Received May 8, 2018, accepted June 17, 2018, date of publication June 28, 2018, date of current version July 25, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2851237

PTfuzz: Guided Fuzzing With Processor Trace Feedback

GEN ZHANG, XU ZHOU, YINGQI LUO, XUGANG WU, AND ERXUE MIN
College of Computer, National University of Defense Technology, Changsha 410073, China

Corresponding author: Xu Zhou (zhouxu@nudt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China (2016YFB0200401), in part by the Program for New Century Excellent Talents in University, in part by the National High-Level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), and in part by the Hunan Province Science Foundation 2017RS3045.

ABSTRACT Greybox fuzzing, such as american fuzzy lop (AFL), is very efficient in finding software vulnerability, which makes it the state-of-the-art fuzzing technology. Greybox fuzzing leverages the branch information collected during program running as feedback to guide choosing seeds. Current greybox fuzzing generally uses two kinds of methods to collect branch information: compile-time instrumentation (AFL) and emulation [AFL extended with QEMU emulation (QAFL)]. Compile-time instrumentation is efficient, but it does not support binary programs. Meanwhile, emulation supports binary programs, but its efficiency is very low. In this paper, we propose a greybox fuzzing approach named PTfuzz, which leverages hardware mechanism (Intel Processor Trace) to collect branch information. Our approach supports binary programs, just like the emulation method, while it gains a comparable performance with the compile-time instrumentation method. Our experiments show that PTfuzz can fuzz the original binary programs without any modification, and we gain a 3× performance improvement compared to QAFL.

INDEX TERMS Feedback, greybox fuzzing, Intel PT, software security.

I. INTRODUCTION

Softwares on computers or smart phones have already become part of our daily life, such as web browsers, players and document processors. However, vulnerabilities in software are still commonplace [1], putting individual users or enterprise users at risk. As a consequence, researches and studies on software security are on a rise.

Symbolic execution and fuzzing are the major two parts of software testing and debugging techniques. Symbolic execution engines [2]–[5] are directly applied to source code and can detecting vulnerabilities effectively. However, symbolic execution triggers a large number of paths in the target program and will result in path explosion. In the contrast, fuzzing avoids the risk of path explosion by exploring possible values of general inputs. And fuzzing techniques can be classified according to the knowledge acquired from program, and they are white-box, grey-box and black-box fuzzing techniques separately. White-box fuzzer can use traditional program analysis techniques to uncover properties of the target, which can be time-consuming. Meanwhile, black-box fuzzer does not have any information of the target program at all. Grey-box fuzzer tries to maintain the simplicity of black-box while improving the effectiveness of fuzzers by adopting additional information.

Moreover, traditional fuzzers are relatively out-of-date in detecting bugs. Without any guidance, blind fuzzers and random mutations of input seed are very unlikely to hit desired branches in programs, leaving some vulnerabilities buried deeply in the execution path, hard to be exposed. Greybox fuzzing is an excellent extension of traditional fuzzing technique. It is an effective fuzzing attempt to analyze programs without much overhead. Greybox fuzzers use lightweight instrumentation or other mechanisms to provide program execution feedback, such as code coverage, for the main fuzzing loop. As mentioned above, initial input seeds are mutated to generate test cases to exercise the program paths. For example, in fuzzers which utilize code coverage feedback, if a certain test case reaches a new path, the fuzzer will retain it as a new input seed for a continuous fuzzing loop, generating even more seeds.

Recent researches on greybox fuzzing technique have attracted much attention. American Fuzzy Lop (AFL) is a pioneer work in this field [6]. It adopts simple but rock-solid compile-time instrumentation to provide code coverage feedback information. Meanwhile, Böhme *et al.* [7] introduced AFLFast. It made improvement on the fuzzing schedule of AFL and exposed 6 CVEs. Furthermore, Rawat *et al.* [8] presented an application-aware fuzzer: VUzzer. And later AFL was added with emulation backend of QEMU to support binary-only fuzzing. Following AFL QEMU mode, Hertz and Newsham [9] proposed TriforceAFL. It uses QEMU full-system emulation to fuzz operating systems.

However, recent works about greybox fuzzing have some common limitations. We can list three drawbacks of previous works as follows:

- **No binary-only fuzzing support.** Greybox fuzzers like AFL, AFLFast and VUzzer all rely on source code of target programs. AFL and AFLFast use bitmap (more information about bitmap is provided in Section II-A) to trace basic block transitions and code coverage. Each basic block has a randomly assigned id from compile-time instrumentation. And this kind of instrumentation CANNOT be done without source code. The same goes for VUzzer, because it analyzes programs with control- and data-flow features from static and dynamic analysis. VUzzer is a typical application-aware fuzzing technique.

  Fuzzers without binary-only support cannot be adopted to situations where compile-time instrumentation or source code is unavailable. Unfortunately, many software vendors prefer not to provide source code of their softwares. This makes fuzzers without binary-only support valueless in detecting vulnerabilities and bugs.
- **Slow feedback mechanism.** In order to solve the problem of dependence on compile-time instrumentation and source code, several feedback mechanisms such as dynamic binary instrumentation (Intel PIN [10]), static rewriting (AFL-dyninst [11]), and emulation (QEMU [12]) are introduced in fuzzing. As we have mentioned, AFL is extended with QEMU emulation (we refer it as QAFL in this paper). Later works such as TriforceAFL, also adopt QEMU to fuzz operating systems. However, Zalewski [13] of AFL pointed out that the usual performance cost of QEMU in fuzzing is 2-5x. The reason for such cost is out of our research scope in this paper. Definitely, such performance overhead due to slow feedback mechanism is intolerable in our fuzzing practice. There is an urgent desire to improve these slow mechanisms.
- **Inaccurate coverage feedback.** As mentioned above, greybox fuzzers like AFL and AFLFast use bitmap to trace basic block transitions and measure code coverage. Each byte of the bitmap represents hit count of a specific edge (e.g. from A to B). The id of block A and B is randomly assigned through run-time instrumentation. A hash value for transition from block A to block B is calculated, used as the key in the bitmap, which is $(A \oplus B)\%BITMAP\_SIZE$ ($\oplus$ means XOR operation). However, hash collision may happen in this scheme. Let us assume that there is a transition edge from A to B and another edge from C to D. When id of A and C is randomly assigned the same value, and B and D is also the same, these two different block transitions will be considered the same, which we call collision or overlap. In this situation, if transition from A to B is not new, and C to D is a new path that has never been hit before, test cases which can trigger transition C to D will not be saved as a new seed. So the fuzzing loop may lose some important seeds, may be incomplete and cannot reach deep paths in programs. Moreover, AFL uses a small bitmap (64KB), so that it can reside in cache to improve performance. The number of edges in a program can be very large, compared to the 64KB bitmap, so the collision ratio may be relatively high. More discussion about branch collision or overlap can be found at Section II-A.

