STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting

Zhuo Zhang[§], Wei You[†*], Guanhong Tao[§], Yousra Aafer[‡], Xuwei Liu[§], Xiangyu Zhang[§]
[§]Purdue University, [†]Renmin University of China, [‡]University of Waterloo
{zhan3299, taog, liu2598, xyzhang}@purdue.edu, youwei@ruc.edu.cn, yousra.aafer@uwaterloo.ca

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00109

***Abstract*—Fuzzing stripped binaries poses many hard challenges as fuzzers require instrumenting binaries to collect runtime feedback for guiding input mutation. However, due to the lack of symbol information, correct instrumentation is difficult on stripped binaries. Existing techniques either rely on hardware and expensive dynamic binary translation engines such as QEMU, or make impractical assumptions such as binaries do not have inlined data. We observe that fuzzing is a highly repetitive procedure providing a large number of trial-and-error opportunities. As such, we propose a novel incremental and stochastic rewriting technique STOCHFUZZ that piggy-backs on the fuzzing procedure. It generates many different versions of rewritten binaries whose validity can be approved/disapproved by numerous fuzzing runs. Probabilistic analysis is used to aggregate evidence collected through the sample runs and improve rewriting. The process eventually converges on a correctly rewritten binary. We evaluate STOCHFUZZ on two sets of real-world programs and compare with five other baselines. The results show that STOCHFUZZ outperforms state-of-the-art binary-only fuzzers (e.g., *e9patch*, *ddisasm*, and *RetroWrite*) in terms of soundness and cost-effectiveness and achieves performance comparable to source-based fuzzers. STOCHFUZZ is publicly available [1].**

I. INTRODUCTION

Grey-box fuzzing [2]–[5] is a widely used security testing technique that generates inputs for a target program to expose vulnerabilities. Starting from some *seed inputs*, a fuzzer repetitively executes the program while mutating the inputs. The mutation is usually guided by coverage information. For instance, a popular strategy is that input mutations leading to coverage improvement are considered important and subject to further mutations. As such, existing fuzzing engines rely on instrumentation to track code coverage. Typically, they leverage compilers to conduct instrumentation before fuzzing when source code is available. However in many cases, only binary executables are available. Various techniques have been developed to support fuzzing applications without source code. We call them *binary-only fuzzing* techniques.

Existing binary-only solutions fall into three categories: (1) leveraging hardware support, (2) leveraging on-the-fly dynamic binary rewriting, and (3) relying on offline static binary rewriting. The first category makes use of advanced hardware support such as Intel PT [6] to collect runtime traces that can be post-processed to acquire coverage information. Such traces record individual executed basic blocks, which are generated at a very high rate, and hence require substantial efforts to process. In addition, it is difficult to collect runtime information other than control-flow traces. The second kind uses dynamic rewriting engines such as QEMU [7] and PIN [8], which instrument a subject binary during its execution. They trap execution of each new basic block and rewrite it on the fly. The rewritten basic block is then executed. The method is sound but expensive due to the heavyweight machinery (4-5 times slower than source based fuzzing according to our experiment in Section VI). The third kind instruments the binary just once before the whole fuzzing process. However, sound static binary rewriting is an undecidable problem [9] due to the lack of symbol information. It entails addressing a number of hard challenges such as separating code and data, especially inlined data [10], [11], and identifying indirect jump and call targets [12], [13]. Existing solutions are either based on heuristics and hence unsound [12], [14], or based on restricted assumptions such as no inlined data is allowed [15] and relocation information must be available [16]. However, these assumptions are often not satisfied in practice. According to our experiment in Section VI, a number of state-of-the-art solutions, such as *e9patch* [15] and *ddisasm* [12] fail on real-world binaries.

We observe that fuzzing is a highly repetitive process in which a program is executed for many times. As such, it provides a large number of chances for trial-and-error, allowing rewriting to be incremental and progress with increasing accuracy over time. We hence propose a novel *incremental* and *stochastic* rewriter that piggy-backs on the fuzzing procedure. It uses probabilities to model the uncertainty in solving the aforementioned challenges such as separating data and code. In other words, it does not require the binary analysis to acquire sound results to begin with. Instead, it performs initial rewriting based on the uncertain results. The rewritten binary is very likely problematic. However, through a number of fuzzing runs, the technique automatically identifies the problematic places and repairs them. The process is stochastic. It does not use a uniform rewritten binary. Instead, it may rewrite the binary differently for each fuzzing run by drawing samples from the computed probabilities. It randomly determines if bytes at some addresses ought to be rewritten based on the likelihood that the addresses denote an instruction. As such, the problematic rewritings are distributed and diluted among

*Corresponding author

© 2021, Zhuo Zhang. Under license to IEEE.
DOI 10.1109/SP40001.2021.00109

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

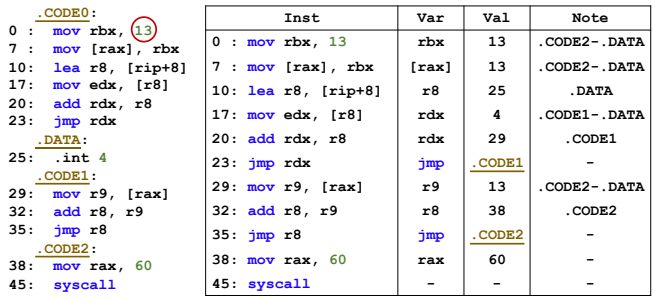
many versions, allowing easy fault localization / repair and ensuring fuzzing progress. Note that if a binary contains too many rewriting problems, the fuzzer may not even make reasonable progress, significantly slowing down the convergence to precise rewriting. In contrast, during stochastic rewriting, while some versions fail at a particular place, many other versions can get through the place (e.g., as they do not rewrite the place), which in turn provides strong hints to fix the problem. The probabilities are updated continuously across fuzzing runs as our technique sees more code coverage and fixes more rewriting problems, affecting the randomly rewritten versions. At the end, the uncertainty is excluded when enough samples have been seen, and the process converges on a stable and precisely rewritten binary.

Our contributions are summarized as follows.

- We propose a novel incremental and stochastic rewriting technique that is particularly suitable for binary-only fuzzing. It piggy-backs on fuzzing and leverages the numerous fuzzing runs to perform trial-and-error until achieving precise rewriting.
- The technique is facilitated by a lightweight approach that determines the likelihood of each address denoting a data byte. We formally define the challenge as a probabilistic inference problem. However, standard inference algorithms are too heavyweight and not sufficiently scalable in our context, which requires recomputing probabilities and drawing samples during fuzzing. We hence develop a lightweight approximate algorithm.
- We develop a number of additional primitives to support the process, which include techniques to automatically locate and repair rewriting problems.
- We develop a prototype StochFuzz and evaluate it on the Google Fuzzer Test Suite [17], the benchmarks from *RetroWrite* [16], and a few commercial binaries. We compare it with state-of-the-art binary-only fuzzers *e9patch* [15], *ptfuzzer* [18], *ddisam* [12], *afl-qemu* [19] and *RetroWrite* [16] and also with source based fuzzers *afl-gcc* [3] and *afl-clang-fast* [20]. Our results show that StochFuzz outperforms these binary-only fuzzers in terms of soundness and efficiency, and has comparable performance to source based fuzzers. For example, it is 7 times faster than *afl-qemu*, and successfully handles all the test programs while other static binary rewriting fuzzers fail on $12.5-37.5\%$ of the programs. Our fuzzer also identifies zero-days in commercial binaries without any symbol information. We have conducted a case study in which we port a very recent source based fuzzing technique IJON [21] that tracks state feedback in addition to coverage feedback, to support binary-only fuzzing. It demonstrates the applicability of StochFuzz. Our system and benchmark corpora are publicly available [1].

II. Motivation

In this section, we use an example to illustrate the limitations of existing binary-only fuzzing techniques and motivate ours. Fig. 1 presents a piece of assembly code for illustration



Inst	Var	Val	Note
0 : mov rbx, 13	rbx	13	.CODE2-.DATA
7 : mov [rax], rbx	[rax]	13	.CODE2-.DATA
10: lea r8, [rip+8]	r8	25	.DATA
17: mov edx, [r8]	rdx	4	.CODE1-.DATA
20: add rdx, r8	rdx	29	.CODE1
23: jmp rdx	jmp	.CODE1	-
29: mov r9, [rax]	r9	13	.CODE2-.DATA
32: add r8, r9	r8	38	.CODE2
35: jmp r8	jmp	.CODE2	-
38: mov rax, 60	rax	60	-
45: syscall	-	-	-

Fig. 1: Motivation example

purpose (its functionality is irrelevant). The right side of the figure depicts its execution trace - where the executed instructions, destination registers, and evaluation results are listed in the first three columns, respectively. The last column presents the related section(s) if the evaluated result is address relevant. For example, the value 25 generated by the instruction at address 10 denotes an address in the `.DATA` section while the value 29 generated by the instruction at address 20 denotes an address in `.CODE1`.

As shown, the snippet consists of three code sections (i.e., `.CODE0`, `.CODE1`, and `.CODE2`) and an interleaved data section `.DATA`. The first two instructions (at addresses 0 and 7) in `.CODE0` load a constant 13 to `rbx`, and then store it in a memory location denoted by `[rax]`. The constant 13 denotes the offset between the `.CODE2` section and the `.DATA` section, i.e., 38-25=13, and will be used later in addressing. The three instructions at addresses 10, 17, and 20 calculate the address of label `.CODE1`. Specifically, `r8` is first set to the address of `.DATA` via a PC-related `lea` instruction. At address 17, an integer 4 representing the offset between labels `.CODE1` and `.DATA` is loaded from the memory address denoted by `[r8]` (i.e., address 25) to `edx`, which consequently updates `rdx`. Next, `r8` is added to `rdx`. The resulting `rdx` denotes the address of `.CODE1`. The subsequent instruction at 23 triggers an indirect jump to label `.CODE1`. The next two instructions at addresses 29 and 32 determine the target of the indirect jump at address 35 (i.e., `.CODE2`) by loading the offset 13 from `[rax]` and adding it to the address of `.DATA` stored in `r8`. A `syscall` is invoked subsequently once the indirect jump is triggered. Observe that the code snippet has inlined data, indirect jumps, and complex address computation, which pose substantial challenges to existing binary-only fuzzers.

A. Limitations of Existing Technique

Recall that fuzzers need to collect runtime feedback such as code coverage to guide input mutation. For binary-only fuzzers, such feedback can be captured by a technique in one of the following three categories: (1) hardware-assisted tracing, (2) dynamic binary instrumentation, and (3) static binary rewriting. In Table I, we summarize the characteristics of existing techniques. Column 1 lists these techniques, with the first two being source-based AFL fuzzers using *gcc* and *clang* compilers, *ptfuzzer* using hardware-assisted tracing, *afl-qemu* using dynamic instrumentation, and the others including

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Summary of different *binary-only fuzzing* instrumentation techniques, along with compiler instrumentation (*afl-gcc* and *afl-clang-fast*). A1 denotes that the binary has symbol and relocation information, A2 denotes that the binary is Position Independent, A3 denotes that all instruction boundaries are correctly identified by upstream disassembler, and A4 denotes that the binary does not contain any inlined data. S1 denotes that the tool supports binaries compiled from C++ programs, and S2 denotes that the tool supports collecting other runtime information than coverage. Note that the soundness of StochFuzz can be guaranteed when there is no inlined data, and probabilisticly guaranteed otherwise.

Tool	Prerequisite				Support		Soundness	Efficiency
	A1	A2	A3	A4	S1	S2		
afl-gcc	Require Source Code						Sound	A
afl-clang-fast	Require Source Code						Sound	A+
ptfuzzer [18]	-	-	-	-	Y	N	Sound	C
afl-qemu	-	-	-	-	Y	Y	Sound	D
afl-dyninst [14]	-	-	✓	-	Y	Y	Unsound	A
e9patch [15]	-	-	✓	✓	Y	Y	Sound	B
RetroWrite [16]	✓	✓	✓	✓	N	Y	Unsound	A
ddisasm [12]	-	-	-	-	Y	Y	Unsound	A
StochFuzz	-	-	-	✓	Y	Y	Sound	A
	-	-	-	-	Y	Y	Prob sound	A

ours using static binary rewriting. Columns 2-5 are the assumptions made by these tools, where ✓ denotes that a specific precondition is required. Columns 6 and 7 show whether C++ programs and other runtime feedback beyond coverage are supported, respectively. Column 8 denotes the soundness guarantee which means if the technique guarantees to rewrite the binary properly and collect the right feedback, and column 9 denotes fuzzing efficiency with A+ the best.

Hardware-assisted Tracing. Modern processors offer a mechanism that captures software execution information using dedicated hardware [6]. PTFuzzer [18] leverages this feature to collect code coverage for binary-only fuzzing. For instance, after executing the code in Fig. 1, two control transfers are recorded, i.e., from 23 to 29 and from 35 to 38. Based on the information, PTfuzzer subsequently recovers the execution path and hence the coverage. Other hardware-assisted fuzzers operate similarly [22], [23]. The performance of these approaches is limited by the costly trace post-processing (4× slower than *afl-clang-fast* according to our experiments). Additionally, hardware-assisted fuzzing cannot capture other runtime feedback than coverage [4], [21].

Dynamic Instrumentation. Dynamic instrumentation translates and instruments the binary during execution [7], [8]. Although it is an attractive solution due to its sound instrumentation, the on-the-fly translation/instrumentation incurs relatively higher runtime overhead compared to other approaches. *Afl-qemu*, to the best of our knowledge, is among the best-performing binary-only fuzzers based on dynamic instrumentation. It still incurs significant overhead (5× slower than *afl-clang-fast* according to our experiments). Other approaches in this category, including *afl-pin* [24] and *afl-dynamorio* [25], induce even higher overhead.

