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

中国人民大學
RENMIN UNIVERSITY OF CHINA

UNIVERSITY OF WATERLOO

Grey-box fuzzing is one of the most important techniques for software testing and vulnerability detection.

Grey-box fuzzing is one of the most important techniques for software testing and vulnerability detection.

Bug Detection

- More than **21,000** bugs in the Chromium projects [1]
- More than **16,000** bugs in other open source projects [2]

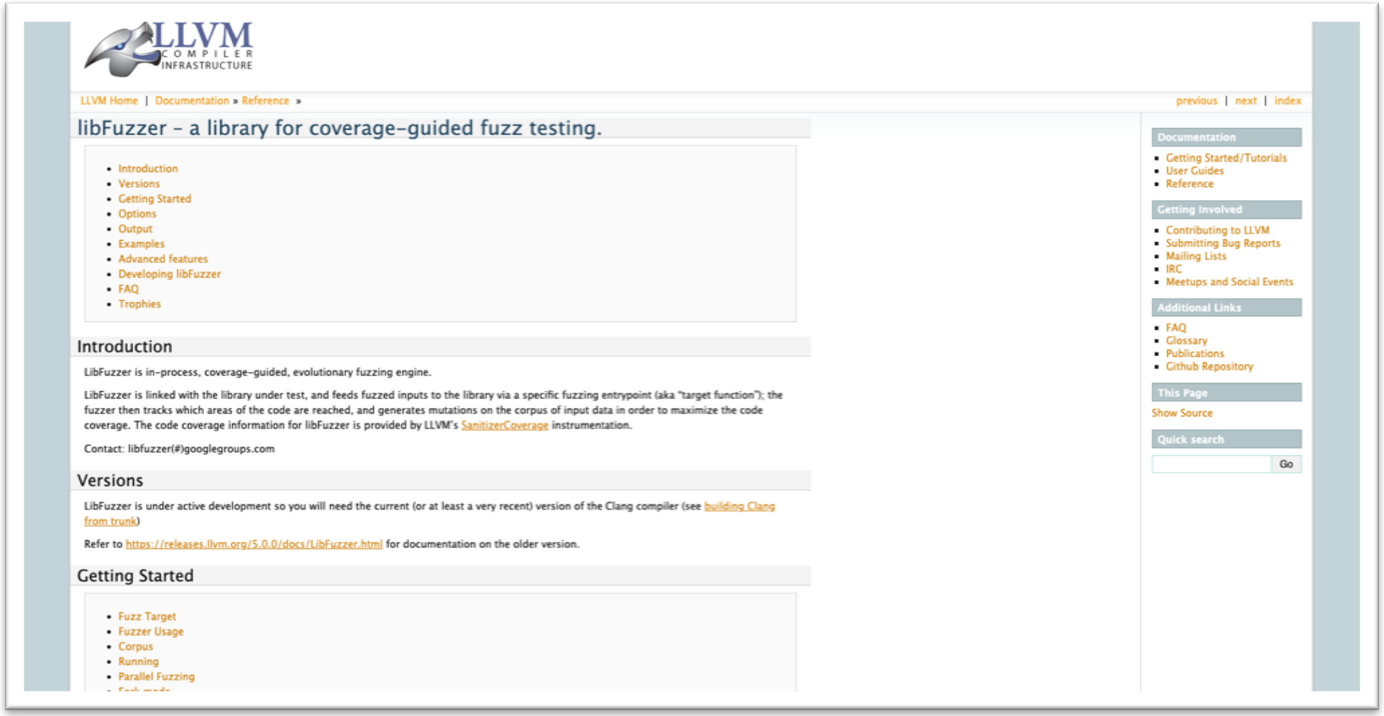
Research

- **79** Papers published in the top security conferences in the recent three years [3]
- **56** Papers published in the top software engineering conferences in the recent three years [3]

[1] https://bugs.chromium.org/p/chromium/issues/list?can=1&q=label%3AClusterFuzz+-status%3AWontFix%2CDuplicate&colspec=ID+Pri+M+Stars+ReleaseBlock+Component+Status+Owner+Summary+OS+Modified&x=m&y=releaseblock&cells=ids
[2] https://bugs.chromium.org/p/oss-fuzz/issues/list?can=1&q=-status%3AWontFix%2CDuplicate+-Infra&colspec=ID+Type+Component+Status+Proj+Reported+Owner+Summary&cells=ids
[3] https://wcventure.github.io/FuzzingPaper/

Grey-box fuzzing leverages runtime feedback to learn how to reach deeper into the subject program.

Grey-box fuzzing leverages runtime feedback to learn how to reach deeper into the subject program.



libFuzzer – a library for coverage-guided fuzz testing.

- Introduction
- Versions
- Getting Started
- Options
- Output
- Examples
- Advanced features
- Developing libFuzzer
- FAQ
- Trophies

Introduction

LibFuzzer is in-process, coverage-guided, evolutionary fuzzing engine.

LibFuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint (aka "target function"); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. The code coverage information for libFuzzer is provided by LLVM's SanitizerCoverage instrumentation.