In this paper, to handle the above drawbacks of previous greybox fuzzing techniques, we propose PTfuzz, which is an improved fuzzer guided with Intel Processor Trace feedback. Intel PT [14] is a new feature of Intel processors. It can expose an accurate and detailed trace of program control flow information, such as conditional jump and unconditional jump. Particularly, PT can trace accurate address of every basic block. Thus in PTfuzz, following the idea of AFL, we use this feature of PT to measure transitions between basic blocks, and provide accurate coverage feedback information for the fuzzing loop. Meanwhile, PTfuzz is capable of fuzzing any binary-only softwares, because we directly grab execution information from the processor and do not depend on any source code at all. We further show the performance overhead of PTfuzz in Section V. Our experiments demonstrate that PT is a much faster feedback mechanism than previous works like QAFL and TriforceAFL.

The main contribution of this paper is summarized as follows:

- **Binary-only fuzzing.** We propose a new greybox fuzzer to fuzz any binary-only softwares and do not need any source code. In situations where source code is unavailable, compile-time instrumentation and thorough program analysis is impossible, and fuzzers like AFL, AFLFast and VUzzer will be of no use. Our approach can gracefully handle these situations and fuzz binaries as usual.
- **Fast feedback mechanism.** We introduce a much faster feedback mechanism. As mentioned above, though previous works tried hard to solve the problem of source code reliance, they all suffer from considerable performance overhead, especially QAFL and TriforceAFL. We utilize fast hardware feedback directly from CPU, and deal with binary-only fuzzing in a much faster way than QAFL. The performance overhead of our fuzzer is much smaller than QAFL according to our experiments.
- **Accurate coverage feedback.** We propose a more accurate measurement for code coverage feedback. Compile-time instrumentation and random id assignment of basic

blocks will measure code coverage inaccurately. We use actual run-time addresses of basic blocks to trace transitions between basic blocks and can provide real control flow information of running code.

- **PTfuzz.** We implement a prototype called PTfuzz based on these insights (https://github.com/hunter-ht-2018/ptfuzzer). And our experiments show that PTfuzz can deal with binary-only fuzzing quickly and accurately.

II. BACKGROUND

A. BITMAP

1) DEFINITION OF BITMAP

As mentioned above, greybox fuzzers like AFL, AFL and Syzkaller utilize code coverage feedback to decide whether a test case should be saved as a new seed or discarded. If a test case exercises a new program path, it will be saved as a seed to generate more test cases. But how to decide whether a test case hits a new path or not? There are several choices. Total block (TBL), new block (NBL), total branch (TBR) and new branch (NBR) coverage. (In this paper, branch and basic block transition are the same thing.) *Total* indicates the whole number of hit blocks or branches, and *new* indicates that new blocks or branches are desired. For example, in TBL, if test case $T1$ hits basic block A, and $T2$ hits blocks A and B. Then we can decide that $T2$ exercises a new path. In NBL, if $T3$ hits basic block A, and $T4$ hits blocks B. Then we can decide that $T4$ exercises a new path. However, $T4$ will not be considered as exercising a new path in TBL, because the number of hit blocks of $T3$ and $T4$ is the same. The same goes for TBR and NBR.

Zalewski [6] of AFL conducted experiments on the 4 methods and found out that NBR has the best performance. So in AFL, if a test case hits a new branch, it is considered as exercising a new path and will be saved as a seed. And every hit branch is recorded in a specific memory space called **bitmap**. For example, if there is a branch from A to B, AFL first reads the value in $bitmap[(A \oplus B)\%BITMAP\_SIZE]$ to see if it is 0, and then the value is added by 1. So by simply reading the bitmap, which is initially set to 0s, AFL knows whether a hit branch is new or not and decides to save or discard a test case.

2) WHAT DO VALUES IN BITMAP INDICATE?

Moreover, the number of hit branches recorded in bitmap is a useful indicator to compare code coverage of different fuzzing techniques [15]. In other words, the more hit branches, the higher code coverage. However, Zalewski [6] himself claimed that, in AFL the number of colliding or overlapping branches is 14% when there are 20,00 hit branches in total, and 30% when 50,000. So the indicator can be relatively low for AFL and AFLFast because of collision and overlap of branches. Later experiments in Section V will also demonstrate that AFL has a low value of hit branches. So we can get the conclusion that, fuzzers like AFL and AFLFast have low code coverage because of inaccurate feedback. And they aren't able to expose deeply buried bugs and vulnerabilities in programs.

B. FEEDBACK IN GREYBOX FUZZING

In this section, we will discuss previous greybox fuzzing techniques in detail. We will make classifications according to feedback mechanism adopted in each fuzzer and point out some common limitations. These limitations are the driving force for us to propose our PTfuzz.

TABLE 1. Conclusion of various feedback mechanism.

Feedback Mechanism	Binary-only	Accuracy	Speed	Stability
Compile-time instrumentation	×	×	√√	√
Dynamic binary instrumentation	√	√	×	√
Static rewriting	√	×	√	×
Emulation	√	√	××	√
Intel Branch Trace Store	√	√	√	√
Intel Processor Trace	√	√	√√	√

For greybox fuzzers, built-in feedback mechanism is the key factor to decide fuzzing performance. We conclude various feedback mechanisms in TABLE 1 and discussion on them is as following.

- Compile-time instrumentation is an approach to compile source code and instrument what we need into binaries with special compilers. Previous works such as Vulcan [16], alto [17] and Diablo [18] are leading works in this field. Fuzzers like AFL, AFLFast and Syzkaller [19] adopt compile-time instrumentation to utilize their feedback mechanism. It is relatively fast and stable, but the drawback of compile-time instrumentation is also obvious. As mentioned above, they provide code coverage feedback to the main fuzzing loop by recording transitions between basic blocks. However, the address of a basic block is a randomly assigned value, not the exact address at run time. This is why we claim that it is not an accurate feedback mechanism. Furthermore, source code is needed to compile target programs. So binary-only fuzzing is not available in compile-time instrumentation.
- Dynamic binary instrumentation can analyze the behavior of binaries at run time through instrumentation code. The instrumentation code works as part of the original code after being injected. PIN [10] and DynamoRIO [20] are the most famous works. Utilization of this kind of feedback mechanism in fuzzing can handle binary-only fuzzing and is accurate because

of run-time instrumentation. But the biggest problem of dynamic binary instrumentation is the considerable overhead. It is slower than compile-time instrumentation in fuzzing jobs.