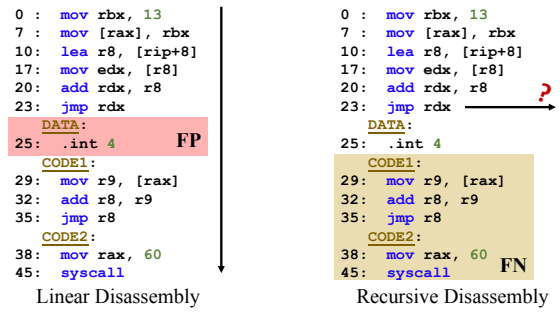


Fig. 2: Limitations of disassembly methods. The red box shows corrupted code, and the yellow box shows missing code.

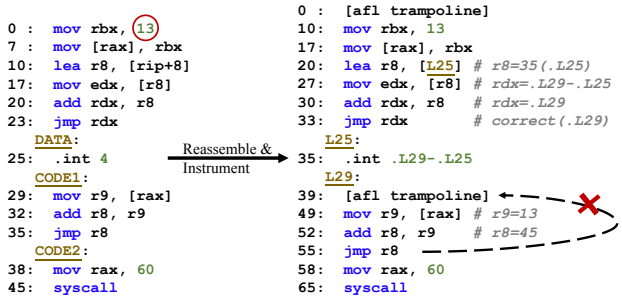


Fig. 3: Reassembly in *RetroWrite*. It crashes as the constant 13 in red circle is not properly symbolized.

Static Binary Rewriting. Static rewriting utilizes binary analysis to disassemble and rewrite the binary before execution. Unfortunately, it is still a hard challenge to rewrite stripped binary with soundness guarantee. Existing solutions often make unsound assumptions about the target binary which may lead to runtime failures.

Afl-dyninst [14], a trampoline-based approach built upon traditional disassembly techniques, assumes the upstream disassemblers can correctly identify all the instructions. However, such assumption may not hold in practice due to code and data interleavings [15], [16]. Fig. 2 demonstrates how the code example in Fig. 1 breaks its assumption. The left of Fig. 2 shows that a *linear disassembly*, which decodes all bytes consecutively, is confused by address 25, the inlined data byte. *Recursive disassembly*, on the other hand, avoids this problem by disassembling instructions following control flow. But it fails to resolve the target of the indirect jump at address 23, missing the code from address 29 to 45.

E9patch [15] makes the same assumption as *afl-dyninst*, and additionally assumes there is no inlined data. With these assumptions, *e9patch* specially engineers jumps that can safely overlap with other instructions. As such, it can insert trampolines without sacrificing the correctness of rewriting. In addition, it uses a sophisticated virtual address space layout for the instrumented binary, which on the other hand might make it susceptible to a large number of cache misses and additional overhead in process forking [26].

RetroWrite [16] is a reassembly technique for *Position Independent Code* (PIC). It converts address related immediate values in the binary to symbols (called *symbolization*) such

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

that they can be easily relocated after instrumentation. For example in Fig. 3, the "`lea r8, [rip+8]`" instruction at address 10 is translated as "`lea r8, [L25]`", because *RetroWrite* recognizes that `rip+8` denotes a reference in the code space and needs to be symbolized. As such, it could be properly relocated after instrumentation. However, sound static symbolization is provably undecidable [9] in general. *RetroWrite* consequently makes strong assumptions such as the requirement of relocation information and the exclusion of C++ exception handlers. However, even if these requirements were satisfied, the soundness of *RetroWrite* still could not be guaranteed due to the need of sound memory access reasoning. In the right side of Fig. 3, recognizing that the constant 13 in the first instruction "`mov rbx, 13`" is an address offset (and needs symbolization) is challenging, due to the long sequence of complex memory operations between this instruction and the final address de-reference at 55, which ultimately discloses constant 13 is an address offset. In the example, *RetroWrite* misclassifies 13 as a regular value. As a result, it is not symbolized. Ideally, it should be symbolized to `.L38-.L25`, which would be concretized to 58-35=23 after instrumentation. As a result, *RetroWrite* crashes on the binary. A recent study [12] shares the same concern.

Ddisasm [12] is a state-of-the-art reassembly technique. Rather than making assumptions about target programs, it relies on a large set of reassembly heuristics such as instruction patterns. These heuristics, although comprehensive, have inherent uncertainty and may fail in many cases.

B. Our Technique

Our technique is inspired by two important insights. *First insight*: *while grey-box fuzzers continuously mutate inputs across test runs, they may as well be enhanced to mutate the program on-the-fly. As such, disassembly and static rewriting (which are difficult due to the lack of symbol information and difficulties in resolving indirect jumps/calls offline) can be incrementally performed over time*.

Example. We use case A in the first row of Fig. 4 to demonstrate how our technique leverages the first insight. The workflow consists of four steps, an initial patching step prior to fuzzing (step ①) and three incremental rewriting steps during fuzzing (steps ②, ③, and ④).

In the snippet to the left of ①, the code sections are filled with a special one-byte `hlt` instruction , which will cause a segfault upon execution. A segfault by a `hlt` instruction indicates that the system has just discovered a code region that has not been properly disassembled or rewritten such that incremental rewriting should be performed. We will explain later how we separate code and data in the first place (as only code is replaced with `hlt` in the snippet). The separation of the two does not have to be precise initially and our stochastic rewriting (discussed later) can gradually improve precision over the numerous fuzzing runs. For instance, the execution of initial patched code is terminated by the `hlt` at address 0, indicating a new code region. For easy description, we call such segfaults *intentional crashes*.

The next step (incrementally) rewrites all the addresses that can be reached along direct control flow from the address where the intentional crash happens. Specifically, STOCHFUZZ places the rewritten code in a new address space, called the *shadow space*; it further redirects all the direct jumps and calls to their new targets in the shadow space by patching immediate offsets; and since data sections are retained in their original space, any PC-dependent data references need to be properly patched too. At last, STOCHFUZZ inserts a jump instruction at the crash address to direct the control flow to the shadow space. In the code snippet in between ① and ②, given the crash address 0, STOCHFUZZ disassembles the instructions from addresses 0 to 23 (highlighted in green shade). These instructions are then rewritten in the shadow space starting from address 90. Specifically, an `afl trampoline` is inserted at the beginning to collect coverage information, and the original "`lea r8, [rip+8]`" instruction (at address 10) is rewritten to "`lea r8, [rip-92]`" (at address 110) to ensure the data reference occurs at the original address. STOCHFUZZ inserts a "`jmp 90`" instruction at 0 to transfer the control flow. Then, the fuzzer continues fuzzing with the new binary and the incremental rewriting is invoked again if other intentional crashes occur (e.g., steps ② and ③). □

A prominent challenge is to separate code and data in executables, especially when inlined data are present. Due to the lack of symbol information, it is in general an undecidable problem [9]. Heuristics or learning based solutions [12], [27] are inevitably unsound. Data may be recognized as instructions and replaced with `hlt`. As a result, the program may execute with corrupted data which may or may not manifest themselves as crashes. Corrupted states may lead to bogus coverage and problematic test results. On the other hand, instructions may be recognized as data and hence not replaced with `hlt`. Consequently, these instructions are invisible to our system and not instrumented.

The following second insight allows us to address the aforementioned problem. *Second insight: fuzzing is a highly repetitive process that provides a large number of opportunities for trial-and-error. That is, we can try different data and code separations, which lead to different instrumented executables, in different fuzzing runs. Over time, an increasing number of samples can be collected, allowing us to achieve the precise separation and correct rewriting*. There are two challenges that we need to overcome in order to leverage the insight. First, we need to distinguish exceptions caused by rewriting errors (introduced by our trial-and-error) and by latent bugs in the subject program. We call both *unintentional crashes* (to distinguish from *intentional crashes* by `hlt`). We also need to pinpoint and repair rewriting errors, i.e., data bytes misclassified as code (and undesirably replaced with `hlt`), and vice versa. We call it the *self-correction requirement*. Second, an executable cannot contain too many rewriting errors. Otherwise, the fuzzing runs of the executable can hardly make progress (as it continues to crash on these errors one after another). Note that we rely on the fuzzer's progress to collect more and more samples to correct our rewritings. We

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

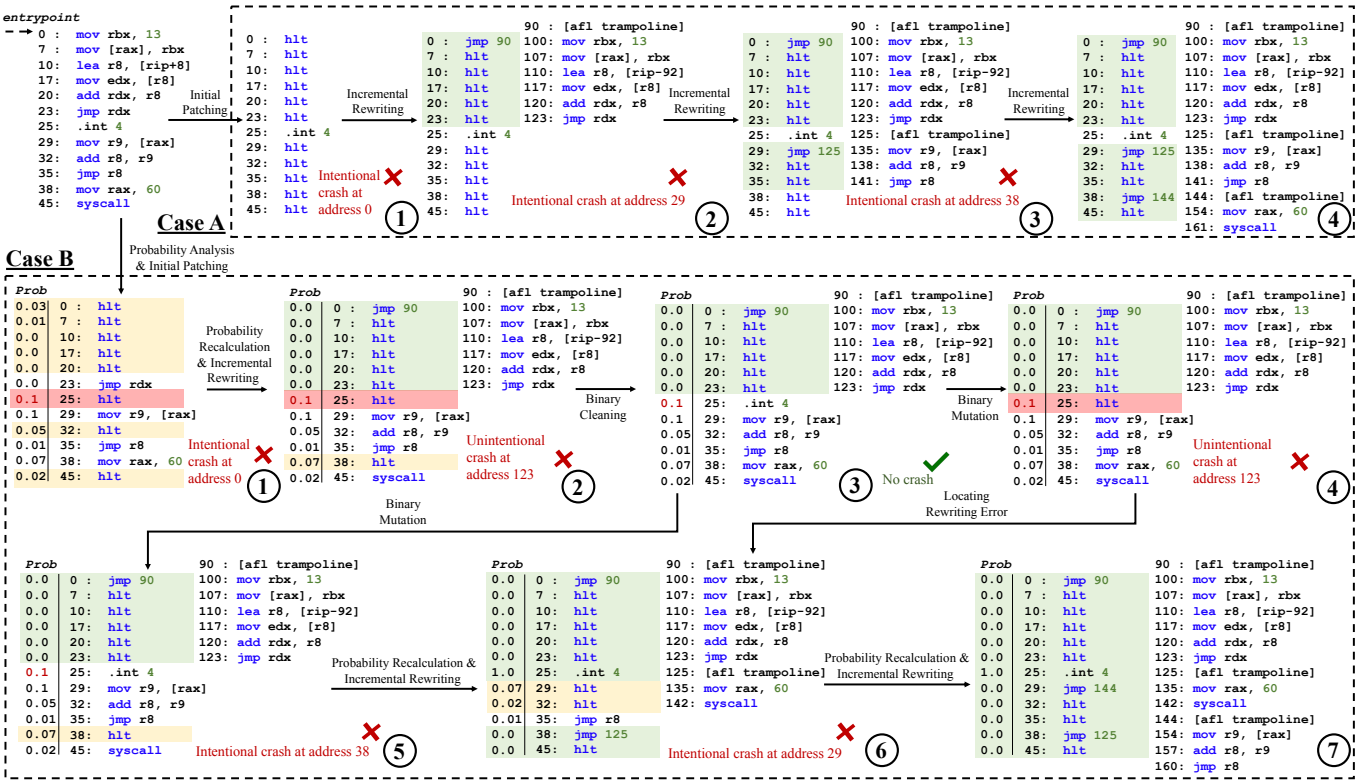


Fig. 4: How StochFuzz handles the motivation example

call it the *progress requirement*.

We therefore propose a novel *stochastic rewriting technique* that piggy-backs on the fuzzing procedure. At first, the technique performs probabilistic inference to compute the likelihood of individual bytes in the original address space belonging to data (or code). Such probabilities are computed based on various hints, such as register definition-use relations that often indicate instructions and consecutive printable bytes that often suggest data. Details of the probabilistic inference can be found in Section III-A. Since these hints are inherently uncertain (e.g., printable bytes may not be data), we use probabilities to model such uncertainty. Based on the computed probabilities, StochFuzz randomly generates a rewritten version for each fuzzing run. In a random version, the bytes replaced with `hlt` are determined by sampling based on their computed probabilities. For instance, a byte with a high probability of being code is more likely replaced with `hlt`. When a segfault is observed, StochFuzz determines if it is caused by a rewriting error, by running the failure inducing input on a binary with all the uncertain rewritings removed and observing if the crash disappears. If so, delta debugging [28], a binary-search like debugging technique, is used to determine the root cause rewriting. Over time, the corrected rewritings, together with the new coverage achieved during fuzzing, provide accumulating hints to improve probabilistic inference and hence rewriting. Note that the proposed solution satisfies the two aforementioned requirements: the rewriting errors are distributed in many random versions such that the fuzzer can make progress in at least some of them; and they can be automatically located and repaired.