Contact: libfuzzer(#)googlegroups.com

Versions

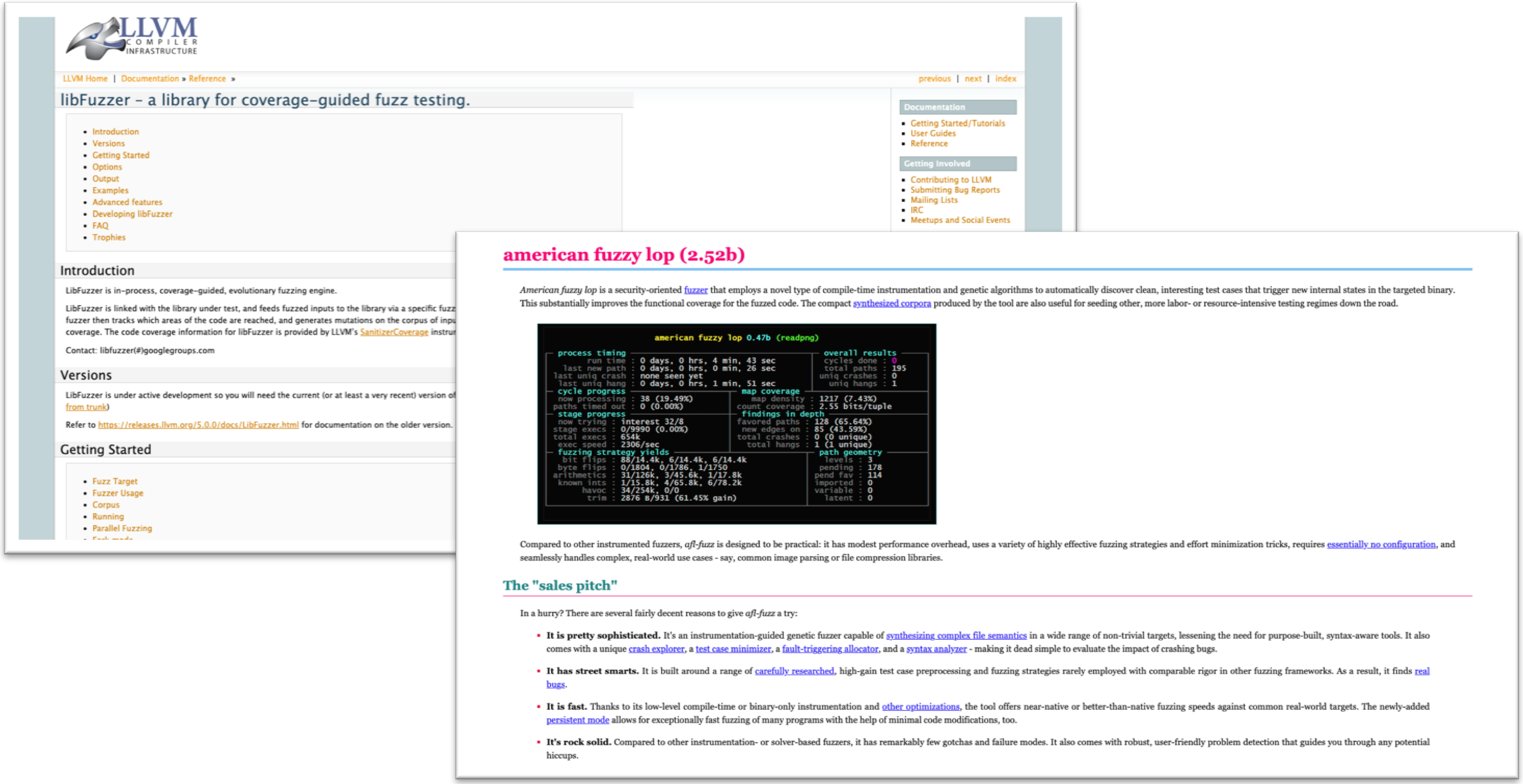
LibFuzzer is under active development so you will need the current (or at least a very recent) version of the Clang compiler (see building Clang from trunk)

Refer to https://releases.llvm.org/5.0.0/docs/LibFuzzer.html for documentation on the older version.

Getting Started

- Fuzz Target
- Fuzzer Usage
- Corpus
- Running
- Parallel Fuzzing

Grey-box fuzzing leverages runtime feedback to learn how to reach deeper into the subject program.



libFuzzer – a library for coverage-guided fuzz testing.

- Introduction
- Versions
- Getting Started
- Options
- Output
- Examples
- Advanced features
- Developing libFuzzer
- FAQ
- Trophies

Introduction

LibFuzzer is in-process, coverage-guided, evolutionary fuzzing engine.

LibFuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzz
fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of inpu
coverage. The code coverage information for libFuzzer is provided by LLVM's SanitizerCoverage instrur

Contact: libfuzzer(#)googlegroups.com

Versions

LibFuzzer is under active development so you will need the current (or at least a very recent) version of
from trunk)

Refer to https://releases.llvm.org/5.0.0/docs/LibFuzzer.html for documentation on the older version.

Getting Started

- Fuzz Target
- Fuzzer Usage
- Corpus
- Running
- Parallel Fuzzing

american fuzzy lop (2.52b)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.

```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec      │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec      │  total paths : 195    │
│ last uniq crash : none seen yet                     │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec      │   uniq hangs : 1      │
├─ cycle progress ────────────────┬─ map coverage ────┴───────────────────────┤
│  now processing : 38 (19.49%)   │    map density : 1217 (7.43%)             │
│ paths timed out : 0 (0.00%)     │ count coverage : 2.55 bits/tuple          │
├─ stage progress ────────────────┼─ findings in depth ───────────────────────┤
│  now trying : interest 32/8     │ favored paths : 128 (65.64%)              │
│ stage execs : 0/9990 (0.00%)    │  new edges on : 85 (43.59%)               │
│ total execs : 654k              │ total crashes : 0 (0 unique)              │
│  exec speed : 2306/sec          │   total hangs : 1 (1 unique)              │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ───────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k        │    levels : 3             │
│  byte flips : 0/1804, 0/1786, 1/1750            │   pending : 178           │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k         │  pend fav : 114           │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k         │  imported : 0             │
│       havoc : 34/254k, 0/0                      │  variable : 0             │
│        trim : 2876 B/931 (61.45% gain)          │    latent : 0             │
└─────────────────────────────────────────────────┴───────────────────────────┘
```

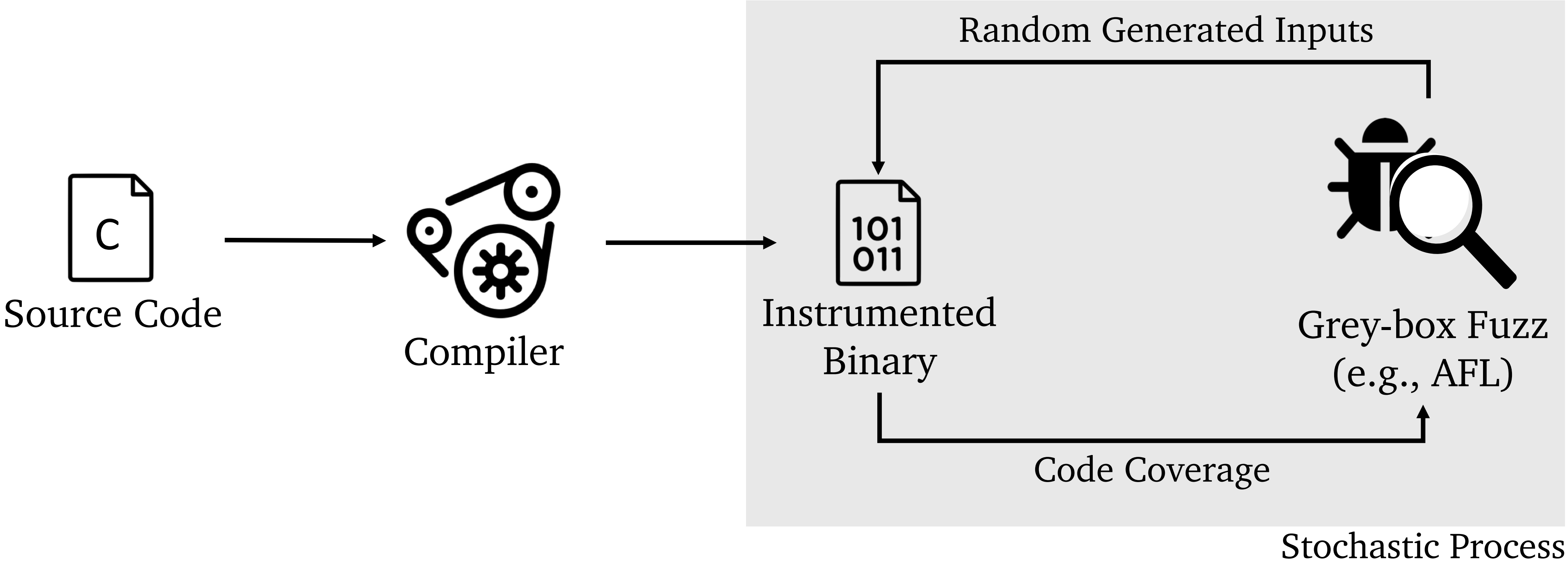
Compared to other instrumented fuzzers, *afl-fuzz* is designed to be practical: it has modest performance overhead, uses a variety of highly effective fuzzing strategies and effort minimization tricks, requires essentially no configuration, and seamlessly handles complex, real-world use cases - say, common image parsing or file compression libraries.