- Static rewriting instruments the binary by inserting callbacks for each basic block and an initialization callback in program entry point [11]. AFL-dyninst uses static rewriting and adopts this technique to fuzz binaries. So static rewriting can handle binary-only fuzzing tasks and can be relatively fast. However, static rewriting has a main drawback that it is unstable and it is fraught with peril [13].
- Emulation like QEMU emulates CPUs through dynamic binary translation, and it is capable of both user-mode and system emulation. And fuzzers are able to run user-level processes and capture program execution flow with the help of QEMU. QAFL and TriforceAFL utilize QEMU and they can fuzz binary-only softwares with accurate run-time feedback. However, due to the architecture of QEMU, these approaches suffer from expensive overhead at 2-5x, compared to no QEMU execution. This is an intolerable overhead to fuzz binaries and cannot be applied to real practice.
- Intel Branch Trace Store is a hardware feature of processors. BTS mechanism enables users to save the branch trace in a specified buffer. It can also provide run-time information without depending on any source code. BTS is similar to PT but setting the BTS flag in CPU can greatly reduce the performance of processors [21]. In other words, we cannot utilize fuzzers with such large execution overhead.
- Intel Processor Trace is also a hardware feature of processors, but it is much more advanced than BTS. PT is capable of accurately tracing program control flow information and we can record basic block transitions based on it. Most importantly, as is shown in Table 1, PT can finish the task without much performance overhead.

In conclusion, fuzzers extended with Intel Processor Trace feedback is able to complete the task of accurately recording basic block transitions in binary-only fuzzing. Moreover, it can be very fast. Based on PT, our PTfuzz successfully handles the problems of previous works.

C. INTEL PROCESSOR TRACE

In this section, we will briefly introduce Intel Processor Trace. With Broadwell architecture and new generation of Core processors, Intel has proposed a new hardware feature called Processor Trace [14], which is an extension of Intel Architecture that captures tracing data about program execution. Intel PT will cause only minimal performance overhead to the program being traced with well-designed hardware facilities. Previous hardware features such as Intel Last Branch Record also performs program tracing, but its output is stored in special registers, instead of main memory. Intel PT utilizes memory space to store tracing data, so continuous PT tracing is only limited to the size of main memory. Thus we are able to perform continuous fuzzing jobs and trace program execution with this output feature of PT.

Specifically, the output of PT is collected in the format of data packets [21]. Packets can be classified into 2 types according to their functionality: basic execution information packet and control flow information packet. Packet Stream Boundary (PSB), Time-Stamp Counter (TSC) and other relevant packets are basic execution information, which demonstrate general program running status. And in initial implementation of PT, control flow tracing packets are provided to be processed by a software decoder. Control flow information includes time, program flow and other information during run time.

Basic block is a continuous code section with no jumps or branches. In this paper, in order to capture transitions between basic blocks, we need to concentrate on program flow information, such as jump targets, and branch taken/not taken. And we need a specially designed decoder to decode program flow tracing data in memory space. Intel PT specifies instructions that can change program flow as Change of Flow Instructions (COFI). Three types of COFI instructions are introduced: Direct transfer COFI, Indirect transfer COFI and Far transfer COFI.

Moreover, Intel PT introduces 4 specific packets to trace COFI instructions:

- Taken Not-Taken (TNT) packet. A specific bit in TNT packet can indicate whether a branch is taken or not in conditional jumps. So it is used to trace direction of conditional branches.
- Target IP (TIP) packet. TIP records the target instruction pointer (IP) of jump or transfer instructions. IP value is stored in specific bits in this packet. In detail, TIP can be classified into TIP, TIP.PGE, TIP.PGD and TIP.FUP according to different application scenarios.
- Flow Update Packet (FUP). When asynchronous events such as interrupts or traps happen, we need FUP to provide source IP addresses, because TIP is out of function in these events.
- MODE packet. MODE provides important program execution information and it has a wide range of formats to indicate the execution mode.

So with the help of TNT, TIP and FUP packets, we are able to write a decoder to capture basic block transitions in program execution. More specifically, we need to record addresses of basic blocks. And when COFI instruction take place, we need to record control flow transitions between basic blocks in the bitmap. For example, when there is an indirect jump instruction (e.g. JMP (FF/4), CALL (FF/2)) from A to B, decoder can recognize this belongs to TIP packets and read the IP address in it. Then *bitmap* $[(A \oplus B)\%BITMAP\_SIZE]$ is incremented by 1 and we complete recording a transition.

D. SYMBOLIC EXECUTION AND FUZZING

Vulnerabilities in software are still common place, putting individual users or enterprise users at risk. In order to detect

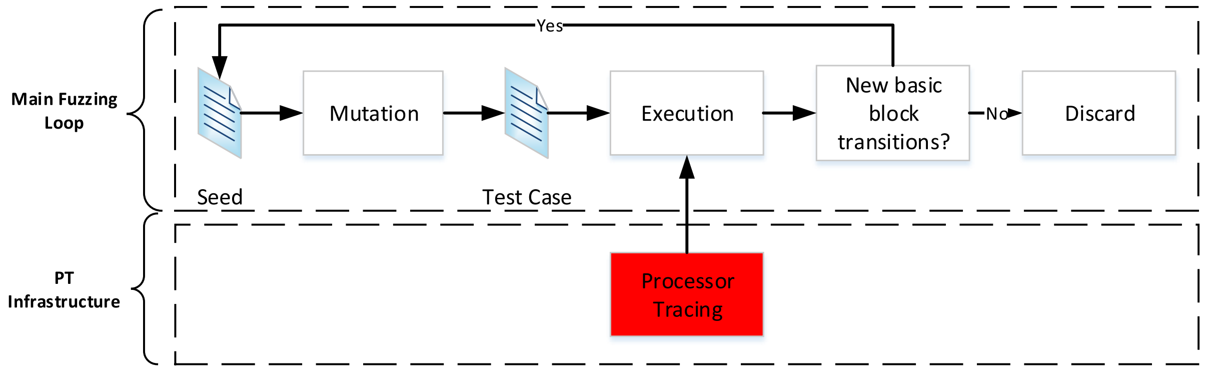

FIGURE 1. High level overview of our model.

vulnerabilities and bugs in softwares, we can examine the source code and match a certain known pattern, and this is a static analysis approach. However, many classes of vulnerabilities such as functional correctness bugs, are difficult to find without executing a piece of code. And exposing bugs with code execution is called dynamic approach. With regard to the problem of code executing, there has been much debate about symbolic execution versus more lightweight fuzzing technique [7].