Example Continued. We use case B (the lower box) in Fig. 4 to illustrate stochastic rewriting. At the beginning (the snippet to the left of ① in case B), StochFuzz computes the initial probabilities (of being data bytes) as shown to the left of the individual addresses. For example, a definition-use relation between addresses 0 and 7 caused by `rbx` decreases their probability of being data. Assume in a random binary version the addresses with color shades are replaced by `hlt`, with the yellow ones being the correct replacements as they denote instructions and the red one erroneous since a data byte is replaced with a `hlt`. The binary is executed and then an intentional crash is encountered at address 0. In the snippet to the right of ①, besides the incremental rewriting mentioned in case A, StochFuzz also performs probability recalculation which updates the probabilities based on the new hints from the execution. Intuitively, as address 0 is code, all addresses (in green shade) reachable from the instruction along control flow must be code. We say that they are "*certainly code*" and their probabilities are set to 0. The probabilities of remaining addresses are updated and new random binaries are generated. *In practice, many of the misclassified bytes such as 25 are proactively fixed by these new hints and updated probabilities, without causing any crashes or even being executed.* This illustrates the importance of the aforementioned progress requirement.

However to make our discussion interesting, we assume 25 (i.e., the data byte) and 38 are still replaced in the new version (i.e., the snippet to the right of ①). During execution, since the data at 25 is corrupted, a wrong target address value is computed for `rdx` in the jump instruction at 123,

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

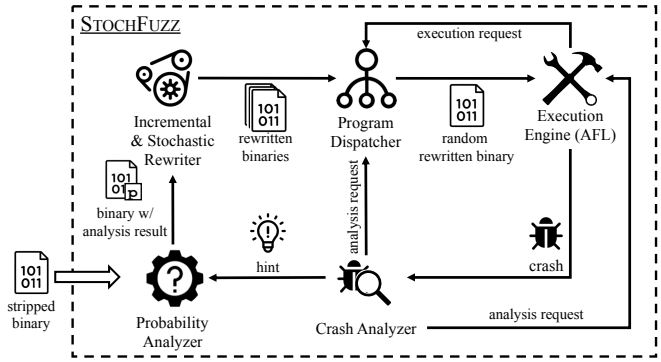


Fig. 5: Architecture

causing a segfault. The diagnosis and self-correction procedure is hence invoked (steps ②-⑤). Specifically, the binary cleaning step ② removes all the rewritings at uncertain addresses (in yellow or red shades) and re-executes the program (to the right of ②). The crash at address 123 disappears, indicating the crash must be induced by a rewriting error. StochFuzz uses delta debugging and generates two binaries, one with only 25 replaced (i.e., the snippet to the left of ④) and the other with 38 replaced (i.e., the snippet to the left of ⑤). The former crashes at the same address 123 whereas the latter crashes at 38 (and hence an intentional crash). As such, StochFuzz determines that the rewriting of address 25 is wrong and fixes it by marking it as "*certainly data*" (i.e., with probability 1.0) in the version to the right of ⑤. This new hint leads to probability updates of other addresses (e.g., 29 and 32). The procedure continues and eventually all addresses have certain classification (i.e., all in green shade) and the program is properly rewritten. □

III. System Design

The architecture of StochFuzz is shown in Fig. 5. It consists of five components: the probability analyzer, the incremental and stochastic rewriter, the program dispatcher, the execution engine, and the crash analyzer. The probability analyzer computes a probability for each address in the given binary to indicate the likelihood of the address denoting a data byte. The rewriter rewrites the binary in different forms by sampling based on the computed probabilities. The program dispatcher selects a rewritten version to execute, either randomly for a normal execution request or strategically for root cause diagnosis. The execution engine, a variant of AFL [3], executes a given binary and monitors for crashes. The crash analyzer triggers incremental rewriting when it determines a crash is intentional; otherwise, it analyzes the root cause and automatically repairs it if the cause is a rewriting error.

StochFuzz has three typical workflows. *Case one* is the most common. It is similar to the standard AFL. Specifically, the execution engine sends a request to the program dispatcher for a binary. The dispatcher randomly selects a rewritten binary (from its pool), which is then executed by the engine. The binary subsequently exits normally without any crash.

In *case two*, the execution is terminated by an intentional crash (i.e., a `hlt` instruction). The crash is reported to the crash analyzer, which identifies the new code coverage indicated by the crash and analyzes the newly discovered code to collect additional hints for distinguishing data and code. The hints are passed on to the probability analyzer, which recomputes the probabilities and invokes the incremental rewriter to generate new binaries.

In *case three*, the execution is terminated by an unintentional crash (i.e., a crash not caused by `hlt`). To verify whether the crash is triggered by some rewriting error, the crash analyzer notifies the program dispatcher to send a binary that has all uncertain rewritings removed for execution. If the previous crash persists, it must be caused by a latent bug in the original program. Otherwise, the crash is caused by rewriting error. The crash analyzer further performs delta-debugging to locate the root cause and repairs it. The repair is passed on as a hint to the probability analyzer and triggers probabilities updates and generation of new binaries. In the remainder of this section, we discuss details of the components.

A. Probability Analyzer

This component computes the probabilities of each address denoting data or code. Initially (before fuzzing starts), it computes the probabilities based on the results of a simple disassembler that we only use to disassemble at *each address in the binary*. During fuzzing, with new observations (e.g., indirect call and jump targets) and exposed rewriting errors, it continuously updates probabilities until convergence. It models the challenge as a *probabilistic inference* problem [29]. Specifically, random variables are introduced to denote individual addresses' likelihood of being data or code. Prior probabilities, which are usually predefined constants as in the literature [30]–[33], are associated with a subset of random variables involved in observable features (e.g., definition-use relations that suggest likely code). Random variables are correlated due to program semantics. The correlations are modeled as probabilistic inference rules. Prior probabilities are propagated and aggregated through these rules until convergence using probabilistic inference algorithms, yielding *posterior probabilities*. In the following, we explain how we define the problem and introduce our lightweight solution.

Definitions and Analysis Facts. As shown in the top of Fig. 7, we use a to denote an address, c a constant, and r a register. The bottom part of Fig. 7 presents the analysis *facts* directly collected from the binary. These facts are deterministic (not probabilistic). $\mathit{Inst}(a, c)$ denotes that the c bytes starting from address a can be encoded as a valid instruction. $\mathit{ExplicitSucc}(a_1, a_2)$ denotes the instruction at address a_2 is an explicit successor of the instruction at address a_1 along control flow. $\mathit{RegWrite}(a, r)$ denotes the instruction at a writes to register r. *RegRead* denotes the read operation. $\mathit{Str}(a, c)$ denotes the c bytes starting from address a constitute a printable null-terminated string.

Initially, StochFuzz disassembles at each address and collects the analysis facts. It collects more facts than those in Fig. 7. They are elided due to space limitations.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

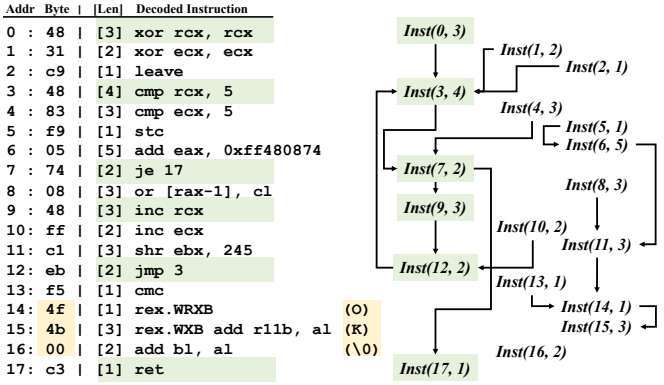


Fig. 6: Universal Control-flow Graph (UCFG) Example. On the left, each address is disassembled (with the real instructions in green shade and the real data in yellow). The corresponding UCFG is in the right.

$a \in \langle\texttt{Address}\rangle ::= \textbf{Integer}$ $\quad c \in \langle\texttt{Constant}\rangle ::= \textbf{Integer}$
$r \in \langle\texttt{Register}\rangle ::= \{\texttt{rax}, \texttt{rbx}, \texttt{rcx}, \texttt{rdx}, \cdots\}$

Inst(a, c) : the c bytes starting from address a can be disassembled as an inst
ExplicitSucc(a_1, a_2)/ : the inst at a_2 is an *explicit* successor of the one at a_1
RegWrite(a, r)/***RegRead***(a, r) : the inst at a writes/reads data into/from reg r
Str*(a, c)* : the c bytes starting from addr a can be interpreted as a printable string

Fig. 7: Definitions for Variables and Analysis Facts

Example. In the left of Fig. 6, StochFuzz disassembles starting from each (consecutive) address of a binary, with the first column showing the addresses, the second column the byte value at the address, the third column the instruction size, and the last column the instruction. For example, the first three bytes "`48 31 c9`" are disassembled to an `xor` instruction and the four bytes starting from address 3 are disassembled to a `cmp` instruction. We highlight the true instructions in green shade, and the true data, an "OK" string, in yellow shade, for discussion convenience. Note that StochFuzz does not assume such separation a priori. A simple sound binary analysis yields the following facts: *Inst*$(7, 2)$ because the instruction at 7 is "`je 17`" whose instruction size is 2, *ExplicitSucc*$(7, 17)$ as the instruction at 7 jumps to 17, *RegWrite*$(9, \texttt{rcx})$, *RegRead*$(9, \texttt{rcx})$, and *Str*$(14, 3)$. □

Predicates. Next, we introduce a set of *predicates* that describe inference results. Different from facts that are deterministic, predicates may be uncertain. A random variable is hence associated with each uncertain predicate, denoting the likelihood of it being true. A subset of the predicates we use are presented in the top of Fig. 8 with those having overline uncertain. *ExplicitReach*(a_1, a_2) denotes that address a_1 can reach a_2 along control flow. In Fig. 6, the path $0-3-7$ leads to *ExplicitReach*$(0, 7)$. *RegLive*(a_1, a_2, r) denotes that register r written by address a_1 is live before the instruction at a_2. As such, we have *RegLive*$(9, 12, \texttt{rcx})$ in Fig. 6. $\overline{\textit{IsInst}(a)}$ denotes the likelihood of address a being code. *IsData*(a) is similar.

(Probabilistic) Inference Rules. In the bottom of Fig. 8, we present a subset of our inference rules. Some of them are probabilistic (i.e., those involving uncertain predicates and having probability on the implication operator). Here,

ExplicitReach(a_1, a_2) : a_1 can *explicitly* reach a_2 along control flow
RegLive(a_1, a_2, r) : register r written by address a_1 is live *before* address a_2
$\overline{\textit{IsInst}(a)}$/$\overline{\textit{IsData}(a)}$: the content at address a is an inst/data byte

① ***ExplicitSucc***$(a_1, a_2) \rightarrow$ ***ExplicitReach***(a_1, a_2)
② ***ExplicitReach***$(a_1, a_2) \wedge$ ***ExplicitSucc***$(a_2, a_3) \rightarrow$ ***ExplicitReach***(a_1, a_3)
③ ***RegWrite***$(a_1, r) \wedge$ ***ExplicitSucc***$(a_1, a_2) \rightarrow$ ***RegLive***(a_1, a_2, r)
④ ***RegLive***$(a_1, a_2, r) \wedge \neg$***RegWrite***$(a_2, r) \wedge$ ***ExplicitSucc***$(a_2, a_3) \rightarrow$ ***RegLive***(a_1, a_3, r)
⑤ ***RegLive***$(a_1, a_2, r) \wedge$ ***RegRead***$(a_2, r) \xrightarrow{p_{\texttt{inst}}\uparrow} \overline{\textit{IsInst}(a_1)} \wedge \overline{\textit{IsInst}(a_2)}$
⑥ $\overline{\textit{IsInst}(a_1)} \wedge$ ***ExplicitReach***$(a_1, a_2) \xrightarrow{1.0} \overline{\textit{IsInst}(a_2)}$
⑦ ***Str***$(a_1, c) \wedge (a_1 \leq a_2 < a_1 + c) \xrightarrow{p_{\texttt{data}}\uparrow} \overline{\textit{IsData}(a_2)}$
⑧ $\overline{\textit{IsData}(a_1)} \wedge \overline{\textit{IsData}(a_2)} \wedge (a_1 \leq a_3 \leq a_2 < a_1 + D) \xrightarrow{p_{\texttt{prop}}\uparrow} \overline{\textit{IsData}(a_3)}$
⑨ $\overline{\textit{IsInst}(a)} \xleftrightarrow{0.0} \overline{\textit{IsData}(a)}$

Fig. 8: Predicates and (Probabilistic) Inference Rules. The predicates with overline are uncertain and rules with probability on top of $\rightarrow$ denote probabilistic inference.

1.0, 0.0, $p_{\texttt{inst}}$, $p_{\texttt{data}}$, and $p_{\texttt{prob}}$ denote prior probabilities that are predefined constants. Rules ① and ② derive control flow relations. Intuitively, an instruction can always reach its explicit successor (rule ①), and if a_1 can reach a_2, it can reach the successors of a_2 (rule ②). Rules ③, ④, and ⑤ are to derive definition-use relations. Specifically, rule ③ denotes that if an instruction writes/defines a register, the register is live before the successor. Rule ④ denotes propagation of register liveness, that is, if a register is live before an instruction and the instruction does not overwrite the register, it remains live after the instruction. Rule ⑤ states that if there is a definition-use relation between a_1 and a_2, both addresses are likely code, with a prior probability $p_{\texttt{inst}}$. Rule ⑥ states that if an address is likely code, all the addresses reachable from the instruction (at the address) have at least the same likelihood of being code. Rule ⑦ states that all bytes in a printable null-terminated string are likely data. Rule ⑧ leverages the continuity property of data and states that if two data addresses are close enough, the addresses in between are likely data too. Rule ⑨ states that an address cannot be code and data at the same time.