The "sales pitch"

In a hurry? There are several fairly decent reasons to give *afl-fuzz* a try:

- **It is pretty sophisticated.** It's an instrumentation-guided genetic fuzzer capable of synthesizing complex file semantics in a wide range of non-trivial targets, lessening the need for purpose-built, syntax-aware tools. It also comes with a unique crash explorer, a test case minimizer, a fault-triggering allocator, and a syntax analyzer - making it dead simple to evaluate the impact of crashing bugs.
- **It has street smarts.** It is built around a range of carefully researched, high-gain test case preprocessing and fuzzing strategies rarely employed with comparable rigor in other fuzzing frameworks. As a result, it finds real bugs.
- **It is fast.** Thanks to its low-level compile-time or binary-only instrumentation and other optimizations, the tool offers near-native or better-than-native fuzzing speeds against common real-world targets. The newly-added persistent mode allows for exceptionally fast fuzzing of many programs with the help of minimal code modifications, too.
- **It's rock solid.** Compared to other instrumentation- or solver-based fuzzers, it has remarkably few gotchas and failure modes. It also comes with robust, user-friendly problem detection that guides you through any potential hiccups.

Grey-box fuzzing leverages runtime feedback to learn how to reach deeper into the subject program.



Source Code

Compiler

Instrumented Binary

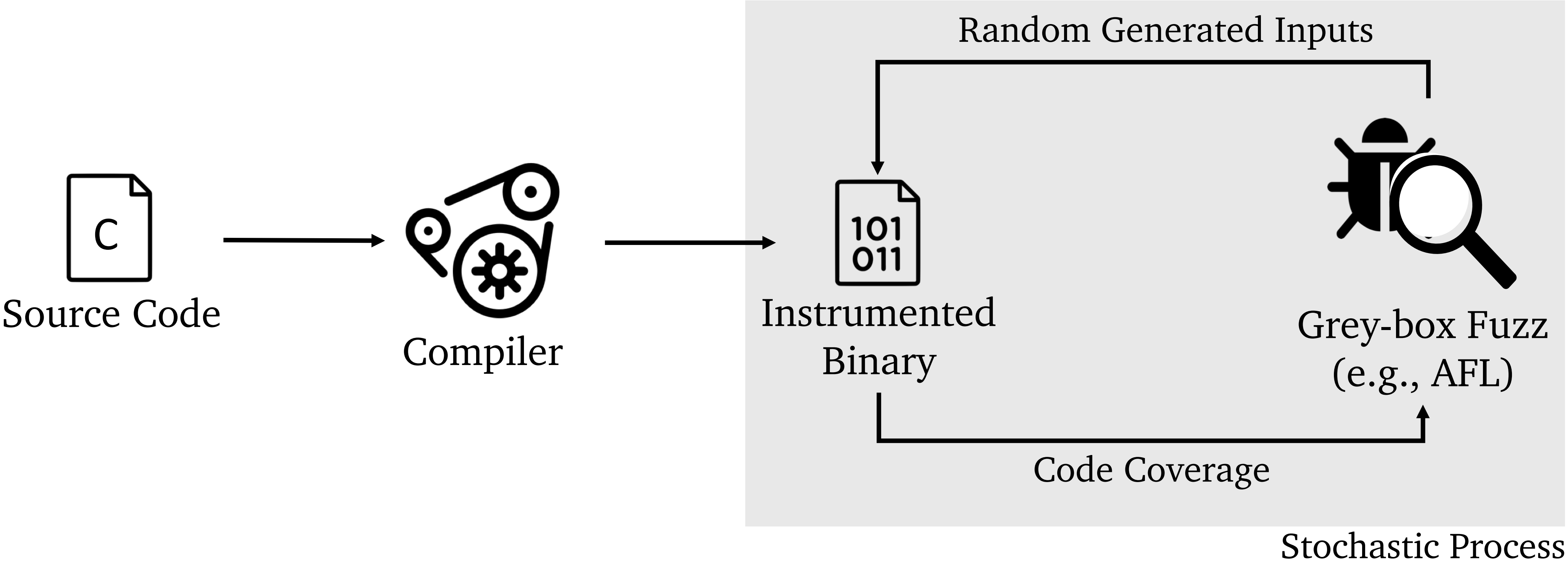
Grey-box Fuzz (e.g., AFL)

Random Generated Inputs

Code Coverage

Stochastic Process

Another scenario: *binary-only fuzzing* (no source code)