Symbolic execution and fuzzing are the major two parts of software testing and debugging techniques.

Symbolic execution engines can be directly applied to source code. They use program analysis to interpret an application, model user input with symbolic variables, track constraints produced by conditional jumps and adopt constraint solving to create interesting inputs to cover specific program paths. Symbolic execution engines such as Veritesting [22], Firmalice [23], under-constrained execution [24] catch wide attention these days. Meanwhile, with the improvement of computing power recently, concolic execution (also known as dynamic symbolic execution) has risen in popularity. And there comes plenty of tools with high performance: EXE [2], KLEE [3], Mayhem [4] and S2E [5]. Despite their promising characteristics, both symbolic concolic execution suffer from the well-known path explosion problem due to the inner working structure of symbolic execution.

Moreover, symbolic execution tools are able to automatically analyze programs with symbolic values and a huge amount of constraint solving [3]. It triggers a large number of paths in the target program and will result in path explosion. Meanwhile fuzzing is an efficient testing technique to expose bugs and vulnerabilities. Given an initial input seed, a bunch of new test cases can be generated by simple mutations, in order to exercise and cover as much program paths as possible. Today most vulnerabilities are exposed by particularly lightweight fuzzers that do not leverage intensive program analysis [6].

In the contrast, fuzzing avoids the risk of path explosion by exploring possible values of general inputs, trying to catch specific values and drive execution flow between program compartments. Existing fuzzers such as Dowsing [25], BROG [26], Flayer [27] and BuzzFuzz [28], lack precise information makes fuzzing unable to solve different conditional jumps and deeper execution in programs. Combining both symbolic execution and fuzzing, Driller [1] proposed a hybrid approach to solve the above issues.

Meanwhile, fuzzing techniques can be classified according to the knowledge acquired from program. Typically, white-box fuzzer have full information of the target program and can use traditional program analysis techniques to uncover properties of the target. White-box fuzzers including SmartFuzz [29], BuzzFuzz [28], Vuzzer [8] and TaintScope [30] achieve expected performance and can be applied to real-world scenarios. Grey-box fuzzer uses specific feedback information to enhancing the process of "blind" fuzzing. This kind of fuzzing tries to maintain the simplicity of black-box while improving the effectiveness of fuzzers by adopting additional information. AFL [31] and AFLFast [7] are the most successful representation of grey-box fuzzers. Meanwhile, black-box fuzzer does not have any information of the target program at all. Recently new ideas are put into black-box fuzzers and Radamsa [32], zzuf [33] and peach [34] did remarkable work in this field.

This kind of classification of fuzzers are based on the interaction with the target program, while it can also be classified into non-kernel and kernel fuzzers depending on whether it can be used to fuzz kernels [19].

III. MODEL OVERVIEW

In this section, we will introduce our model and the fuzzing steps of PTfuzz. As a greybox fuzzer, PTfuzz follows the fuzzing architecture of AFL. It starts with initial seed files and tries to generate new seeds to exercise as much as program paths as possible. As shown in Fig. 1, PTfuzz mainly contains two relevant parts: the main fuzzing loop and the PT infrastructure.

The main fuzzing loop works as a parent thread and its jobs include:

- Configuration. This part includes screen display initialization, locating input seed files, signal setup, timer setup and other configurations. This configuration step is essential for later fuzzing steps.

- Pre-build COFI map and write MSR registers. The target binary will be loaded and instructions are dumped to construct COFI map mentioned in Section II-C for decoding purpose. And specific MSR registers are written to set up ip filtering for PT process.
- Load seeds. Input seeds are listed sequentially in a queue. Every time when execution of last seed is finished, next seed in the queue is automatically loaded for mutation.
- Mutation. Mutation means changing certain parts of a seed to generate several test cases. And it can be classified into deterministic and non-deterministic mutation. Deterministic mutation includes: sequential bit flips with different lengths and stepovers, adding or subtracting small numbers and inserting known integers. And non-deterministic mutation includes stacked bit flips, inserting, deleting and splicing. For example, there is a seed and its content is "10011". A program parser needs to read "10010" to execute following code. Obviously, this seed cannot pass the program parser. When bit flip on the first bit is performed, this seed turns into a test case "10010". And this test case passes the program parser and following code is executed.
- Execution. After mutations or test cases are generated, they are taken as input for program execution.
- Fork. At the beginning of execution, a child thread is called up to perform as Processor Trace infrastructure by *fork*(). And this child thread is reaped after execution is finished, and decoded tracing data will be sent to the parent thread.
- Save or discard. After execution of a test case is over, the main fuzzing loop needs to decide whether to save or discard this one based on the information provided by the PT infrastructure. Saved test cases will be put into the queue of seeds for continuous fuzzing.
- Display and report. When fuzzing is running, screen display is necessary to monitor the fuzzing status. Additionally, after fuzzing is over, a detailed report about all kinds of running information is created by the main fuzzing loop.

As mentioned above, the Processor Trace infrastructure is created by the main fuzzing loop through fork(). And it mainly completes these tasks:

- Enable PT. In order to perform continuous tracing of program execution, Processor Trace needs to be enabled. After fork() operation, the PT infrastructure enables PT at the beginning of program execution.
- Record tracing data. After PT is enabled, it will captures program execution information non-stop. And the PT infrastructure specifies a certain memory space to store tracing data after enabling PT. Thus PT can write tracing data in this specific space.
- Decode tracing data. Our purpose is to find basic block transitions in program execution, so the PT infrastructure decodes the raw tracing data and captures TNT, TIP and FUP packets as described in Section II-C and other relevant information. And basic block transitions are written into the bitmap so the main fuzzing loop can access it and make decisions.
- Disable PT. After execution is over, PT is disabled and nothing will be recorded by PT.

