted protocol without key is useless because it will always go to failed path during parsing message.

5.4 Future Work

Although Protocol Fuzzer's performance is a multitude of times better than the current state-of-the-art, we meet some problems when scale up to apply widely in real-world systems due to its limitations.

In order to turn all those negatives into positives, we want to try new design patterns that can generalize network stack instead of hooking system apis. Moreover, according to a recent research[13], we can isolate each message to a state, then we can somehow distinguish each context fuzzer discovered completely.

We also want to try a new approach of fuzzing: *topology traversal*. At the beginning, we have empty state, then we start fuzzing the first message which make a new context in the protocol. Fuzzing next message perhaps bring a new context (theory concept from the research[13] can help us to detect duplications). Then by exploring all contexts of protocol, we believe that this fuzzer will give a much more better result.

References

- [1] Michał Zalewski. american fuzzy lop (2.52b).
- [2] J. Barbier J. Bradley and D. Handler. Embracing the internet of everything to capture your share of \$14.4 trillion.
- [3] Sai Kopparthi Maksim Gomov. A survey on fuzzing technique.
- [4] Dobin Rutishauser. Area 41: Fuzzing for worms. 2018.
- [5] Fabian YamaguchiDaniel Hugo Gascon, Christian Wressnegger and ArpKonrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols.
- [6] <https://github.com/oxagast/ansvif/wiki/basic-blackbox-fuzzing>.
- [7] H. Moore. Security flaws in universal plug and play: Unplug. don't play. 2013.
- [8] <https://github.com/zardus/preeny>.
- [9] <https://gitlab.com/akihe/radamsa>.
- [10] <https://github.com/google/honggfuzz>.
- [11] <https://www.tcpdump.org/>.
- [12] <https://github.com/yorickdewid/chat-server>.
- [13] Sergej Schumilo Cornelius Aschermann. Redqueen: Fuzzing with input-to-state correspondence.