Source Code

Compiler

Instrumented Binary

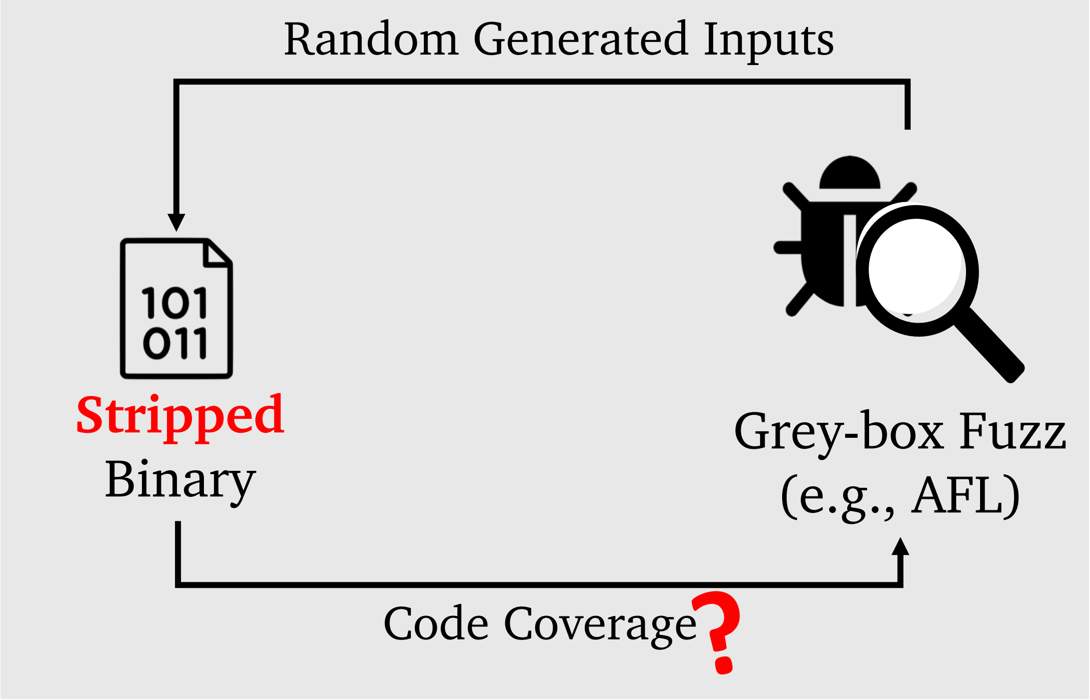
Grey-box Fuzz (e.g., AFL)

Random Generated Inputs

Code Coverage

Stochastic Process

Another scenario: *binary-only fuzzing* (no source code)



Stochastic Process

Another scenario: *binary-only fuzzing* (no source code)

- Bugs in close-sourced programs can also have unprecedented impact (e.g., WannaCry ransomware attack).
- It is important to effectively detect bugs in programs without source.

Payment will be raised on
5/16/2017 00:47:55
Time Left
02:23:57:37

Your files will be lost on
5/20/2017 00:47:55
Time Left
06:23:57:37

What Happened to My Computer?

Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

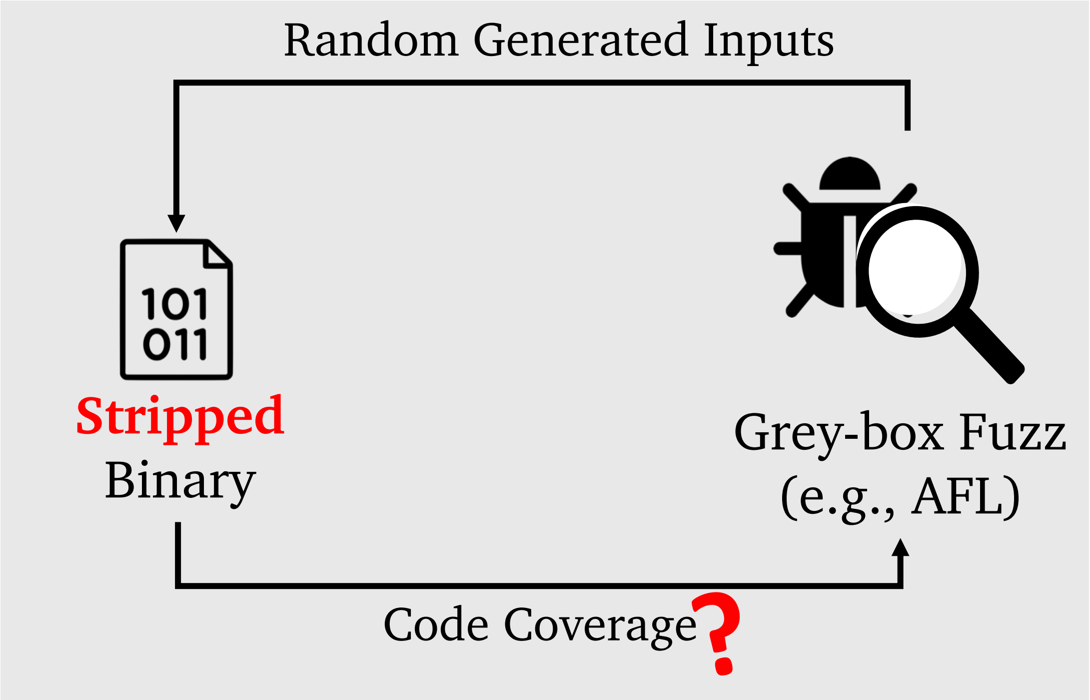
Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?

Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Send $300 worth of bitcoin to this address:
12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw



Stochastic Process

Existing solutions fall into three categories.

Existing solutions fall into three categories.

Dynamic Binary Translation: Translate a subject binary during its execution. It is sound but expensive (high overhead **>600%**).

Existing solutions fall into three categories.

Dynamic Binary Translation: Translate a subject binary during its execution. It is sound but expensive (high overhead **>600%**).

Hardware-Assisted Tracing: Make use of advanced hardware support such as Intel PT to collect runtime traces that can be post-processed (Relatively high overhead and **only coverage-based feedback**).

Existing solutions fall into three categories.

Dynamic Binary Translation: Translate a subject binary during its execution. It is sound but expensive (high overhead **>600%**).

Hardware-Assisted Tracing: Make use of advanced hardware support such as Intel PT to collect runtime traces that can be post-processed (Relatively high overhead and **only coverage-based feedback**).

Static Binary Instrumentation: Leverage advanced binary analysis to directly instrument binaries (**cost-effective** but **usually unsound**).

Existing solutions fall into three categories.

Dynamic Binary Translation: Translate a subject binary during its execution. It is sound but expensive (high overhead **>600%**).

Hardware-Assisted Tracing: Make use of advanced hardware support such as Intel PT to collect runtime traces that can be post-processed (Relatively high overhead and **only coverage-based feedback**).

Static Binary Instrumentation: Leverage advanced binary analysis to directly instrument binaries (**cost-effective** but **usually unsound**).