And then we will describe running steps of our model in detail. As shown in Fig. 2, specifically, PTfuzz works as follows:

- (1) The main fuzzing loop configures screen display initialization, locating input seed files, signal setup, timer setup and other related events.
- (2) Target binary is loaded into memory and instructions in text section are dumped to build COFI map. And MSR registers in CPU are modified that ip filtering is set up.
- (3) One seed in the seed queue is loaded, ready for mutation.
- (4) This seed is given to mutation engine to perform deterministic and non-deterministic mutations, generating several test cases.
- (5) Test cases from mutation are taken as input into the target program for execution one by one.
- (6) At the beginning of execution, the main fuzzing loop calls up a child thread by fork(), so Processor Trace infrastructure is alive.
- (7) PT infrastructure enables PT right after it is forked.
- (8) PT tracing data are recorded into a specific memory space.
- (9) The tracing data is decoded to expose basic block transitions and bitmap is updated so the main fuzzing loop can access it to make decisions.
- (10) PT is disabled after execution is finished.
- (11) This child process is reaped so PT infrastructure is not alive.
- (12) If there is a new basic block transition triggered by a test case according to PT feedback, the main fuzzing loop will save this particular test case and put it in the queue of seeds; if no new transition is detected, this test case will be discarded.

After step (12), the main fuzzing loop loads the next seed in the queue and starts next loop of fuzzing. PTfuzz requires user provided initial seeds to start with. And we won't discuss how to select initial seeds in this paper because it is partly out of our research scope. Moreover, PTfuzz can report program crashes, execution speed and number of hit branches in time. At last, PTfuzz ends with user's halt operation or pre-set running time.

IV. IMPLEMENTATION

Based on our model overview discussed in Section III, we implement our prototype PTfuzz. We will describe some details about our implementation in the following.

- Building COFI map. As described in Section II-C, Change of Flow Instruction (COFI) is control flow sequence in Processor Trace. When target program is

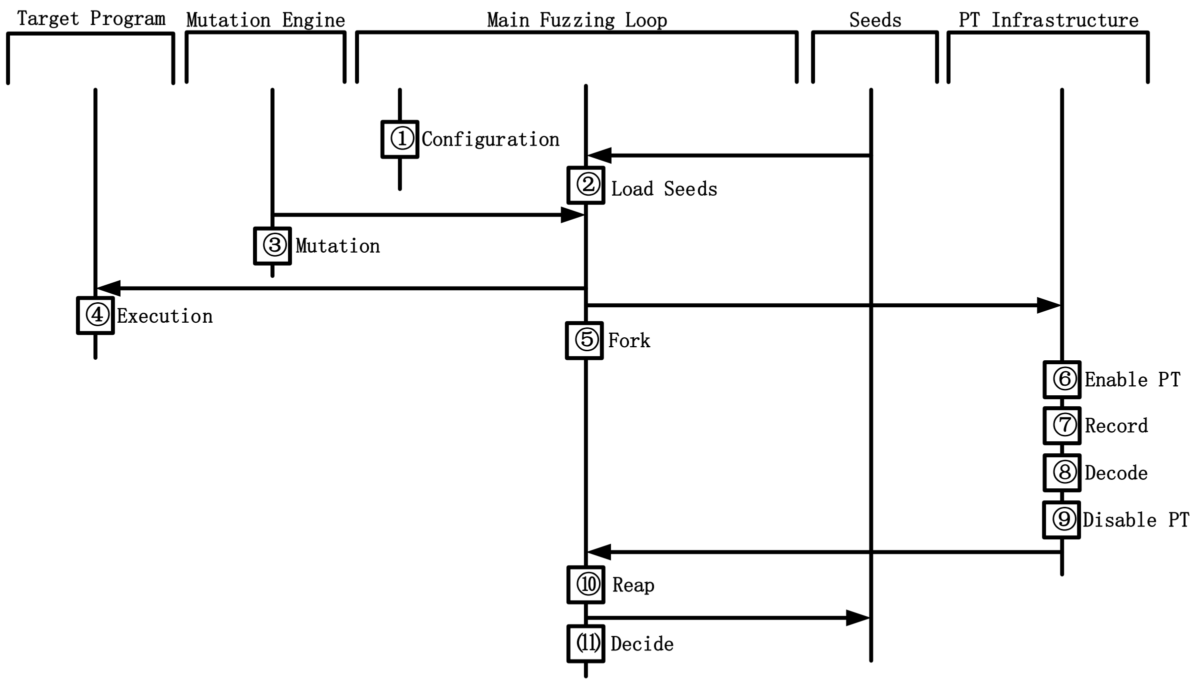

FIGURE 2. **Detail steps of our model.**

loaded, we can dump it to extract all the instructions in text section and the range of ip address. In our implementation, we adopt *Python* – *CLE*[1] to load the binaries and *capstone*[2] to dump the text section. Decoding PT trace packets requires COFI map, and by pre-building it, PTfuzz can save considerable decoding time and greatly reduce execution overhead.

- Writing MSRs and setting up ip filtering. As described in Intel's programming guide [21], we can filter PT packets generation by setting up certain MSR registers. In PTfuzz, we only need to decode PT packets in the ip range of the target binary, not including some library or system calls. So we follow *msr* – *tools* [3] to write MSR registers and limit PT packets generation to the target binary. By doing this, the decoding of PTfuzz can escape a huge number of irrelevant PT packets and gain a distinct performance improvement.
- Enabling and disabling PT in fuzzing. As described in Intel's System Programming Guide, we have to set a specific bit of model specific register (MSR) to 1 to enable Intel PT [21]. After enabling PT, it will trace any execution information in CPUs. However, setting MSR cannot be done directly by user-level compartments. So in PTfuzz, we utilize system call *ioctl*() to perform this operation of setting MSR. In details, the main fuzzing loop will call up a child thread to enable PT with ioctl() and start tracing program execution. The child thread will then disable PT by ioctl() after fuzzing iteration is over.
- Processor Trace output. Previous works about PT such as Simple-PT [35], cannot perform continuous tracing because they store tracing information in files. Writing files could be slow and is not capable of in-time interaction with the main fuzzing loop. So in PTfuzz we use *mmap_page* feature in perf_event [36]. This feature configures a specific memory space to store tracing data. And PTfuzz records tracing information in this memory. Accessing memory can be much faster than files and interaction with the main fuzzing loop is timely.
- Decoding PT information. After PT information is recorded in memory, we need to decode it to find basic block transitions. And Intel has proposed its own PT Decoder Library [37]. However, it is a general purpose decoder and does not fit our demand well, because it does not provide the API to trace basic block transitions. So we write a new decoder for PTfuzz based on the System Programming Guide [21]. Our decoder is a special purpose one that only concentrates on basic block transitions in program execution. It accesses information in our specified memory space and decodes tracing data to find what the main fuzzing loop needs.