Incremental Fact and Rule Updates. New information can be derived during fuzzing and allows facts and rules to be updated. Specifically, new code coverage would allow deriving new facts such *ExplicitSucc(...)* (e.g., newly discovered indirect control flow). When a rewriting error that replaces a data byte a with `hlt` is located, the corresponding predicate $\overline{\textit{IsData}(a)}$ is set to a 1.0 prior probability, meaning "*certainly data*". $\overline{\textit{IsInst}(a)}$ can be similarly updated. These updates will be leveraged by probabilistic inference to update other random variables and eventually affect stochastic rewriting.

Probabilistic Inference by One-step Sum-product. The essence of probabilistic inference is to derive posterior probabilities for random variables by propagating and aggregating prior probabilities (or *observations*) following inference rules. A popular inference method is *belief propagation* [34] which transforms the random variables (i.e., the uncertain predicates) and probabilistic inference rules to a *factor graph* [29], [35], which is bipartite graph containing two kinds of nodes, a

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

variable node for each random variable and a *factor node* for each probabilistic inference rule. A factor can be considered a function over variables such that edges are introduced between a factor node to the variables involved in the rule. Prior probabilities are then propagated and aggregated through the factor graph by an algorithm like *sum-product* [35], which is an iterative message-passing based algorithm. In each iteration, each variable node receives messages about its distribution from the factors connected to the variable, aggregates them through a *product* operation and forwards the resulted distribution through outgoing messages to the connected factor nodes. Each factor receives messages from its variables and performs a marginalization operation, or the *sum* operation. The posterior probabilities of random variables can be derived by normalizing the converged variable values.

However, belief propagation is known to be very expensive, especially when loops are present [36]. Most existing applications handle graphs with at most hundreds of random variables and factors [30]–[33]. However in our context, we have tens of thousands of random variables and factors (proportional to the number of bytes in the binary). Resolving the probabilities may take hours. We observe that the factor graph is constructed from program that has a highly regular structure. The rounds of sum and product operations in the factor graph can be simplified to non-loopy explicit operations along the program structure. We hence propose a *one-step sum-product* algorithm that has linear complexity. The algorithm constructs a *universal control flow graph* (UCFG) that captures the control flow relations between the instructions disassembled at all addresses. Note that the binary's real control flow graph is just a sub-graph of the UCFG. *Observations* (i.e., deterministic facts and predicates that suggest data or code) are explicitly propagated and aggregated along the UCFG, instead of the factor graph. In the last step, a simplest factor graph is constructed for each address to conduct a one-step normalization (from the observations propagated to this address) to derive the posterior probability (of the address holding a data byte). The factor graphs of different addresses are independent, precluding unnecessary interference.

Universal Control Flow Graph. In UCFG, a node is introduced for each address in the binary regardless of code or data, denoting the one instruction disassembled from that address. Edges are introduced between nodes if there is explicit control flow between them. UCFG is formally defined as $G = (V, E)$, where $V = \{a \mid \exists c \text{ s.t. } \mathit{Inst}(a, c)\}$ and $E = \{(a_1, a_2) \mid \mathit{ExplicitSucc}(a_1, a_2)\}$. The right side of Fig. 6 presents the UCFG for the binary on the left. Note that only the shaded sub-graph is the traditional CFG. After UCFG construction, STOCHFUZZ identifies the *strongly connected components* (SCCs) in the UCFG (i.e., nodes involved in loops). A node not in any loop is an SCC itself. For example in Fig. 6, $\mathit{Inst}(0, 3)$ itself is a SCC. $\mathit{Inst}(3, 4)$, $\mathit{Inst}(7, 2)$, $\mathit{Inst}(9, 3)$, and $\mathit{Inst}(12, 2)$ form another SCC. □

One-step Sum-product. The overall inference procedure is described as follows. STOCHFUZZ first performs deterministic inference (following deterministic rules such as rules ①-④). The resulted deterministic predicates such as the antecedents in rules ⑤ and ⑦ are called *observations*, with the former a code observation (due to the definition-use relation) and the latter a data observation. Prior probabilities p_{inst} and p_{data} are associated with them, respectively.

STOCHFUZZ starts to propagate and aggregate these observations using UCFG. Specifically, it uses a product operation to aggregate all the observations in an SCC (i.e., multiplying their prior probabilities), inspired by the sum-product algorithm that uses a product operation to aggregate information across factors. All the addresses within the SCC are assigned the same aggregated value. Intuitively, we consider all the addresses in an SCC have the same likelihood of being code because any observation within an SCC can be propagated to any other nodes in the SCC (through loop). The lower the aggregated value, the more likely the address being code. We say the belief is stronger. The aggregated observations are further propagated across SCCs along control flow, until all addresses have been reached.

Data observations are separately propagated, mainly following rule ⑧. Specifically, STOCHFUZZ scans through the entire address space in order, if any two data observations are close to each other (less than distance D), the addresses in between are associated with a value computed from the prior probabilities of the two bounding observations.

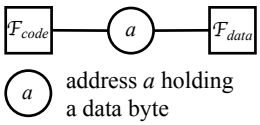
After propagation, each address a has two values denoting the aggregated code observation and the aggregated data observation, respectively. A simple factor graph is constructed for a as shown in Fig. 9. The circled node a is the variable node, representing the likelihood of a being data. It has two factor nodes $\mathcal{F}_{code}$ and $\mathcal{F}_{data}$, denoting the aforementioned two values. According to the sum-product algorithm [35], the posterior probability of a is the normalized product of the two factors as shown in the bottom of the figure. The detailed algorithm and its explanation can be found in Appendix X-H.

Comparison with Probabilistic Disassembly. In probabilistic disassembly [37], researchers use probabilistic analysis to disassemble stripped binaries. It computes probabilities for each address to denote the likelihood of the address belonging to an instruction. However, its problem definition and probability computation are ad-hoc. Its algorithm is iterative and takes tens of minutes to compute probabilities for a medium-sized binary. It has a lot of false positives (around 8%), i.e., recognizing data bytes as instructions. These make it unsuitable for our purpose. In contrast, we formulate the problem as probabilistic inference and propose an algorithm with linear complexity. Piggy-backing on fuzzing, STOCHFUZZ can achieve precise disassembly and rewriting with probabilistic guarantees.

B. Incremental and Stochastic Rewriting

The rewriter is triggered initially and then repetitively when new code is discovered or rewriting errors are fixed. It rewrites instructions in the shadow space (for better instrumentation flexibility) and retains data in the original space. And the original code is replaced with `hlt`. Its rewriting ensures a

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.



a	$\mathcal{F}_{code}$	$\mathcal{F}_{data}$
0	1 - o_{neg}	o_{pos}
1	o_{neg}	1 - o_{pos}

$$P(a=1) = \frac{\mathcal{F}_{code}(1)\cdot\mathcal{F}_{data}(1)}{\mathcal{F}_{code}(1)\cdot\mathcal{F}_{data}(1)+\mathcal{F}_{code}(0)\cdot\mathcal{F}_{data}(0)} = \frac{o_{neg}\cdot(1-o_{pos})}{o_{neg}\cdot(1-o_{pos})+o_{pos}\cdot(1-o_{neg})}$$

Fig. 9: Factor Graph for Each Address

critical property: *a rewritten instruction should evaluate to the same value(s) as its original version*. This ensures all data accesses (to the original space) are not broken. For example, a rewritten read of `rip` must be patched with an offset such that the read yields the corresponding value in the original space as the rewritten read must be executed in the shadow space.

Specifically, it performs the following code transformations. It directly patches direct jump instructions by an offset statically computed based on the offset between the shadow and original address spaces and the instrumentations. The computation of such offset is standard and elided [38]. It instruments all indirect jumps to perform a runtime address lookup that translates the target to the shadow space. It may throw an intentional segfault if it detects the target is not in the shadow space, meaning the corresponding code has not been rewritten. Client analysis instrumentation such as coverage tracking code is inserted in the shadow space.

Handling Call Instructions to Support Data Accesses through Return Addresses. There are programs that access data using addresses computed from some return address on the stack. As such, we need to ensure return addresses saved on the stack must be those in the original space. Therefore, StochFuzz rewrites a call instruction to a `push` instruction which pushes a patched return address (pointing to the original address) to the stack, followed by a `jmp` instruction to the callee in the shadow space. We then instrument `ret` instructions to conduct on-the-fly lookup just like in handling indirect jumps.

Our design allows keeping the control flow in the shadow space as much as possible, which can improve instruction cache performance. An exception is callbacks from external libraries, which cause control flow to the original space, even though it quickly jumps back to the shadow space.

Generating Random Binary Versions. Besides the aforementioned transformations, StochFuzz also performs the following stochastic rewriting to generate a pool of N different binaries (every time the rewriter is invoked). Specifically, for addresses whose their probabilities of being data are smaller than a threshold p_θ but not 0 (i.e., not "certainly code" but "likely code"), they have a chance of $1-p_\theta$ to be replaced with `hlt`. In our setting, we have $N=10$ and $p_\theta=0.01$.

C. Crash Analyzer

Recall that the crash analyzer needs to decide if a crash is due to a rewriting error. If so, it needs to locate and repair the crash inducing rewriting error. Let S be a set of uncertain addresses (that *may* be replaced with `hlt`), and $\mathcal{R}(S)$ the execution result of a rewritten binary where all the addresses in S are replaced with `hlt`. Assume $\mathcal{R}(S_1)$ yields an unintentional crash. To determine whether the crash is caused by a rewriting error, the analyzer compares the results of $\mathcal{R}(S_1)$ and $\mathcal{R}(\emptyset)$. If $\mathcal{R}(S_1)=\mathcal{R}(\emptyset)$, the crash is caused by a latent bug in the subject program, and vice versa.

Then, locating the crash inducing rewriting error can be formalized as finding a *1-minimal* subset $S_2 \subseteq S_1$, which satisfies $\mathcal{R}(S_2)=\mathcal{R}(S_1)$ and $\forall\ a_i \in S_2 : \mathcal{R}(S_2 \setminus \{a_i\}) \neq \mathcal{R}(S_1)$ [28]. Intuitively, all the addresses in S_2 must be erroneously replaced with `hlt`. It can be proved by contradiction. Assuming $a_j \in S_2$ is a code byte (and hence its rewriting is correct), not replacing address a_j (with `hlt`) should not influence the execution result, that is $\mathcal{R}(S_2 \setminus \{a_j\}) = \mathcal{R}(S_2)$. As $\mathcal{R}(S_2)=\mathcal{R}(S_1)$, $\mathcal{R}(S_2 \setminus \{a_j\}) = \mathcal{R}(S_1)$, directly contradicting with the 1-minimal property. Delta debugging [28] is an efficient debugging technique that guarantees to find 1-minimal errors. It operates in a way similar to binary search. Details are elided.

D. Optimizations

We develop three optimizations for StochFuzz, which are directly performed on rewritten binaries without lifting to IR. They are reusing dead registers, removing flag register savings, and removing redundant instrumentation. Details can be found in Appendix X-A.

IV. Probabilistic Guarantees

In this section, we study the probabilistic guarantees of StochFuzz. We focus on two aspects. The first is the likelihood of rewriting errors (i.e., data bytes are mistakenly replaced with `hlt`) corrupting coverage information without triggering a crash. Note that if it triggers a crash, StochFuzz can locate and repair the error. The second is the likelihood of instruction bytes not being replaced with `hlt` so that we miss coverage information. Note there is no crash in this case but rather some instructions are invisible to our system and not rewritten. Our theoretical analysis shows that the former likelihood is 0.05% and the latter is 0.01% (with a number of conservative assumptions). They are also validated by our experiments. Details can be found in Appendix X-B.

V. Practical Challenges

We have addressed a number of practical challenges such as supporting exception handling in C++, reducing process set up cost, safeguarding non-crashing rewriting errors, and handling occluded rewriting. Details can be found in Appendix X-C.

VI. Evaluation

StochFuzz is implemented from scratch with over $10,000$ lines of C code, leveraging Capstone [39] and Keystone [40] that provide basic disassembling and assembling functionalities, respectively. Our evaluation takes more than 5000 CPU hours and is conducted on three benchmark sets, including

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Soundness on Google FTS (✗ means failure)

Program	*afl-qemu*	*ptfuzzer*	*e9patch*	*ddisasm*	StochFuzz
boringssl			✗	✗	✓
freetype2		✗		✗	✓
guetzli		✗			✓
harfbuzz				✗	✓
lcms				✗	✓
libarchive		✗			✓
libxml2	✗			✗	✓
openssl-1.0.1f			✗	✗	✓
openssl-1.0.2d			✗		✓
openssl-1.1.0c			✗	✗	✓
openthread				✗	✓
sqlite		✗			✓
wpantund				✗	✓

the Google Fuzzer Test Suite (Google FTS) [17], a variant of Google FTS which is compiled with inlined data, and the fuzzing benchmarks from *RetroWrite* [16]. We compare StochFuzz with the state-of-the-art binary-only fuzzers, including *ptfuzzer*, *afl-qemu*, *RetroWrite*, *e9patch*, and *ddisasm*. In addition, we use StochFuzz on 7 commercial binaries and find 2 zero-days. We port a recent work IJON [21] on state-based fuzzing to support stripped binaries, demonstrating StochFuzz can collect other feedback than coverage.

All the benchmarks are compiled by Clang 6.0 with their default compilation flags ("*-O2*" in most cases). For *e9path*, as it cannot recover CFG from a stripped binary, we instrument all the control flow transfer instructions (e.g., `jmp`) to trace the execution paths. For *ddisasm*, the version we use is 1.0.1, and the reassembly flags we use are "*--no-cfi-directives*" and "*--asm*". The reassembly of *ddisasm* is performed on a server equipped with a 48-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz) and 188G main memory. All others are conducted on a server equipped with a 12-cores CPU (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz) and 16G main memory.