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note
`0:   lea rax, [rip+8]`	rax	15	.DATA

Challenges of static binary rewriting

```
   .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

   .DATA:
15:   .long 8

   .CODE2:
23:   ret
```

Inst	Var	Val	Note
0: lea rax, [rip+8]	rax	15	.DATA
7: mov rbx, [rax]	rbx	8	.CODE2-.DATA

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note
`0: lea rax, [rip+8]`	rax	15	.DATA
`7: mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10: add rax, rbx`	rax	23	.CODE2

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note
`0: lea rax, [rip+8]`	rax	15	.DATA
`7: mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10: add rax, rbx`	rax	23	.CODE2
`13: jmp rax`	jmp	.CODE2	-

Challenges of static binary rewriting

```
   .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

   .DATA:
15:   .long 8

   .CODE2:
23:   ret
```

Inst	Var	Val	Note
`0:  lea rax, [rip+8]`	rax	15	.DATA
`7:  mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10: add rax, rbx`	rax	23	.CODE2
`13: jmp rax`	jmp	.CODE2	-
`23: ret`	-	-	-

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note
0: lea rax, [rip+8]	rax	15	.DATA
7: mov rbx, [rax]	rbx	8	.CODE2-.DATA
10: add rax, rbx	rax	23	.CODE2
13: jmp rax	jmp	.CODE2	-
23: ret	-	-	-

Challenges of static binary rewriting

```
    .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

    .DATA:
15:   .long 8

    .CODE2:
23:   ret
```

Inst	Var	Val	Note
`0:   lea rax, [rip+8]`	rax	15	.DATA
`7:   mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10:  add rax, rbx`	rax	23	.CODE2
`13:  jmp rax`	jmp	.CODE2	-
`23:  ret`	-	-	-

- **Identify the interleaved data section**: due to the inline data (`.DATA`), rewriters may not only mis-rewrite data as code, but also fail to identify the indirect jump target (`.CODE2`).
- **Distinguish between scalars and the address offsets**: misclassifying an address offset (`.CODE2-.DATA`) as a scalar may break the rewritten binaries (note that addresses have changed after instrumentation).

Challenges of static binary rewriting

```
  .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   or  [rax], al
17:   add [rax], al
19:   add [rax], al
21:   add [rax], al
```

Inst	Var	Val	Note
0: lea rax, [rip+8]	rax	15	.DATA
7: mov rbx, [rax]	rbx	8	.CODE2-.DATA
10: add rax, rbx	rax	23	.CODE2
13: jmp rax	jmp	.CODE2	-
23: ret	-	-	-

- **Identify the interleaved data section**: due to the inline data (.DATA), rewriters may not only mis-rewrite data as code, but also fail to identify the indirect jump target (.CODE2).
- **Distinguish between scalars and the address offsets**: misclassifying an address offset (.CODE2-.DATA) as a scalar may break the rewritten binaries (note that addresses have changed after instrumentation).

Challenges of static binary rewriting

```
   .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   or  [rax], al
17:   add [rax], al
19:   add [rax], al
21:   add [rax], al
```

Inst	Var	Val	Note
`0: lea rax, [rip+8]`	rax	15	.DATA
`7: mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10: add rax, rbx`	rax	23	.CODE2
`13: jmp rax`	jmp	.CODE2	-
`23: ret`	-	-	-

- **Identify the interleaved data section**: due to the inline data (.DATA), rewriters may not only mis-rewrite data as code, but also fail to identify the indirect jump target (.CODE2).
- **Distinguish between scalars and the address offsets**: misclassifying an address offset (.CODE2-.DATA) as a scalar may break the rewritten binaries (note that addresses have changed after instrumentation).

Challenges of static binary rewriting

```
   .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   or  [rax], al
17:   add [rax], al
19:   add [rax], al
21:   add [rax], al
```

[redacted]

Inst	Var	Val	Note
0: lea rax, [rip+8]	rax	15	.DATA
7: mov rbx, [rax]	rbx	8	[redacted]
10: add rax, rbx	rax	23	.CODE2
13: jmp rax	jmp	???	-
23: ret	-	-	-

- **Identify the interleaved data section**: due to the inline data (.DATA), rewriters may not only mis-rewrite data as code, but also fail to identify the indirect jump target (.CODE2).
- **Distinguish between scalars and the address offsets**: misclassifying an address offset (.CODE2-.DATA) as a scalar may break the rewritten binaries (note that addresses have changed after instrumentation).

Challenges of static binary rewriting

```
   .CODE1:
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax

   .DATA:
15:   .long 8

   .CODE2:
23:   ret
```

RetroWrite, e9patch, and datalog disassembly (the version before we reported the issue) all fail on a similar case.

Inst	Var	Val	Note
`0: lea rax, [rip+8]`	rax	15	.DATA
`7: mov rbx, [rax]`	rbx	8	.CODE2-.DATA
`10: add rax, rbx`	rax	23	.CODE2
`13: jmp rax`	jmp	.CODE2	-
`23: ret`	-	-	-

- **Identify the interleaved data section**: due to the inline data (`.DATA`), rewriters may not only mis-rewrite data as code, but also fail to identify the indirect jump target (`.CODE2`).
- **Distinguish between scalars and the address offsets**: misclassifying an address offset (`.CODE2`-`.DATA`) as a scalar may break the rewritten binaries (note that addresses have changed after instrumentation).

How we handle the motivation case: **Incremental Rewriting**

How we handle the motivation case: **Incremental Rewriting**

The first technique we introduced is named **Incremental Rewriting**.

- 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*.

How we handle the motivation case: **Incremental Rewriting**

The first technique we introduced is named **Incremental Rewriting**.

- 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*.

Our basic idea is to trigger an intentional crash once an unresolved control flow target is reached. Starting from the address where the crash happens, we can incrementally rewrite all directly reachable addresses.

The fuzzer continues fuzzing with the new binary and the incremental rewriting is invoked again if other intentional crashes occur.