V. EXPERIMENT

In this section, we will discuss experiment results of our PTfuzz compared to AFL and QAFL. AFL is an extraordinary work in application-aware fuzzers with compile-time instrumentation. Comparison to it will demonstrate that PTfuzz is

[1]https://github.com/angr/cle
[2]http://www.capstone-engine.org/
[3]https://01.org/zh/msr-tools?langredirect=1

capable of binary-only fuzzing and has a relatively higher code coverage than AFL. And QAFL represents fuzzing techniques with binary-only support. By comparing to QAFL, we can show that PTfuzz is relatively faster in fuzzing tasks.

TABLE 2. Input parameter for target programs.

Program	Parameter
cxxfilt	FILE
nm	-C FILE
objdump	-d FILE
readelf	-a FILE
size	FILE
strings	FILE
gif2tiff	FILE out.tif
tiffinfo	FILE
mpg321	FILE
tcpdump	-r FILE
base64	-d FILE
md5sum	-c FILE
uniq	FILE
who	FILE

Experimental Setup: The experiments are conducted on a desktop with Intel Core i7 3.4GHz 8 Core CPU and 8GB RAM running Ubuntu 16.04. And our target programs are chosen intentionally. We have *cxxfilt*, *nm*, *objdump*, *readelf*, *size* and *strings* from GNU Binutils [38] and *base*64, *md*5*sum*, *uniq*, *who* from LAVA-M data set. Binutils are a widely used collection of binary tools. And we have image processing tools *gif* 2*tiff* and *tiffinfo* from TIFF [39]. Moreover, *mpg*321 [40] and *tcpdump* [40] are included. Thus we have 10 target programs to conduct experiments in total. And TABLE 2 shows our input parameter for these programs. For each of the programs, we individually fuzz it with PTfuzz, AFL and QAFL for 24 hours. Moreover, we record 3 indicators for each experiment:

- Crashes. This is the number of unique crashes when executing the programs. And crashes result from unique test cases that cause the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT). This is a widely-used indicator in [7], [8], [41], and [42] to decide whether a fuzzer has good fuzzing performance or not.
- Speed. We measure execution speed of each fuzzer in **exe/s** to demonstrate fuzzing overhead. This indicator means the number of executed test cases each second.
- Branches. As mentioned in Section II-A, the number of hit branches (basic block transitions) is an important indicator to measure code coverage for fuzzers. The more branches, the higher code coverage. And covering more code of programs will definitely lead to more deeply buried vulnerabilities and bugs.

And experiment results and detailed discussion are as following.

A. SOURCE CODE AVAILABLE FUZZING

In this section, we will compare the experiment results for PTfuzz and AFL in details. As discussed above, before we can fuzz our programs with AFL, we have to compile the programs with AFL's own special compiler called *afl-clang-fast*. The compiler instruments randomly assigned values as addresses for basic blocks. But our PTfuzz doesn't need to go through these steps. So the 10 programs are instrumented-programs when running on AFL, and not on PTfuzz.

TABLE 3. Source code available fuzzing.

	PTfuzz			AFL		
Program	**Crashes**	**Speed**	**Branches**	**Crashes**	**Speed**	**Branches**
cxxfilt	9	120/s	17428	8	125/s	14194
nm	17	703/s	32137	14	736/s	28012
objdump	1	305/s	36980	2	332/s	27511
readelf	0	391/s	26898	0	403/s	19891
size	2	499/s	31699	1	536/s	28577
strings	0	139/s	1136	0	150/s	908
gif2tiff	7	702/s	21366	**Odd, check syntax**		
tiffinfo	141	398/s	11053	127	415/s	8107
mpg321	5	295/s	7476	3	318/s	5258
tcpdump	17	451/s	39587	12	484/s	31850

As shown in TABLE 3, PTfuzz finds out more unique crashes than AFL in 7 out of 10 target programs (cxxfilt, nm, size, gif2tiff, tiffinfo, mpg321 and tcpdump), and the same number of crashes in 2 programs (readelf and strings). However, as for executed test cases per second, the speed of PTfuzz is about 7% slower than AFL and this is due to our implementation of PT decoder. (We discuss this problem in detail and provide future work about this in Section VI.) But an overhead of 7% is relatively acceptable in fuzzing tasks. In hit branches, PTfuzz has more branches than AFL in all the 10 programs. And this indicator also demonstrates that PTfuzz can cover more code and paths in programs and is able to expose deeper bugs.

Moreover, when the input files and parameter are all the same with PTfuzz, AFL is not able to normally fuzz gif2tiff and only results in "**Odd, check syntax**" on the screen. According to documents of AFL [43], this information indicates that fuzzing in AFL goes wrong and we should stop fuzzing immediately. We assume that this problem is due to AFL's instrumentation when compiling gif2tiff. AFL's special compiler afl-clang-fast may destroy some internal features of the program and we are not able to fuzz it with

TABLE 4. Binary-only fuzzing.

	PTfuzz			AFL	QAFL		
Program	**Crashes**	**Speed**	**Branches**	**No binary-only fuzzing**	**Crashes**	**Speed**	**Branches**
cxxfilt	9	120/s	17428	×	8	21/s	15107
nm	17	703/s	32137	×	10	412/s	21927
objdump	1	305/s	36980	×	0	85/s	24608
readelf	0	391/s	26898	×	0	16/s	20296
size	2	499/s	31699	×	0	80/s	27542
strings	0	139/s	1136	×	0	9/s	602
gif2tiff	7	702/s	21366	×	3	546/s	22620
tiffinfo	141	398/s	11053	×	85	150/s	7340
mpg321	5	295/s	7476	×	**Gets stuck after fork server is up**		
tcpdump	17	451/s	39587	×	**Gets stuck after fork server is up**		

AFL. More importantly, this is a solid prove that AFL cannot be used in every source code available fuzzing tasks, but our PTfuzz can handle this problem well.

Comparison between PTfuzz and AFL is a strong evidence that PTfuzz is able to discover more program crashes, achieve higher code coverage and is more effective in exposing vulnerabilities. The primary reason for this result is that we overcome the internal drawback of AFL's compile-time instrumentation and utilize actual run-time addresses for basic blocks, instead of randomly assigned values. And PTfuzz is capable of accurately capturing basic block transitions in program execution and providing useful feedback for the main fuzzing loop. In conclusion, PTfuzz outperforms AFL in crash-exposing ability and code coverage, with a 7% overhead, which can be eliminated with code optimization.