A. Evaluation on Google FTS

Google FTS is a standard benchmark widely used to evaluate fuzzing techniques [4], [41], [42], consisting of 24 complex real-world programs. We compare StochFuzz with *ptfuzzer*, *afl-qemu*, *e9patch*, and *ddisasm*. We additionally compare with two compiler-based baselines (*afl-gcc* and *afl-clang-fast*). However, we cannot compare with *RetroWrite* on Google FTS as *RetroWrite* cannot instrument stripped binaries and it requires the binaries not written in C++, while all the binaries are stripped in this experiment and 1/3 of them are C++ ones.

Soundness. Table II presents the overall soundness of binary-only fuzzing solutions. The first column shows the programs. Columns 2-6 show whether *afl-qemu*, *ptfuzzer*, *e9patch*, *ddisasm*, and StochFuzz successfully generate binaries that the fuzzer can execute, respectively. Note that we only present the programs which at least one tool fails to instrument (due to the space limitations). Specifically, *afl-qemu* fails on *libxml2* due to a known implementation bug [43], *ptfuzzer* fails on 4 out of the 24 programs due to unsolved issues in their implementation [44], *e9patch* fails on 4 programs as these programs contain hand-written assembly code interleaved with data, *ddisasm* fails on 9 programs which crash on the seed inputs after reassembly due to uncertainty in their heuristics[1], and StochFuzz succeeds on all the 24 programs.

TABLE III: Mean and standard deviation of time-to-discovery (in minutes) for bugs in Google FTS

Tool	guetzli	json	llvm-libcxxabi
afl-gcc	513.25 ± 114.84	0.85 ± 0.63	0.08 ± 0.00
afl-clang-fast	539.56 ± 240.83	0.18 ± 0.17	0.08 ± 0.00
afl-qemu	$+\infty$	2.64 ± 3.56	0.23 ± 0.05
ptfuzzer	$+\infty$	49.08 ± 82.35	0.79 ± 0.25
e9patch	$+\infty$	21.87 ± 36.21	0.35 ± 0.00
ddisasm	505.22 ± 93.45	N/A	0.08 ± 0.00
StochFuzz	363.37 ± 120.14	0.67 ± 1.02	0.08 ± 0.00

Tool	pcre2	re2	woff2
afl-gcc	763.61 ± 40.44	2.21 ± 2.14	12.89 ± 0.44
afl-clang-fast	461.73 ± 219.89	3.08 ± 3.93	12.09 ± 4.91
afl-qemu	$+\infty$	$+\infty$	67.23 ± 26.94
ptfuzzer	$+\infty$	42.92 ± 68.08	29.18 ± 0.19
e9patch	$+\infty$	$+\infty$	30.73 ± 0.28
ddisasm	913.90 ± 495.42	N/A	14.60 ± 0.25
StochFuzz	768.91 ± 264.82	2.32 ± 0.54	7.43 ± 0.27

Fuzzing Efficiency. To assess the fuzzing efficiency achieved by StochFuzz, we run AFL to fuzz the instrumented binaries for 24 hours. Fig. 10 presents the total number of fuzzing executions, where we take *afl-gcc* as a baseline and report the ratio of each tool to *afl-gcc*. Larger numbers indicate better performance. The average numbers of fuzzing executions over the 24 programs are presented in the legend (on the top) associated with the tools. StochFuzz outperforms *afl-gcc* in 13 out of 24 programs. For the remaining 11 programs, StochFuzz also achieves comparable performance with *afl-gcc*. *Afl-clang-fast* achieves the best performance among all the tools, as it does instrumentation at the IR level. Compared with it, StochFuzz has 11.77% slowdown on average due to the additional overhead of extra control flow transfers (from the original space to the shadow space) and switching between binary versions. *Ddisasm* also achieves good performance. However, due to its inherent soundness issues, it fails on 9 out of the 24 programs. Other tools have relatively higher overhead.

Bug Finding. As Time-to-discovery (TTD) (of bugs) directly reflects fuzzing effectiveness, and hence suggests instrumentation effectiveness and fuzzing throughput, we additionally conduct an experiment to show the time needed to find the first bug for each tool. We run each tool three times with a 24-hour timeout. Table III shows the average TTD (in minutes) and the standard deviation. We only report the programs for which at least one tool can report a bug within the time bound. The first column presents the tools. Columns 2-4 show the TTDs for different programs. The symbol $+\infty$ denotes the tool cannot discover any bug within the time bound. N/A denotes the crash(es)[2] discovered by the tool cannot be reproduced

[1] After being reported to the developers of *ddisasm*, 6 out of 9 test failures got fixed in the latest release (via strengthening heuristics). Details can be found at https://github.com/GrammaTech/ddisasm/issues/20.

[2] The latest *ddisasm* can correctly reassemble all N/A programs.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

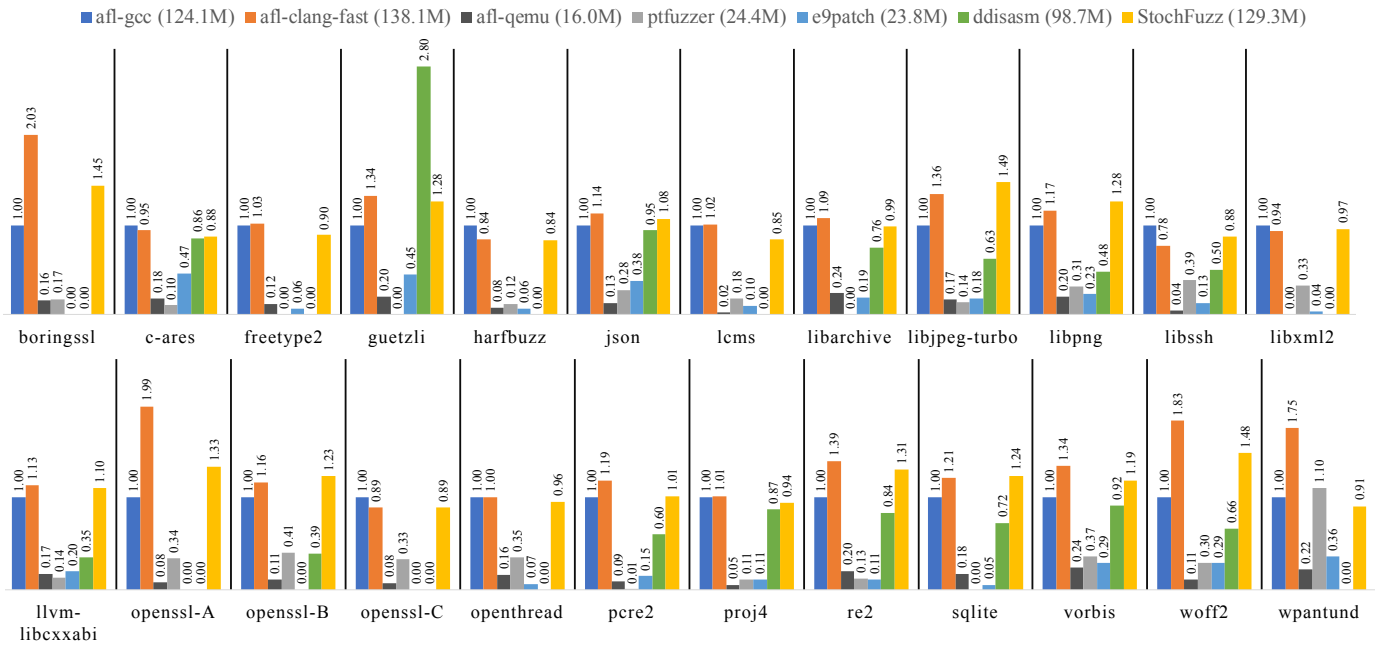


Fig. 10: Total number of fuzzing executions of each tool in 24 hours. We take *afl-gcc* as a baseline, and report the ratio of each tool to *afl-gcc*. In the legend, we additionally present the average number of fuzzing executions over the 24 programs. Larger numbers indicate better performance.

by executing the non-instrumented binary. Due to their high overhead, *afl-qemu*, *ptfuzzer*, and *e9patch* cannot discover bugs in multiple programs. Although *ddisasm* achieves good performance in the programs that it can instrument, it generates invalid crashes for some programs due to its soundness issues. StochFuzz has a similar TTD to *afl-gcc*. This shows the soundness and effectiveness of StochFuzz.

We also collect the path coverage in 24 hours. The average coverage for *afl-gcc*, *afl-clang-fast*, and StochFuzz is 2572, 2239, and 2493, respectively. As other tools do not work on all the programs, their numbers are not comparable, and hence elided. We also omit the details due to the page limitations.

Optimization Effectiveness. Table IV presents the effects of optimizations. The second column presents the number of executed blocks during fuzzing. Columns 3-4, 5-6, and 7-8 present the results for removing flag register savings (FLAG), general purpose register reuse (GPR), and removing instrumentation for single successors, respectively. For each optimization, we report both the number (of applying these optimizations) and the percentage. In the last column, we present the slow-down when the optimizations are disabled. Overall, FLAG is most effective, removing 99% of cases. Intuitively, the use of flag registers has very strong locality. We then conduct a study on the evaluated binaries and find that almost all flag registers are defined and used within the last three instructions of basic blocks, with the most common instruction pattern being a `cmp` or `test` instruction followed by a conditional jump. As such, they are mostly dead at the instrumentation points. GPR can be applied in 82.2% cases on average. The observation is that many basic blocks start with instructions that write to at least one general purpose register. StochFuzz hence is able to reuse the register in the instrumented code (Section III-D). The average percentage of instrumentation removal for blocks with a single successor is 44.49%, which is not that significant but still helpful. The slowdown is 22.45% on average when we disable these optimizations. The optimizations have negative effects on some programs such as *lcms*. Further inspection seems to indicate that the optimizations cause some tricky complications in

TABLE IV: Effects of Optimizations. #B denotes the number of basic blocks instrumented by StochFuzz, #O denotes the number of blocks where an optimization is applied at least once, %R denotes the percentage, and %S denotes the slowdown when disabling the optimizations

Program	#B	FLAG		GPR		Single-Succ		%S
		#O	%R	#O	%R	#O	%R	
boringssl	5,112	5,068	99.1	4,294	84.0	2,225	43.53	37.81
c-ares	98	96	98.0	83	84.7	48	48.98	-4.22
freetype2	13,590	13,508	99.4	11,422	84.0	6,126	45.08	1.86
guetzli	10,680	10,621	99.4	8,230	77.1	5,312	49.74	3.49
harfbuzz	10,365	10,208	98.5	8,679	83.7	4,256	41.06	28.39
json	2,308	2,296	99.5	1,886	81.7	1,125	48.74	30.48
lcms	4,256	4,181	98.2	3,341	78.5	1,712	40.23	-16.58
libarchive	7,134	7,046	98.8	5,862	82.2	2,540	35.60	33.25
libjpeg-turbo	2,953	2,927	99.1	2,609	88.4	1,362	46.12	35.48
libpng	2,815	2,797	99.4	2,173	77.2	1,153	40.96	18.07
libssh	4,441	4,393	98.9	3,578	80.6	1,816	40.89	30.43
libxml2	13,546	13,487	99.6	10,786	79.6	5,531	40.83	15.96
llvm-libcxxabi	4,257	4,244	99.7	3,314	77.8	2,171	51.00	28.77
openssl-1.0.1f	15,912	15,750	99.0	13,595	85.4	7,028	44.17	43.88
openssl-1.0.2d	2,347	2,285	97.4	2,036	86.7	961	40.95	64.24
openssl-1.1.0c	6,964	6,902	99.1	5,856	84.1	1,970	42.66	16.03
openthread	6,074	6,048	99.6	4,878	80.3	2,387	39.30	14.27
pcre2	6,889	6,798	98.7	5,863	85.1	3,292	47.79	45.86
proj4	1,983	1,915	96.6	1,443	72.8	984	49.62	3.58
re2	6,693	6,655	99.4	5,140	76.8	3,382	50.53	31.05
sqlite	24,264	24,128	99.4	20,541	84.7	11,314	46.63	38.87
vorbis	3,297	3,263	99.0	2,539	77.0	1,375	41.70	16.95
woff2	2,406	2,374	98.7	1,990	82.7	1,191	49.50	30.92
wpantund	27,549	27,146	98.5	22,765	82.6	11,587	42.06	2.10
Average	7,747	7,672	99.0	6,371	82.2	3,410	44.49	22.45

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

cache performance. It is worth pointing out that compiler based fuzzers such as *afl-gcc* and *afl-clang* directly benefit from built-in compiler optimizations, some of which have similar nature to ours. Dynamic instrumentation engines such as QEMU and PIN have their own optimizations although they typically reallocate all registers. Performing optimizations during unsound static rewriting is very risky. In contrast, optimizations work well in our context as STOCHFUZZ can fix disassembly and rewriting errors automatically.

B. Evaluation on Google FTS with Intential Data Inlining

Programs built by popular compilers (e.g., GCC and Clang) with default settings may not contain (substantial) code and data interleavings [10]. It is interesting to study the performance of various tools when substantial interleavings are present. We hence modify the compilation tool-chain of Google FTS to force `.rodata` sections to be interleaved with `.text` sections. We extract the ground-truth of data byte locations from the debugging information and then strip the binaries. *E9patch* fails on 22 out of the 24 programs, due to its assumption of no inlined data. It succeeds on two programs because they do not contain static data sections. *Ddisasm* fails on 21 programs. In contrast, STOCHFUZZ succeeds on all the programs. Details can be found in Appendix X-E.