How we handle the motivation case: **Incremental Rewriting**

How we handle the motivation case: Incremental Rewriting

```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

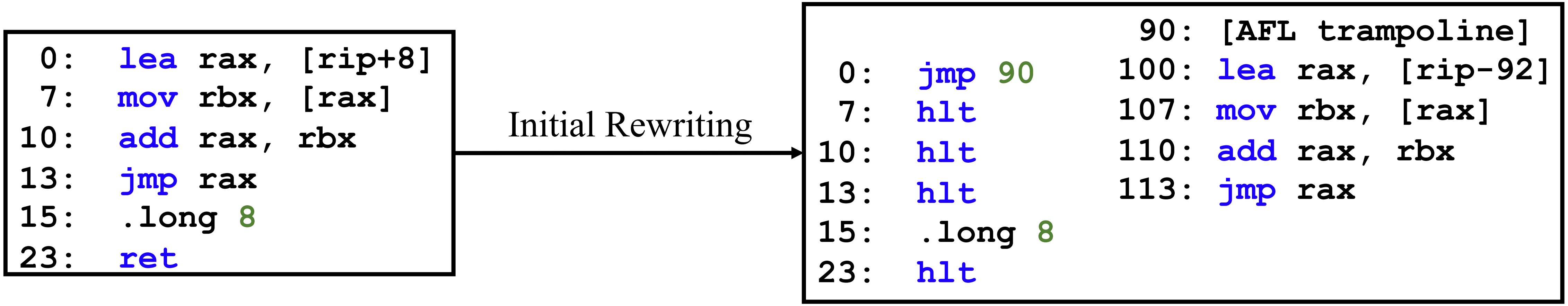
How we handle the motivation case: Incremental Rewriting

```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

For easy understanding, let’s first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: **Incremental Rewriting**



```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

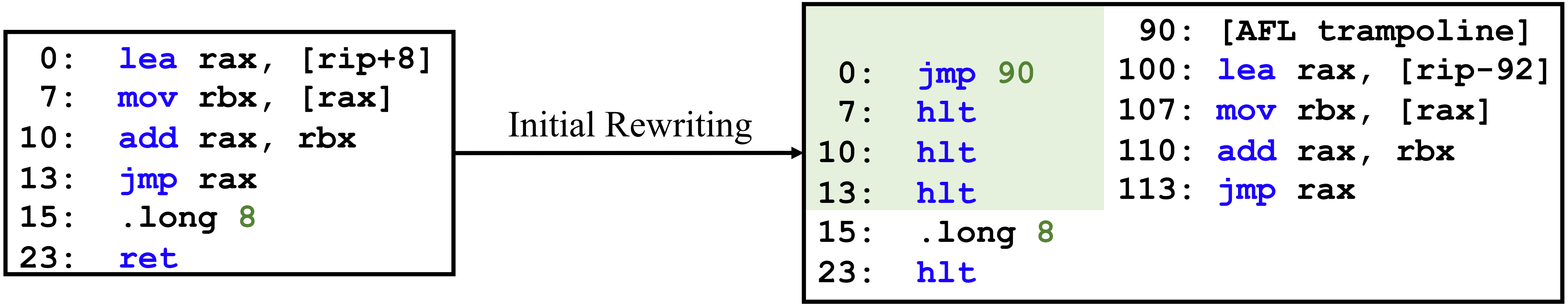
Initial Rewriting

```
                          90: [AFL trampoline]
 0:   jmp 90             100: lea rax, [rip-92]
 7:   hlt                107: mov rbx, [rax]
10:   hlt                110: add rax, rbx
13:   hlt                113: jmp rax
15:   .long 8
23:   hlt
```

For easy understanding, let's first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: **Incremental Rewriting**



```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

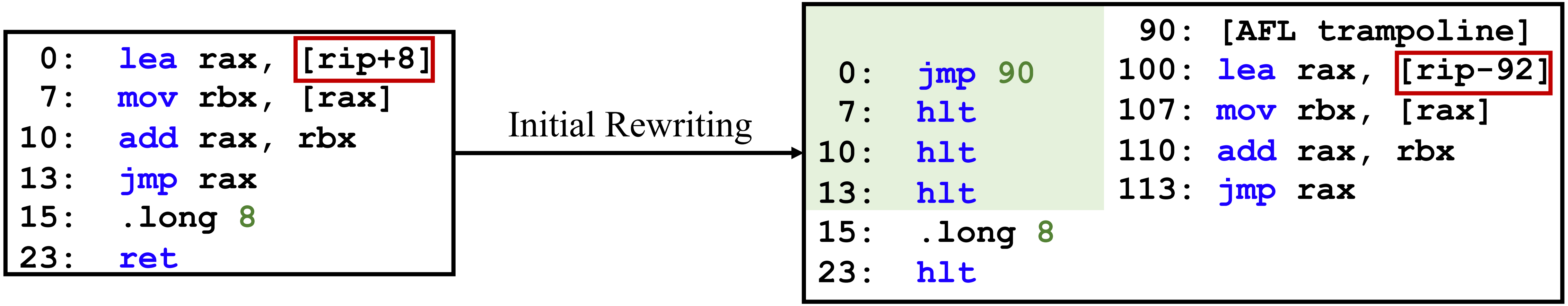
Initial Rewriting

```
                          90:  [AFL trampoline]
 0:   jmp 90             100:  lea rax, [rip-92]
 7:   hlt                107:  mov rbx, [rax]
10:   hlt                110:  add rax, rbx
13:   hlt                113:  jmp rax
15:   .long 8
23:   hlt
```

For easy understanding, let’s first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: Incremental Rewriting



```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

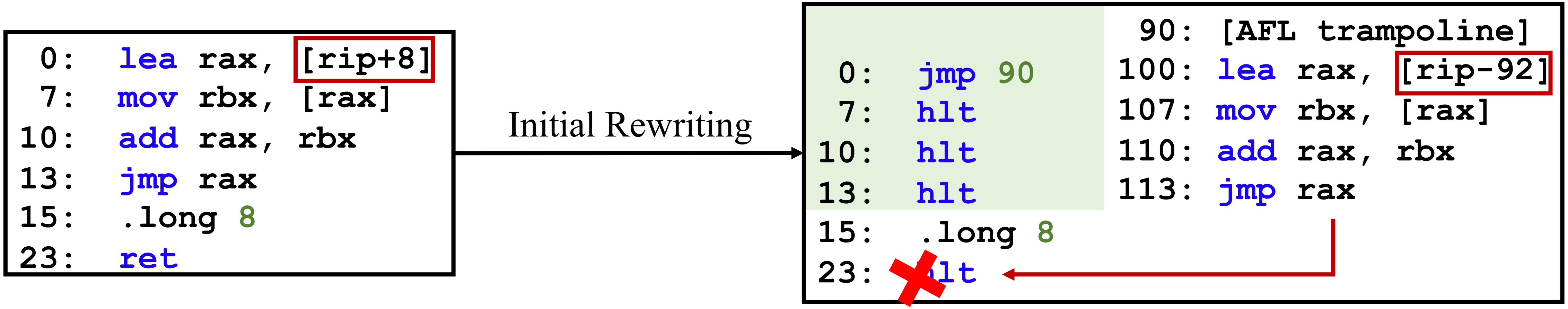
Initial Rewriting