B. BINARY-ONLY FUZZING

In this section, we conduct experiments on binaries of the 10 programs in PTfuzz, AFL and QAFL, without source code. And we still list 3 indicators, crashes, speed and branches, in TABLE 4. These binaries are compiled with ordinary gcc compiler by other computers and we only have 10 binaries in our desktop, without any source code.

First, we will look into the comparison between PTfuzz and AFL in this table. It is not the same as experiments when source code is available because we can't compile the 10 programs with AFL's special compiler at all. As we can see, AFL cannot do anything in these binary-only fuzzing jobs and it is valueless when source code is unavailable. However, our PTfuzz can fuzz all these binaries without exception. In binary-only fuzzing scenarios like this, PTfuzz definitely outperforms AFL.

Then we will examine the comparison between PTfuzz and QAFL. Generally, PTfuzz exposes more crashes than QAFL in 8 out of 10 programs (cxxfilt, nm, objdump, size, gif2tiff, tiffinfo, mpg321 and tcpdump), and the same crashes in 2 programs (readelf and strings). As for execution speed, PTfuzz is faster than QAFL in all 10 experiments. Especially for objdump, PTfuzz is about 24x faster than QAFL in execution. And PTfuzz hit more branches than QAFL in 10 programs, which means PTfuzz always has a higher code coverage in fuzzing than QAFL.

In addition, when fuzzing mpg321 and tcpdump, QAFL **gets stuck after fork server is up**. (Starting fork server is a basic step in AFL and QAFL fuzzing.) We assume this problem is due to slow executing speed of QAFL and it cannot continue to later steps. However, our PTfuzz has not faced this situation in fuzzing jobs. This problem also proves the slow execution of QAFL.

So by comparing to AFL and QAFL in binary-only fuzzing, we can find out that PTfuzz outperforms them in many ways. PTfuzz is capable of fuzzing binaries where AFL cannot, because PTfuzz doesn't need to specially compile target programs like AFL does. Moreover, PTfuzz runs much faster than QAFL, exposes more crashes and covers more program paths. The internal reason for this is that we abandon QEMU emulation adopted in QAFL and concentrate on Processor Trace, which is capable of fuzzing binaries fast and accurately. So in binary-only fuzzing tasks, PTfuzz earns much more potential to discover deeper bugs and vulnerabilities than AFL and QAFL.

C. EXPERIMENTS ON LAVA-M DATA SET

In this section, we will compare experiment results of PTfuzz, AFL and QAFL on LAVA-M data set. LAVA is a technique to produce ground-truth corpora by injecting real bugs into target programs [42]. The authors created LAVA-M data set by injecting 4 commonly seen GNU programs: base64, md5sum, uniq and who. Recent works, such as VUzzer [8],

TABLE 5. Experiments on LAVA-M data set.

	PTfuzz			AFL			QAFL		
Program	**Crashes**	**Speed**	**Branches**	**Crashes**	**Speed**	**Branches**	**Crashes**	**Speed**	**Branches**
base64	1	167/s	2154	0	241/s	1940	0	86/s	813
md5sum	-	-	-	-	-	-	-	-	-
uniq	1	429/s	3698	0	651/s	2510	0	285/s	1460
who	0	730/s	2268	0	925/s	2051	0	463/s	1211

tend to use LAVA-M data set as benchmarks to compare performance of different fuzzers. We conduct experiments on base64, md5sum, uniq and who separately and the experiment results are shown in Table 5. Specifically, in the table, target programs are compile-time instrumented for AFL because AFL cannot fuzz raw binaries, and not instrumented for PTfuzz and QAFL.

As shown in the table, PTfuzz exposes more crashes than AFL and QAFL in base64 and uniq. As for execution speed, PTfuzz outperforms QAFL in all the target programs, but slower than AFL, which is the same as previous experiments. Moreover, PTfuzz is able to cover more edges than AFL and QAFL in all experiments, which means that with accurate coverage feedback, PTfuzz is able to fuzz the target programs more precisely. In the experiment of md5sum, we could not fuzz it in PTfuzz, AFL, and QAFL, because it crashed on the first input without allowing the program to execute more of other inputs. And the same phenomenon occurs in VUzzer, so we mark "–" in the experiment results of md5sum.

According to the table, PTfuzz performs poorly on finding crashes in LAVA-M data set, the same goes for AFL and QAFL. The reason for this is that bugs injected into LAVA-M are all protected by conditional checks on values copied from input against hardcoded magic bytes. Thus, without static analysis tools, such as in VUzzer, PTfuzz cannot easily recover the expected values used in the checks that guard the injected bugs.

So in conclusion, we conduct experiments on widely used benchmark LAVA-M data set in this section. By comparison to AFL and QAFL, we can prove that PTfuzz has better fuzzing performance in this situation. And our PTfuzz can fuzz most of the target programs except md5sum, so we can make the conclusion that PTfuzz can be scalable to most of the fuzzing situations.

VI. DISCUSSION

A. LIMITATIONS

Although our PTfuzz has relatively better fuzzing performance than AFL and QAFL, there are some limitations of our approach.

- Hardware and OS requirement. Intel Processor Trace is a new feature only after Intel Core processors (Broadwell architecture), so PTfuzz cannot run on computer with older version of Intel CPUs. Moreover, our implementation of Processor Trace output space (ToPA [21]) relies on a feature called *mmap_page* in perf_event [36], and this feature is only available after Kernel 4.1.x. So PTfuzz needs a new version of kernel to run. But this ToPA implementation problem can be solved by manually specifying a memory space as ToPA, and which is totally the same as using mmap_page.
- Implementation of decoder. As shown in Section V, PTfuzz is slower than AFL in fuzzing jobs. By analyzing our code of PTfuzz, we find out that our decoder of recorded Processor Trace information can be tremendously accelerated by performing a pipeline-like decoding. In other words, in our present implementation, PTfuzz decodes tracing data of a test case in ToPA only after the test case finishes its execution. So between execution start and execution finish, decoder has nothing to do. And we can improve our PTfuzz by performing decoding when test case is executing, instead of after execution finish.
- Only on Linux platform. Currently our PTfuzz can only fuzz Linux programs due to our PT implementation relies on Linux system. However, Processor Trace recently adds Windows support to help operation of PT on Windows,[4] So in the future we plan to transplant PTfuzz to Windows platform and fuzz Windows applications in the same way.