Fuzzing Efficiency. We run the tools for 24 hours on each program. Fig. 13 (in Appendix) presents the number of fuzzing executions by our tool and its ratio over *afl-gcc*. We omit the results for other tools as inlined data do not impact their efficiency in theory. The results show that STOCHFUZZ still has comparable performance as *afl-gcc*. Moreover, our tool's efficiency has a slight degradation compared to without intentional data inlining (124.7M v/s 129.3M), due to the extra time needed to fix more rewriting errors.

Progress of Incremental and Stochastic Rewriting. We study how the numbers of false positives (FPs) (i.e., a data byte is replaced with `hlt`) and false negatives (FNs) (i.e., a code byte is not replaced with `hlt`) change over the procedure. Here, we use debugging information and the aggregated coverage information (over 24-hour fuzzing) to extract the ground-truth. In other words, we do not consider data bytes that are not accessed in the 24 hours and code bytes that are not covered in the 24 hours. Note that they have no influence on the fuzzing results and hence rewriting errors in them are irrelevant to our purpose. And as long as they are covered/accessed, STOCHFUZZ can expose and repair their rewriting errors. The results are presented in Table V. The second column presents the number of instrumented basic blocks. Columns 3-6 present the numbers of intentional crashes caused by `hlt` (indicating discovery of new code), unintentional crashes caused by rewriting errors, and unintentional crashes caused by program bugs, and their sum, respectively. The last four columns show the percentage of FN and FP at the beginning and the end of fuzzing process. Observe that at the beginning, with the initial probability analysis results, STOCHFUZZ has 11.74% FNs and 1.48% FPs on average.

TABLE V: Incremental and Stochastic Rewriting. #IC, #UCE, #UCB, and Sum denote the number of intentional crashes, unintentional crashes caused by rewriting errors, unintentional crashes caused by real bugs, and their sum, respectively. FN and FP denote false negative and false positive, respectively. "Begin" and "End" denote the beginning and end of fuzzing.

Program	Crashes				Rewriting			
					Begin		End	
	#IC	#UCE	#UCB	Sum	%FN	%FP	%FN	%FP
boringssl	114	98	0	212	12.59	6.18	0.09	0.08
c-ares	2	0	0	2	17.49	0.00	0.00	0.00
freetype2	335	10	0	461	10.58	2.47	0.03	0.05
guetzli	200	1	0	201	8.46	0.16	0.01	0.00
harfbuzz	448	5	0	453	9.25	4.64	0.04	0.14
json	80	0	0	80	14.40	0.00	0.02	0.00
lcms	137	0	0	137	16.90	0.04	0.06	0.01
libarchive	215	0	0	215	11.35	0.00	0.04	0.00
libjpeg-turbo	77	4	0	81	9.11	2.91	0.03	0.26
libpng	32	0	0	32	8.17	0.00	0.01	0.00
libssh	123	1	0	124	19.56	0.09	0.04	0.00
libxml2	315	2	0	317	8.80	0.05	0.04	0.00
llvm-libcxxabi	304	0	7,258	7,562	12.86	0.00	0.00	0.00
openssl-1.0.1f	166	45	0	211	12.29	0.50	0.18	0.01
openssl-1.0.2d	25	3	0	28	9.74	0.00	0.03	0.00
openssl-1.1.0c	183	186	0	369	11.11	2.61	0.13	0.08
openthread	19	7	0	26	13.77	0.37	0.06	0.00
pcre2	398	2	37	437	5.64	0.97	0.00	0.00
proj4	46	1	0	47	13.16	0.14	0.02	0.00
re2	133	2	0	135	18.80	0.37	0.09	0.02
sqlite	693	7	0	700	9.03	0.31	0.02	0.00
vorbis	51	7	0	58	8.74	3.25	0.04	0.10
woff2	33	19	0	52	5.31	10.45	0.02	0.04
wpantund	893	1	0	894	14.53	0.00	0.05	0.00
Average	209	17	304	535	11.74	1.48	0.04	0.03

At the end, they are reduced to almost non-existent (0.04% and 0.03%, respectively). These results are consistent with our theoretical bounds developed in Section IV. We randomly inspect some of the FPs and find that all of them are data bytes that have no effect on execution path (and hence have no negative impact on fuzzing results). Neither do they cause crashes. Also note that the FNs are at the byte level. If we look at the basic block level, STOCHFUZZ does not miss any basic blocks. In other words, in very rare cases (0.04%), it may miss the first one or two bytes in a basic block, but recognizes and instruments the following instructions. These FNs hence have no impact on fuzzing results. Also observe that the number of crashes by rewriting errors is very small (17) compared to that of intentional crashes (209). The former entails the relatively more expensive error diagnosis and repair process. It implies that most rewriting errors are fixed by observing new coverage, without triggering unintentional crashes. Fig. 11 shows how these numbers change over time for *freetype2*. Observe that they stabilize/converge quickly. The results for others are similar and elided.

C. Comparison with RetroWrite

Different from other techniques, *RetroWrite* has a number of strong prerequisites about target binaries. The binary has to contain symbols and relocation information, should not be written in C++, should not contain inlined data, and is position independent. Hence, *RetroWrite* cannot be used in

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

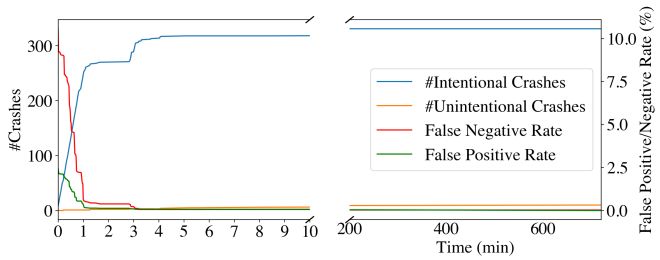


Fig. 11: Change of intentional/unintentional crashes and false positive/negative rate over time for *freetype2*

the Google FTS experiments. To compare with *RetroWrite*, we use their benchmarks that satisfy all the above conditions. Fig. 12 (in Appendix) and Table VI (in Appendix) show the numbers of fuzzing executions and the path coverage in 24 hours, respectively. StochFuzz led to 98.7M executions and *RetroWrite* led to 94.7M executions, on average. The results show StochFuzz achieves similar performance to *RetroWrite*.

VII. Case Studies

A. Finding Zero-days in Closed-source Programs

We further run StochFuzz on a set of 7 closed-source or Commercial Off-The-Shelf (COTS) binaries and find two zero-day vulnerabilities in a week. One is in CUDA Binary Utilities (by NVIDIA), a set of utilities that can extract information from CUDA binary files [45] and the other is in PNGOUT, a closed-source PNG file compressor adopted by multiple commercial or non-commercial image optimizers used in thousands of websites [46], [47]. We have reported the bugs to the vendors. The former has been fixed by NVIDIA and the latter has been assigned a CVE ID. Details can be found in Appendix X-F.

B. Collect Other Runtime Feedback Than Coverage

We conduct a case study in which we use StochFuzz to collect other runtime feedback than coverage. IJON [21], a state-aware fuzzing technique, increases fuzzing effectiveness by observing how the values of given variables change. Specifically, the tester annotates important variables in source code and the compiler instruments accesses to these variables to track their runtime changes. The changes, together with code coverage, guide input mutation. As reported in [21], it substantially improves fuzzer performance for specific kinds of programs such as complex format parsers. We port IJON to support binary-only fuzzing based on StochFuzz, and conduct the same maze experiment in the IJON paper, which was used to show the effectiveness of state-aware fuzzing. In the experiment, the target programs are games where the player has to walk through an ASCII art maze. Fuzzers instead of a human player are used to walk the mazes. IJON has advantages over vanilla fuzzers as it observes maze states and uses them to guide input mutation. The ported IJON can resolve the mazes as fast and as effective as the original source-based version, and much more effective than running IJON on *afl-qemu*. Details can be found in Appendix X-G.

VIII. Related Work

Binary-only Fuzzing. Closely related to StochFuzz is binary-only fuzzing that targets on closed-source software which has only binary executables available [12], [14]–[16], [18], [22]–[25]. As aforementioned, these works either reply on expensive operations or make impractical assumptions, limiting their wide adoption on real-world stripped binaries.

Probabilistic Analysis. Probabilistic techniques have been increasingly used in program analysis in recent years. Successful cases include symbolic execution [48], [49], model checking [50]–[52], type inference [32], etc. By introducing stochastic algorithms, those hard-to-solve problems using traditional program analysis techniques can be (partially) solved in a light-weight manner, whose correctness has probabilistic guarantees under practical assumptions. StochFuzz leverages probabilistic analysis to aggregate evidence through many sample runs and improve rewriting on-the-fly.

N-version Programming. N-version programming [53] is a software fault-tolerance technique, in which multiple variants of a program are executed in parallel and the results of individual executions are aggregated to reduce the likelihood of errors. It has been adopted to ensure memory safety [54], [55], concurrency security [56], [57], and computing correctness [58], [59], etc. UnTracer [60] continuously modifies target programs on the fly during fuzzing using source instrumentation so that they self-report when a test case causes new coverage, in order to improve fuzzing efficiency. Inspired by these works, StochFuzz also uses many versions of rewritten binaries whose validity can be approved/disapproved by numerous fuzzing runs. The difference lies that our versioning is driven by a rigorous probability analysis that updates probabilities on-the-fly. Our idea of disassembling at all addresses is inspired by Superset Disassembly [38], which however does not leverage probabilities.

IX. Conclusion

We develop a new fuzzing technique for stripped binaries. It features a novel incremental and stochastic rewriting technique that piggy-backs on the fuzzing procedure. It leverages the large number of trial-and-error chances provided by the numerous fuzzing runs to improve rewriting accuracy over time. It has probabilistic guarantees on soundness. The empirical results show that it outperforms state-of-the-art binary-only fuzzers that are either not sound or having higher overhead.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments.

The Purdue authors were supported in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, and IARPA TrojAI W911NF-19-S-0012. The RUC author was supported in part by NSFC under grants 62002361 and U1836209. The Waterloo author was supported, in part by NSERC under grant RGPIN-07017. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] https://github.com/ZhangZhuoSJTU/StochFuzz.

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016, pp. 1032–1043.

[3] "american fuzzy lop (2.52b)," https://lcamtuf.coredump.cx/afl/, 2020.

[4] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: fuzzing without valid seed inputs," in *ICSE*, 2019, pp. 712–723.

[5] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS'17*, 2017.

[6] "Processor tracing," https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, 2020.

[7] "Qemu," https://www.qemu.org/, 2020.

[8] "Pin," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2020.

[9] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[10] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security*, 2016, pp. 583–600.

[11] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," *arXiv preprint arXiv:2007.14266*, 2020.

[12] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *USENIX Security*, 2020.

[13] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[14] https://github.com/talos-vulndev/afl-dyninst.

[15] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *PLDI*, 2020, pp. 151–163.

[16] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *SP*, 2020.

[17] https://github.com/google/fuzzer-test-suite.

[18] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, 2018.

[19] https://github.com/google/AFL/tree/master/qemu_mode.

[20] https://github.com/google/AFL/tree/master/llvm_mode.

[21] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *SP*, 2020, pp. 1597–1612.

[22] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Asia CCS*, 2019, pp. 633–645.

[23] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," in *USENIX Security*, 2017, pp. 167–182.

[24] https://github.com/vanhauser-thc/afl-pin.

[25] https://github.com/vanhauser-thc/afl-dynamorio.

[26] https://github.com/GJDuck/e9afl.

[27] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *USENIX Security*, 2014, pp. 845–860.

[28] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software engineering notes*, pp. 253–267, 1999.

[29] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, and F. R. Kschischang, "The factor graph approach to model-based signal processing," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1295–1322, 2007.

[30] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 75–86, 2009.

[31] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *PLDI*, 2011, pp. 211–221.

[32] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *FSE*, 2016, pp. 607–618.

[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *OSDI*, 2006, pp. 161–176.

[34] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Generalized belief propagation," in *NIPS*, 2001, pp. 689–695.

[35] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[36] K. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," *arXiv preprint arXiv:1301.6725*, 2013.

[37] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *ICSE*, 2019, pp. 1187–1198.

[38] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[39] "The ultimate disassembler," https://www.capstone-engine.org/, 2020.

[40] "The ultimate assembler," https://www.keystone-engine.org/, 2020.

[41] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *SP*, 2019.

[42] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *USENIX Security*, 2020, pp. 2255–2269.

[43] https://github.com/AFLplusplus/AFLplusplus/issues/24.

[44] https://github.com/hunter-ht-2018/ptfuzzer.

[45] "Cuda binary utilities," https://www.clear.rice.edu/comp422/resources/cuda/html/cuda-binary-utilities/index.html, 2020.

[46] "Ewww image optimizer," https://ewww.io/, 2020.

[47] https://github.com/ImageOptim/ImageOptim.

[48] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *ISSTA*, 2012, pp. 166–176.

[49] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu, "Iterative distribution-aware sampling for probabilistic symbolic execution," in *FSE*, 2015, pp. 866–877.

[50] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *CAV*, 2011, pp. 585–591.

[51] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *ICSE*, 2011, pp. 341–350.

[52] A. F. Donaldson, A. Miller, and D. Parker, "Language-level symmetry reduction for probabilistic model checking," in *2009 Sixth International Conference on the Quantitative Evaluation of Systems*, 2009, pp. 289–298.