```
                      90:  [AFL trampoline]
 0:   jmp 90         100:  lea rax, [rip-92]
 7:   hlt            107:  mov rbx, [rax]
10:   hlt            110:  add rax, rbx
13:   hlt            113:  jmp rax
15:   .long 8
23:   hlt
```

For easy understanding, let’s first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: Incremental Rewriting



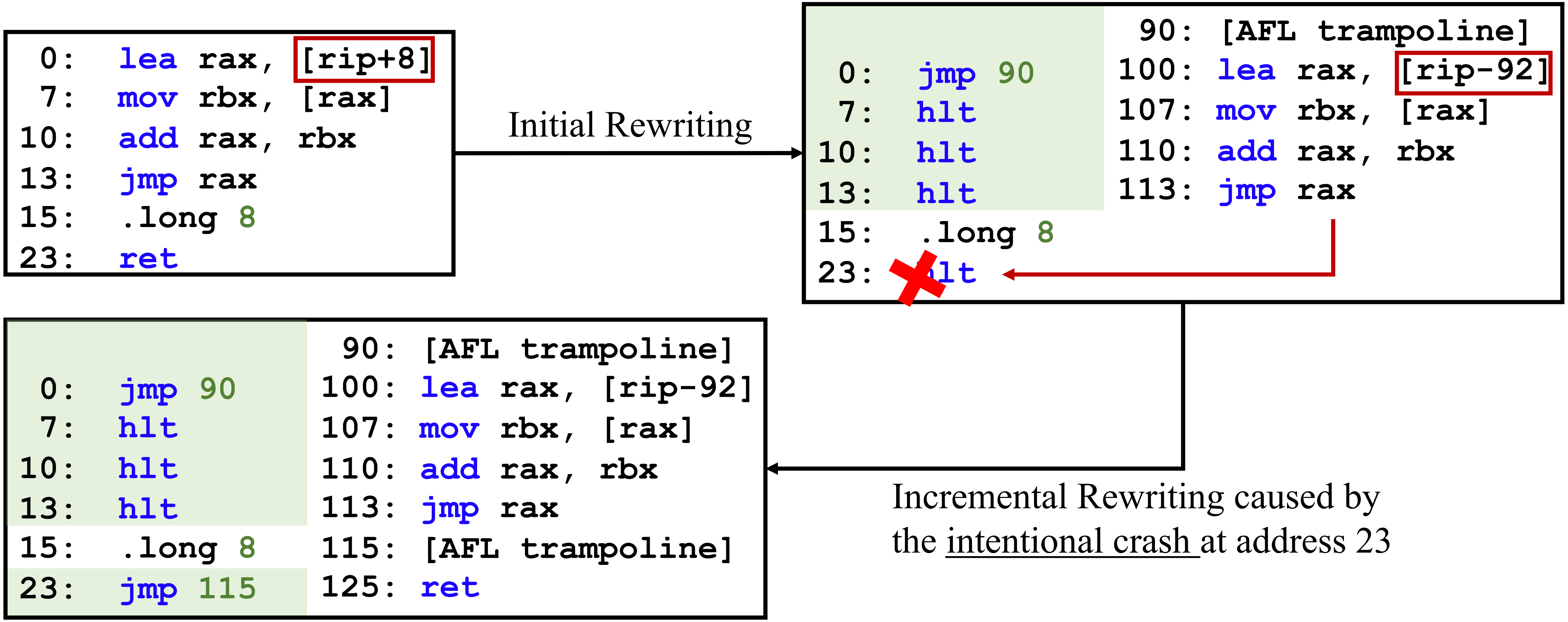
```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

```
                     90: [AFL trampoline]
 0:   jmp 90        100: lea rax, [rip-92]
 7:   hlt           107: mov rbx, [rax]
10:   hlt           110: add rax, rbx
13:   hlt           113: jmp rax
15:   .long 8
23:   hlt
```

For easy understanding, let’s first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: **Incremental Rewriting**



```
 0:   lea rax, [rip+8]
 7:   mov rbx, [rax]
10:   add rax, rbx
13:   jmp rax
15:   .long 8
23:   ret
```

Initial Rewriting

```
                  90:  [AFL trampoline]
 0:   jmp 90     100:  lea rax, [rip-92]
 7:   hlt        107:  mov rbx, [rax]
10:   hlt        110:  add rax, rbx
13:   hlt        113:  jmp rax
15:   .long 8
23:   hlt
```

Incremental Rewriting caused by the intentional crash at address 23

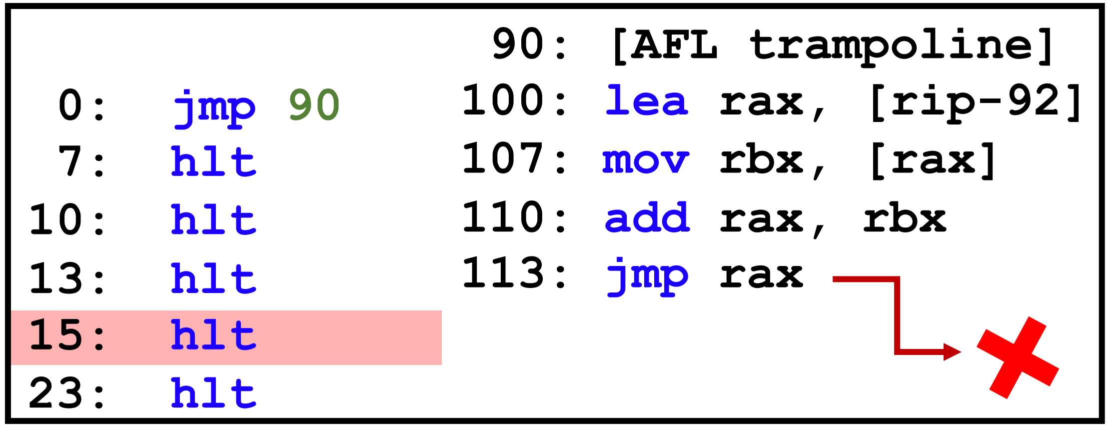
```
                  90:  [AFL trampoline]
 0:   jmp 90     100:  lea rax, [rip-92]
 7:   hlt        107:  mov rbx, [rax]
10:   hlt        110:  add rax, rbx
13:   hlt        113:  jmp rax
15:   .long 8    115:  [AFL trampoline]
23:   jmp 115    125:  ret
```

For easy understanding, let's first assume:

- Our underlying binary analysis cannot find the indirect jump target (address 23 `.CODE2)`.
- We can 100% accurately distinguish code and data (later, I will explain what if this assumption does not hold).