VII. CONCLUSION

In this paper, we concentrate on greybox fuzzing to expose bugs and vulnerabilities in softwares. Meanwhile we examine previous greybox fuzzers and find out some common drawbacks of them. Some fuzzers cannot support binary-only fuzzing, some has low code coverage and some suffer from huge overhead. To address these limitations, we introduce a greybox fuzzing technique assisted by Intel Processor Trace technology and implement a prototype called PTfuzz. We accurately record basic block transitions in program execution with PT in a relatively fast execution speed, and achieve higher code coverage than previous fuzzers. And experiment results demonstrate that PTfuzz outperforms AFL and QAFL in most of the 3 indicators, crashes, speed, and branches. The result is a strong evidence that PTfuzz is much

[4]https://github.com/intelpt/WindowsIntelPT

more effective in fuzzing jobs and is able to expose deeper bugs and vulnerabilities in programs.

REFERENCES

[1] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.
[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, p. 10, 2008.
[3] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, vol. 8, 2008, pp. 209–224.
[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.
[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for *in-vivo* multi-path analysis of software systems," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2011, pp. 265–278.
[6] M. Zalewski. *American Fuzzy Lop (AFL) Fuzzer-Technical Details*. Accessed: Jan. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt
[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.
[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2017, pp. 1–14.
[9] J. Hertz and T. Newsham. *Project Triforce*. Accessed: Jan. 1, 2018. [Online]. Available: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything
[10] Intel Corporation. *Intel PIN*. Accessed: Jan. 1, 2018. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[11] Talos Vulndev *et al. AFL-Dyninst*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/talos-vulndev/afl-dyninst
[12] P. Bonzini *et al. Qemu*. Accessed: Jan. 1, 2018. [Online]. Available: https://www.qemu.org
[13] Michal Zalewski. *Readme.QEMU*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu
[14] Intel Corporation. *Intel Processor Trace*. Accessed: Jan. 1, 2018. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
[15] Michal Zalewski. *AFL-Status_Screen.Txt*. Accessed: Jan. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/status_screen.txt
[16] A. Edwards, H. Vo, and A. Srivastava, "Vulcan: Binary transformation in a distributed environment," Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2001-50, 2001.
[17] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "Alto: A link-time optimizer for the Compaq Alpha," *Softw. Pract. Exper.*, vol. 31, no. 1, pp. 67–101, 2001.
[18] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 882–945, 2005.
[19] Dmitry Vyukov. *Syzkaller*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/google/syzkaller
[20] D. Bruening and S. Amarasinghe, "Efficient, transparent, comprehensive runtime code manipulation," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2004.
[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, System Programming Guide*. Accessed: Jan. 1, 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf
[22] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1083–1094.
[23] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice—Automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.
[24] D. Engler and D. Dunbar, "Under-constrained execution: Making automatic code destruction easy and scalable," in *Proc. Int. Symp. Softw. Test. Anal.*, 2007, pp. 1–4.
[25] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. Usenix Conf. Secur.*, 2013, pp. 49–64.
[26] M. Neugschwandtner, P. M. Comparetti, I. Haller, and H. Bos, "The BORG: Nanoprobing binaries for buffer overreads," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 87–97.
[27] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proc. Usenix Workshop Offensive Technol.*, 2007, pp. 1–9.
[28] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE Int. Conf. Softw. Eng.*, May 2009, pp. 474–484.
[29] D. Molnar, C. L. Xue, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proc. Conf. Usenix Secur. Symp.*, 2009, pp. 67–82.
[30] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 497–512.
[31] M. Zalewski. *American Fuzzy Lop (AFL) Fuzzer*. Accessed: Jan. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl
[32] Aki Helin. *A General-Purpose Fuzzer*. Accessed: Sep. 1, 2017. [Online]. Available: https://github.com/aoh/radamsa
[33] *Sam Hocevar. ZZUF*. Accessed: Sep. 1, 2017. [Online]. Available: https://github.com/samhocevar/zzuf
[34] *Peach Tech. Peach*. Accessed: Sep. 1, 2017. [Online]. Available: https://www.peach.tech
[35] *Andi Kleen Simple-PT*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/andikleen/simple-pt
[36] *Frederic Weisbecker*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/perf_event.h
[37] *Markus Metzger Intel PT Decoder Library*. Accessed: Jan. 1, 2018. [Online]. Available: https://github.com/01org/processor-trace
[38] GNU. *GNU Binutils*. Accessed: Jan. 1, 2018. [Online]. Available: http://www.gnu.org/software/binutils
[39] *Frank Warmerdam TIFF*. Accessed: Jan. 1, 2018. [Online]. Available: http://www.remotesensing.org/libtiff
[40] *Joe Drew. MPG321*. Accessed: Jan. 1, 2018. [Online]. Available: http://mpg321.sourceforge.net
[41] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. 24th ACM Conf. Comput. Commun. Secur. (CCS)*, 2017, pp. 1–16.
[42] B. Dolan-Gavitt *et al.*, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.
[43] M. Zalewski. *American Fuzzy Lop (AFL) README*. Accessed: Jan. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt

GEN ZHANG was born in Chongqing, China, in 1993. He received the bachelor's degree majoring in computer science and software analysis from the National University of Defense Technology, Changsha, China, in 2016, where he is a currently pursuing the master's degree with the College of Computer.

He has authored two papers in ISCID 2017 and ICSMO 2018. His current research interests are fuzzing, software analysis, and binary analysis. He received the Extraordinary Student Award from the College of Computer, National University of Defense Technology, for his good performance in 2015.

XU ZHOU was born in Shanxi, China, in 1985. He received the Ph.D. degree majoring in computer science and parallel from the National University of Defense Technology (NUDT), Changsha, China, in 2013. He is currently an Assistant Researcher with the College of Computer, NUDT.

He has authored papers on PPoPP and several top transactions on parallel. His research interests are computer system and parallel.

YINGQI LUO was born in Hubei, China, in 1994. He received the B.S. degree from the National University of Defense Technology (NUDT), Changsha, China, in 2016.

He is currently pursuing the master's degree with the School of Computer, NUDT. His research interests include software analysis and reverse engineering.

XUGANG WU was born in Guangdong, China, in 1995. He received the B.S. degree from the National University of Defense Technology (NUDT), Changsha, China, in 2017.

He is currently pursuing the Ph.D. degree with the School of Computer, NUDT. His research interests include software analysis and fuzzing.

ERXUE MIN was born in Jiangsu, China, in 1994. He received the B.S. degree from the National University of Defense Technology (NUDT), Changsha, China, in 2016.

He is currently pursuing the master's degree with the School of Computer, NUDT. His research interests include machine learning, data mining, optimization, and intrusion detection.