[53] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[54] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *PLDI*, M. I. Schwartzbach and T. Ball, Eds., 2006, pp. 158–168.

[55] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "PMP: Cost-effective forced execution with probabilistic memory pre-planning," in *SP*, 2020, pp. 381–398.

[56] J. Xu, B. Randell, A. Romanovsky, C. M. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, 1995, pp. 499–508.

[57] J. Xu, A. Romanovsky, and B. Randell, "Concurrent exception handling and resolution in distributed object systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1019–1032, 2000.

[58] J. Oberheide, E. Cooke, and F. Jahanian, "Cloudav: N-version antivirus in the network cloud," in *USENIX Security*, 2008, pp. 91–106.

[59] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *ICSE*, 2013.

[60] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *SP*, 2019, pp. 787–802.

[61] https://github.com/google/AFL/blob/master/afl-as.h#L77.

[62] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[63] https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt#L446.

[64] "Afl user guide," https://afl-1.readthedocs.io/en/latest/user_guide.html, 2020.

[65] https://www.nvidia.com/en-us/security/acknowledgements/.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

X. Appendix

A. Details of Optimizations

Register Reuse. Instrumentation may need to use registers. To avoid breaking program semantics, inside each instrumentation code block, registers need to be saved at the beginning and restored at the end. These context savings become performance bottleneck. We perform a register liveness analysis such that *dead registers*, which hold some value that will never be used in the future, can be reused in instrumentation. The difference between our liveness analysis and a traditional liveness analysis is that ours is performed on the UCFG.

Algorithm 1 presents the analysis. It takes a binary and outputs a mapping from an address i to a set of registers which are dead at i. The algorithm traverses all addresses in a descendent order (line 3). For each address i, the algorithm first collects the explicit successors of i in UCFG (line 4). If there is at least one successor whose address is smaller than i, which indicates the successor has not been analyzed (line 5), the algorithm conservatively assumes all the registers are not dead *after* i (line 6). Otherwise, the registers that are dead *at* all successors are marked as dead *after* i (line 8). At last, the dead registers *at* i are computed from the dead registers *after* i and the i instruction itself (line 10). Specifically, the registers written by i become dead (as the original values in those registers are no longer used beyond i); the ones read by i are marked live and removed from the dead set as i needs their values. Upon instrumentation, StochFuzz reuses the dead registers at the instrumentation point.

Removing Flag Register Savings. Saving and restoring flag registers has around $10\times$ more overhead compared with general purpose registers [61]. We perform the same register liveness analysis on flag registers and avoid saving/restoring the dead ones.

Removing Redundant Instrumentation. If a basic block has only one successor, its successor is guaranteed to be covered once the block is covered [62]. We hence avoid instrumenting these single successors.

B. Theoretical Analysis of Probabilistic Guarantees

Likelihood of Rewriting Error Not Causing Crash But Corrupting Coverage Feedback. If the rewriting error does not change execution path, it does not corrupt coverage feedback. In this case, we are not worried about the rewriting error even if it does not cause a crash. In other words, we are only interested in knowing the likelihood of a rewriting error changes program path but does not induce crash *over all the fuzzing runs*. Note that as long as it causes crash in one fuzzing run, StochFuzz can catch and repair it. This is the strength of having a stochastic solution. In our study, we use the following definitions.

- M: the number of fuzzing executions
- $p_{\texttt{fp}}$: the likelihood that a data byte is classified as code and subject to replacement (with `hlt`), we call it a false positive (FP).

Algorithm 1 Register Liveness Analysis on UCFG

Input:	B	binary indexed by address
Output:	$D[i] \subseteq \{r_1, r_2, ...\}$	dead registers at address i

1: **function** AnalyzeDeadReg(B)
2: D = CreateEmptyMapping()
3: **for** each address i of B in decreasing order **do**
4: $Succ = \{j \mid ExplicitSucc(i, j)\}$ ▷ $Succ = \emptyset$ if i is an indirect jump/call
5: **if** $\exists j \in Succ$, s.t. $j \leq i$ **then**
6: $d_{\text{after}} = \{\}$ ▷ Assume there is no dead variable after executing address i
7: **else**
8: $d_{\text{after}} = \bigcap_{j \in Succ} D[j]$
9: **end if**
10: $D[i] = (d_{\text{after}} \cup \{r_w \mid RegWrite(i, r_w)\}) \setminus \{r_r \mid RegRead(i, r_r)\}$
11: **end for**
12: **end function**

- $p_{\texttt{patch}} = 1 - p_\theta$: how likely a code byte (classified by StochFuzz) is selected for replacement in a rewritten binary.
- $p_{\texttt{crash}}$: the likelihood that a mistakenly replaced data byte changes program path and crashes *in a single execution*.

From the above definitions, the likelihood of a data byte is mistakenly patched is $p_{\texttt{fp}} \times p_{\texttt{patch}}$. The likelihood of a data byte being patched and triggering a crash (hence StochFuzz observes and repairs it) is $p_{\texttt{fp}} \times p_{\texttt{patch}} \times p_{\texttt{crash}}$.

The likelihood of the error escapes StochFuzz in M executions is hence the following.

$$(1 - p_{\texttt{fp}} \times p_{\texttt{patch}} \times p_{\texttt{crash}})^M$$

With a conservative setting of $p_{\texttt{fp}} = 0.015$, the average initial FP rate according to our experiment (Section VI-B, $p_{\texttt{patch}} = 0.99$, $p_{\texttt{crash}} = 0.0005$ (a very conservative setting as in practice it is over 90%), and $M = 1,000,000$, StochFuzz has 0.05% chance missing the error. We want to point out that if $p_{\texttt{crash}}$ is 0, meaning the error always changes path without crashing, StochFuzz can never detect it. We haven't seen such cases in practice. One way to mitigate the issue is to use other instructions similar to `hlt` in patching.

Likelihood of Missing Coverage Due to Code Bytes Not Being Patched. Intuitively, the likelihood is low for two reasons. First, coverage information is collected at the basic block level. Missing coverage only happens when StochFuzz misclassifies all the code bytes in a basic block to data. Second, even if StochFuzz considers a code byte is likely data, there is still a chance it is chosen for patching during stochastic rewriting. Over a large number of fuzzing runs, StochFuzz can expose it through an intentional crash.

To simplify our discussion, we only consider the second reasoning. In other words, we consider missing coverage at the byte level (not basic block level). We use the following definitions in addition to the previous ones.

- $p_{\texttt{fn}}$: the likelihood StochFuzz mis-classifies a code byte to data, called a false negative (FN).
- $p_{\texttt{exe}}$: the likelihood a code byte is covered in an execution.

The likelihood of a code byte being chosen for patching in a binary version is $(1 - p_{\texttt{fn}}) \times p_{\texttt{patch}}$. The likelihood of a code byte being patched and covered in an execution (and hence StochFuzz detects it) is $(1 - p_{\texttt{fn}}) \times p_{\texttt{patch}} \times p_{\texttt{exe}}$.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

The likelihood that the rewriting error escapes from StochFuzz in M runs is hence the following.

$$(1 - (1 - p_{\texttt{fn}}) \times p_{\texttt{patch}} \times p_{\texttt{exe}})^M$$

With a practical setting of $p_{\texttt{fn}} = 0.12$ (the average initial FN rate of StochFuzz according to our experiment), $p_{\texttt{patch}} = 0.99$, $p_{\texttt{exe}} = 1e-5$ (a very conservative setting), $M = 1,000,000$, StochFuzz has 0.01% chance missing the error. We want to point out that if $p_{\texttt{exe}}$ is 0, meaning the code byte is never executed in any runs, StochFuzz can never detect it. However, in such cases, the error has no effect on fuzzing and hence unimportant. Also note that if we consider coverage at basic block level, the bound can be lower.

C. Details of Practical Challenges

Supporting Exception Handling in C++. Exception handling in C++ poses additional challenges for static rewriting [16]. Specifically, when handling exceptions, the program needs to acquire the return addresses pushed by previous call instructions to unwind stack frames. To support this, StochFuzz additionally intercepts calls to external library functions and replace their return addresses (in the shadow space) with the corresponding addresses in the original space. Note that this is different from our transformation of call instructions to a push followed by a jump. As such, when execution returns from external libraries, it goes to the original space instead of the shadow space, incurring additional control flow transfers. To reduce the overhead, a white-list of widely-used library functions, for which we do not need to intercept the calls, is used. We argue it is a one-time effort and can be done even for closed-source programs, as the symbols of external library functions are always exposed. To understand the worst-case performance of StochFuzz, we disable the white-list optimization during evaluation.

Efficient Process Set Up. Setting up a process (e.g., linking and library initialization) has a relatively high overhead. To avoid it, a fork server, which communicates with the fuzzer through Linux pipe and forks the subject process once requested, is instrumented into the subject program by AFL [63] In StochFuzz, the dispatcher is a component of AFL, which sets up N fork servers prior to fuzzing and randomly selects one to communicate with when requesting an execution instance. Additionally, for each rewritten binary, its original and shadow spaces are both re-mmaped as shared memory with the incremental rewriter. As such, during fuzzing, the incremental and stochastic rewriting does not trigger any process set up cost.

Safeguarding Non-crashing Rewriting Errors. During fuzzing, AFL automatically monitors an input *stability* metric which measures the consistency of observed traces [64]. That is, if the subject program always behaves the same for the same input data, the fuzzing stability earns a score of 100%. A low score suggests low input consistency. This metric can help StochFuzz detect rewriting error which does not trigger a crash but changes execution trace. Specifically, once this metric becomes smaller than a given threshold, the rewriting error localization procedure is triggered. As such, the soundness guarantee of StochFuzz can be stronger than the one calculated in Section IV in practice. In our evaluation, we turn it off to measure worst-case performance.

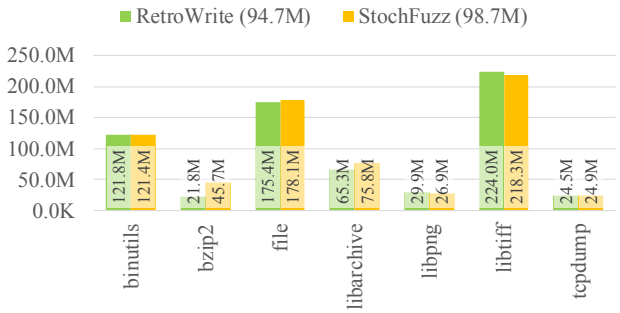


Fig. 12: The number of total fuzzing executions in 24 hours on *RetroWrite*'s fuzzing benchmarks

TABLE VI: Path Coverage on *RetroWrite*'s benchmarks

Tools	binutils	bzip2	file	libarchive	libpng	libtiff	tcpdump	Average
RetroWrite	6200	636	29	2706	977	969	3673	2170
StochFuzz	6392	1416	29	2384	928	969	3344	2209

Handling Occluded Rewriting. Another practical challenge is to handle the case in which `hlt` is mistakenly placed inside a true instruction (e.g., replacing address 1 inside the true `mov` instruction at 0 in Fig. 6). As such, the address which triggers a crash may not be the address of the inserted `hlt`. Although it is highly unlikely in practice, our crash analyzer could not repair the error properly when it happens. To handle the problem, we design a set of advanced rewriting rules, which guarantees control flow will be terminated at a set of pre-selected addresses once an occluded instruction gets executed. As such, we can infer there is an occluded rewriting error. Specifically, for a given address a with $\mathit{Inst}(a, c)$, we use the following rules to rewrite it:

1) Check whether a is occluded with any control flow transfer instruction (starting at an earlier address). If so, avoid replacing it;
2) Replace all addresses between a and a_m where $a_m = \max(\{a_i + c_i \mid \mathit{Inst}(a_i, c_i) \wedge (a_i < a < a_i + c_i)\})$, meaning the maximum end address of an instruction occluded with a.

As such, any execution that encounters an instruction occluded with some injected `hlt` must be terminated at an address in $S_c = \{a_i \mid \mathit{Inst}(a_i, c_i) \wedge (a_i < a < a_i + c_i)\} \cup [a, a_m]$. The proof of soundness is elided due to the space limitations.

D. Analysis and Rewriting Overhead on Google FTS.

Different from techniques leveraging hardware features or dynamic translation, techniques based on static rewriting incur analysis and rewriting cost. We further study such overhead on the standard Google FTS for *e9patch*, *ddisasm*, and StochFuzz. Table VII shows the results (measured by total CPU time). The second column shows the overhead of *e9patch*. The third and fourth columns show the overhead

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE VII: Analysis and Rewriting Overhead

Program	*e9patch*	*ddisasm*	*ddisasm*	StochFuzz	
		default (*-j48*)	*-j8*	rewriting	prob. anly.
boringssl	-	67h 43m 20s	126.90s	9.77s	67.35s
c-ares	0.02s	0h 47m 22s	1.17s	0.05s	0.02s
freetype2	0.76s	28h 57m 28s	96.24s	21.39s	91.59s
guetzli	0.38s	8h 51m 19s	76.05s	5.47s	95.84s
harfbuzz	0.51s	8h 02m 28s	70.89s	5.33s	64.17s
json	0.10s	4h 44m 48s	12.93s	1.30s	8.33s
lcms	0.34s	10h 39m 50s	36.58s	3.56s	13.19s
libarchive	0.51s	11h 53m 49s	61.67s	4.09s	34.29
libjpeg-turbo	0.45s	30h 16m 04s	108.79s	10.49s	24.33s
libpng	0.13s	3h 29m 24s	10.87s	1.48s	3.54s
libssh	0.36s	54h 03m 58s	50.22s	2.74s	23.98s
libxml2	2.03s	23h 52m 25s	188.59s	19.86s	177.20s
llvm-libcxxabi	0.19s	4h 33m 28s	15.57s	1.90s	19.78s
openssl-1.0.1f	-	83h 57m 03s	209.57s	22.95s	153.62s
openssl-1.0.2d	-	25h 05m 28s	37.91s	2.55s	4.82s
openssl-1.1.0c	-	117h 15m 42s	354.86s	31.57s	229.91s
openthread	0.70s	20h 24m 43s	57.96s	6.10s	13.33s
pcre2	0.33s	26h 35m 04s	481.04s	4.38s	24.38s
proj4	0.42s	10h 43m 34s	39.25s	4.69s	20.62s
re2	0.40s	17h 12m 33s	41.62s	4.60s	84.82s
sqlite	1.02s	16h 49m 43s	117.92s	14.38s	233.97s
vorbis	0.22s	16h 07m 57s	32.29s	2.26s	12.61s
woff2	0.49s	39h 09m 27s	123.50s	6.34s	21.11s
wpantund	1.58s	33h 08m 02s	176.65s	14.55s	579.94s
Average	0.55s	27h 41m 02s	105.38s	8.41s	83.41s

of *ddisasm* using different reassembly flags, and the last two columns show the overhead of StochFuzz which is broken down to rewriting and probability analysis overhead. Note that *ddisasm* uses all 48 cores by default. However, after communicating with the developers, we were notified that there are some parallelism issues with the default setting. As such, running with *-j8* (for using 8 cores) produces much better results. *E9patch* does not distinguish code and data, as it assumes exclusion of such interleavings. Hence, it has the lowest cost. Although the aggregated overheads of StochFuzz are not trivial, they are amortized over the 24 hours period. Also observe that StochFuzz's overhead is comparable to *ddisasm* (*-j8*).

E. Evaluation on Google FTS with Intential Data Inlining

Table VIII presents the overall effectiveness results for the experiment on Google FTS with intentional data inlining. The numbers of inlined data bytes are presented in the second column (i.e., data bytes in between two code sections), and whether the binaries instrumented by *e9patch*, *ddisasm*, and StochFuzz can be successfully fuzzed are presented in the next three columns, respectively. *E9patch* fails on 22 out of the 24 programs, due to its assumption of no inlined data. It succeeds on two programs because they do not contain static data sections. *Ddisasm* fails on 21 programs due to three reasons. Specifically, ✗$_1$ denotes a recompilation error that a byte value is larger than 256. It happens when *ddisasm* mis-classifies a data byte as an offset of two labels. Hence, when instrumentation code is inserted, the offset increases, making the data byte larger than 256. Symbol ✗$_2$ denotes a recompilation error that the target of a jump instruction is an integer (instead of a symbol). It happens when *ddisasm* misclassifies some data bytes as a jump instruction whose target cannot be symbolized. Symbol ✗$_3$ denotes that instrumentation code crashes on seed inputs (due to some recompilation error).

TABLE VIII: Effectiveness on Google FTS w/ Intentional Data Inlining

Program	# Inlined Data Bytes	*e9patch*	*ddisasm*	StochFuzz
boringssl	263,539	✗	✗$_3$	✓
c-ares	7	✗		✓
freetype2	91,960	✗	✗$_2$	✓
guetzli	18,543	✗	✗$_3$	✓
harfbuzz	63,061	✗	✗$_3$	✓
json	0			✓
lcms	22,576	✗	✗$_2$	✓
libarchive	55,698	✗	✗$_3$	✓
libjpeg-turbo	79,329	✗	✗$_3$	✓
libpng	9,054	✗	✗$_2$	✓
libssh	141,943	✗	✗$_3$	✓
libxml2	128,007	✗	✗$_3$	✓
llvm-libcxxabi	0			✓
openssl-1.0.1f	169,787	✗	✗$_3$	✓
openssl-1.0.2d	43,796	✗	✗$_2$	✓
openssl-1.1.0c	369,397	✗	✗$_2$	✓
openthread	32,691	✗	✗$_3$	✓
pcre2	95,763	✗	✗$_1$	✓
proj4	30,978	✗	✗$_2$	✓
re2	35,336	✗	✗$_2$	✓
sqlite	35,467	✗	✗$_3$	✓
vorbis	59,986	✗	✗$_2$	✓
woff2	494,994	✗	✗$_2$	✓
wpantund	89,203	✗	✗$_2$	✓

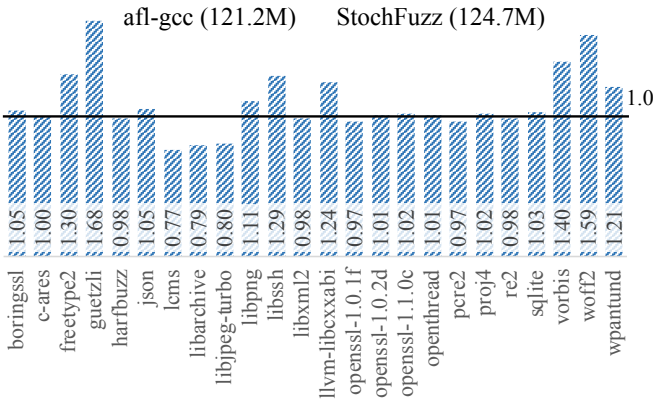


Fig. 13: Total Number of Fuzzing Executions in 24 hours. We use *afl-gcc* as the baseline, and report the ratio of StochFuzz to *afl-gcc*. In the legend, we present the average number of fuzzing executions over the 24 programs.

In contrast, StochFuzz successfully instruments and fuzzes all the programs.

F. Finding Zero-days in Closed-source Programs

In this experiment, we demonstrate StochFuzz's applicability in closed-source or COTS binaries. We run StochFuzz on a set of 7 such binaries including CUDA Toolkit (*cuobjdump*, *nvdisasm*, *cu++filt*, and *nvprune*), PNGOUT, RAR (*rar* and *unrar*) for a week. It discloses two zero-day vulnerabilities, as listed in Table IX. The first column presents the programs, and columns 2-5 present the release date of subject programs, the size, the first 4 bytes of MD5 Hash, and current bug status, respectively. CUDA Binary Utilities, developped by NVIDIA, are a set of utilities which can extract information from CUDA binary files [45]. The bug has been Fixed in CUDA 11.3 [65]. PNGOUT is a closed-source PNG

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE IX: Zero-day vulnerabilities disclosed by StochFuzz

Program	Released Date	Size	MD5	Status
CUDA Binary Utilities	2020-09-20	33M	edaf12b5	Fixed
PNGOUT	2020-01-15	89K	64f6899d	CVE-2020-29384

TABLE X: Maze Solving by Different Approaches. Three runs are performed, each with a timeout of 12 hours, according to the setting of the original paper. Symbol ✗ denotes no solution was found in any run, ✓ denotes that all runs solved the maze.

Maze		Plain	IJON-Source	IJON-Binary	
		afl-clang-fast	*afl-clang-fast*	*afl-qemu*	StochFuzz
Small	Easy	2/3	✓	✓	✓
	Hard	1/3	✓	✓	✓
Large	Easy	✗	✓	✓	✓
	Hard	✗	✓	✓	✓

file compressors, which is adopted by multiple commercial or non-commercial image optimizers [46], [47]. These optimizers are further used by thousands of website to speed up image uploading. The PNGOUT vulnerability has been assigned a CVE ID.

G. Collect Other Runtime Feedback Than Coverage

We follow the exact same setup in IJON, with two maze sizes (large and small) and two sets of rules. With the easy rule, a game is terminated once an incorrect step is taken. With the hard one, the player is allowed to backtrack. Note that in the later case, the state space is much larger. We experiment with 4 tools, *afl-clang-fast* without IJON plugin, *afl-clang-fast* with IJON plugin, binary-only *afl-qemu* with ported IJON plugin, and binary-only StochFuzz with ported IJON plugin. We run each tool three times with a 12-hour timeout. Table X shows the overall effectiveness. The first column presents the different mazes under different rules. Columns 2-5 denote whether the maze is solved by the 4 different tools, respectively. *Afl-clang-fast* solves the small maze with the easy rule 2 out of 3 trials, and the small maze with the hard rule 1 out of 3 trials. The other tools successfully solve all the mazes. Table XI shows the average time (in minutes) needed to solve the mazes and the standard deviation. Observe that although *afl-clang-fast* can solve some small mazes, it takes the longest time. Regarding the two binary-only approaches, StochFuzz is around $8\times$ faster than *afl-qemu*. Additionally, StochFuzz only has around 8% slowdown compared with *afl-clang-fast* plus IJON, which demonstrates the capabilities of StochFuzz.

H. One-step Sum-product Algorithm

Algorithm 2 describes the one-step sum-product inference procedure. O_{code} and O_{data} denote the aggregated code and data observation values for each address, respectively. Note that *a small value means strong belief*. Line 3 performs the deterministic inference. Line 9 identifies SCCs and transforms UCFG to a DAG of SCCs. Step 1 in lines 13-20 propagates

TABLE XI: Different approaches are solving the small / large maze. The tables shows the average time-to-solve in minutes $\pm$ the standard deviation.

Maze		Plain	IJON-Source	IJON-Binary	
		afl-clang-fast	*afl-clang-fast*	*afl-qemu*	StochFuzz
Small	Easy	95.42 ± 40.47	1.52 ± 0.45	20.96 ± 10.56	1.64 ± 0.51
	Hard	149.78 ± 0.0	0.46 ± 0.09	3.85 ± 1.90	0.52 ± 0.06
Large	Easy	-	20.66 ± 9.19	150.28 ± 30.27	22.94 ± 14.49
	Hard	-	5.31 ± 1.59	96.85 ± 16.61	5.12 ± 1.89

Algorithm 2 One-step Sum-product

Input:	B	binary indexed by address
Output:	$P[a] \in [0,1]$	probability of address a holding a data byte
Local:	$G = (V, E)$	$V = \{a \mid \exists c \text{ s.t. } Inst(a,c)\}$
		$E = \{(a_1, a_2) \mid ExplicitSucc(a_1, a_2)\}$
	$O_{code}[a] \in [0,1]$	aggregated code observations on address a
	$O_{data}[a] \in [0,1]$	aggregated data observations on address a

1: **function** CalcProbability(B)
2: G = BuildUCFG(B)
3: O_{code}, O_{data} = CollectObservations(B)
4: P = OneStepSumProduct(G, O_{code}, O_{data})
5: **return** P
6: **end function**
7:
8: **function** OneStepSumProduct($G, O_{\text{code}}, O_{\text{data}}$)
9: G_{DAG} = TransfromIntoDAG(G) ▷ Transform G into a Directed Acyclic Graph (DAG) via collapsing each Strongly Connected Component (SCC) into a vertex
10: $O_{\text{DAG\_code}}$ = CreateEmptyMapping() ▷ DAG-related mapping, initialized as empty
11:
12: ▷ **Step 1**: Aggregate code observation values
13: **for** each SCC x **in** topological order of G_{DAG} **do**
14: o_1 = PRODUCT($\{O_{\text{DAG\_code}}[y] \mid$ SCC y is a predecessor of SCC $x\}$)
15: o_2 = PRODUCT($\{O_{\text{code}}[i] \mid$ address i belongs to SCC $x\}$)
16: $O_{\text{DAG\_code}}[x] = o_1 \times o_2$
17: **for** each address i in all addresses belonging to SCC x **do**
18: $O_{\text{code}}[i] = O_{\text{DAG\_code}}[x]$
19: **end for**
20: **end for**
21: ▷ **Step 2**: Aggregate data observation values
22: $i_{prev} = \infty$ ▷ The last address whose $O_{data} > 0$
23: **for** each address i of B in increasing order **do**
24: **if** $O_{data}[i] \neq \perp$ **then**
25: **if** $1 < i - i_{prev} < D$ **then**
26: $o_1, o_2 = O_{data}[i], O_{data}[i_{prev}]$
27: **for** each address $j \in (i_{prev}, i)$ **do**
28: $O_{data}[j] = 1 - \frac{p_{data}}{2p_{data} + (1-o_1)(1-o_2) - 2p_{data}(1-o_1)(1-o_2)}$
29: **end for**
30: **end if**
31: $i_{prev} = i$
32: **end if**
33: **end for**
34: ▷ **Step 3**: one-step sum-product for each address
35: **for** each address i of B in increasing order **do**
36: $o_{neg} = (O_{code}[i] = \perp \ ? \ 0.5 \ : \ O_{code}[i])$
37: $o_{pos} = (O_{data}[i] = \perp \ ? \ 0.5 \ : \ O_{data}[i])$
38: $P[i] = o_{neg} \cdot (1 - o_{pos}) \ / \ (o_{pos} \cdot (1 - o_{neg}) + o_{neg} \cdot (1 - o_{pos}))$
39: **end for**
40: **return** P
41: **end function**

code observations. Step 2 in lines 22-33 propagates data observations. The formula in line 28 is derived from a simple factor graph involving three variables (i.e., addresses i, i_{prev}, and j), and three factors (for $O_{data}[i_{\text{prev}}]$, $O_{data}[i]$, and rule ⑦). Details are elided. Step 3 in lines 35-39 performs the one-step sum-product for each address. Lines 36 and 37 assign observation value 0.5 if there is no belief propagated to the address. The formula in line 38 is derived from that in Fig. 9.

Authorized licensed use limited to: Renmin University. Downloaded on September 24,2021 at 10:06:24 UTC from IEEE Xplore. Restrictions apply.