How we handle the motivation case: **Stochastic Rewriting**

How we handle the motivation case: **Stochastic Rewriting**



```
                        90: [AFL trampoline]
 0:   jmp 90           100: lea rax, [rip-92]
 7:   hlt              107: mov rbx, [rax]
10:   hlt              110: add rax, rbx
13:   hlt              113: jmp rax
15:   hlt
23:   hlt
```

How we handle the motivation case: Stochastic Rewriting

The second technique we introduced is named **Stochastic Rewriting**.

- During fuzzing, we can try different data and code separations.
- More samples we collect, more precise separation we can have.

How we handle the motivation case: Stochastic Rewriting

The second technique we introduced is named **Stochastic Rewriting**.

- During fuzzing, we can try different data and code separations.
- More samples we collect, more precise separation we can have.

Stochastic Rewriting is piggy-backing on the fuzzing procedure.

- A probabilistic inference to compute the likelihood of each byte being data (or code)
- Generating different binaries for different fuzzing runs
- A error diagnosis process to locate and repair rewriting errors

How we handle the motivation case: Stochastic Rewriting

The second technique we introduced is named **Stochastic Rewriting**.

- During fuzzing, we can try different data and code separations.
- More samples we collect, more precise separation we can have.

Stochastic Rewriting is piggy-backing on the fuzzing procedure.

- A probabilistic inference to compute the likelihood of each byte being data (or code)
- Generating different binaries for different fuzzing runs
- A error diagnosis process to locate and repair rewriting errors

How we handle the motivation case: Stochastic Rewriting

The second technique we introduced is named **Stochastic Rewriting**.

- During fuzzing, we can try different data and code separations.
- More samples we collect, more precise separation we can have.

Stochastic Rewriting is piggy-backing on the fuzzing procedure.

- A probabilistic inference to compute the likelihood of each byte being data (or code)
- Generating different binaries for different fuzzing runs
- A error diagnosis process to locate and repair rewriting errors

How we handle the motivation case: Stochastic Rewriting

The second technique we introduced is named **Stochastic Rewriting**.

- During fuzzing, we can try different data and code separations.
- More samples we collect, more precise separation we can have.

Stochastic Rewriting is piggy-backing on the fuzzing procedure.

- A probabilistic inference to compute the likelihood of each byte being data (or code)
- Generating different binaries for different fuzzing runs
- A error diagnosis process to locate and repair rewriting errors

How we handle the motivation case: **Stochastic Rewriting**

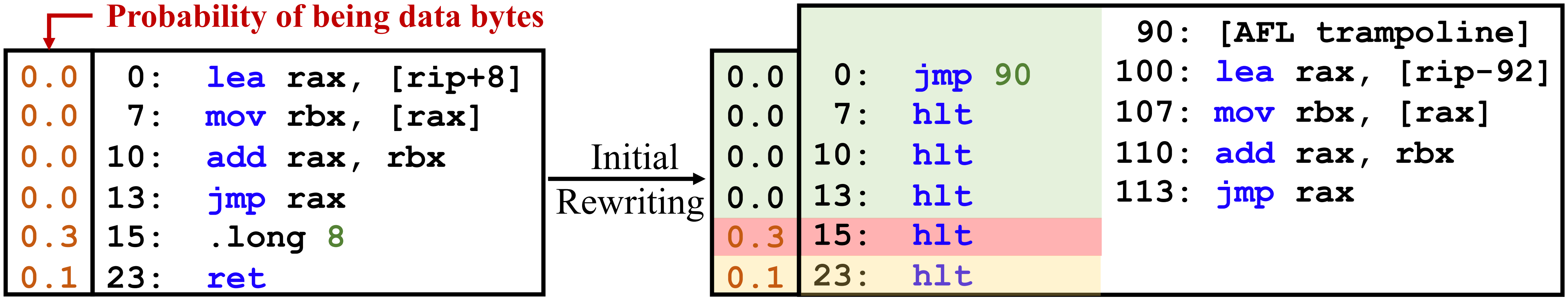
```
 0:  lea rax, [rip+8]
 7:  mov rbx, [rax]
10:  add rax, rbx
13:  jmp rax
15:  .long 8
23:  ret
```

How we handle the motivation case: **Stochastic Rewriting**

Probability of being data bytes

```
0.0 |  0:   lea rax, [rip+8]
0.0 |  7:   mov rbx, [rax]
0.0 | 10:   add rax, rbx
0.0 | 13:   jmp rax
0.3 | 15:   .long 8
0.1 | 23:   ret
```

How we handle the motivation case: Stochastic Rewriting



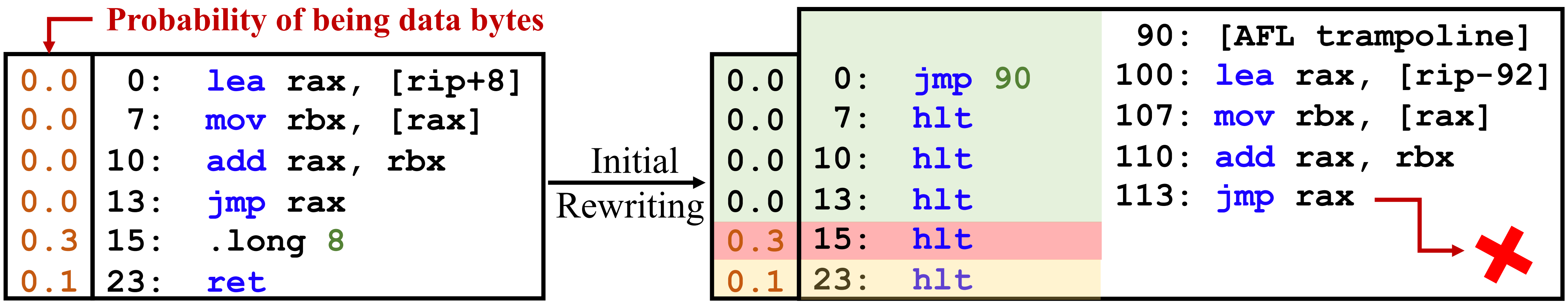
Probability of being data bytes

```
0.0   0:   lea rax, [rip+8]
0.0   7:   mov rbx, [rax]
0.0  10:   add rax, rbx
0.0  13:   jmp rax
0.3  15:   .long 8
0.1  23:   ret
```

Initial Rewriting

```
0.0   0:   jmp 90            90:  [AFL trampoline]
0.0   7:   hlt              100:  lea rax, [rip-92]
0.0  10:   hlt              107:  mov rbx, [rax]
0.0  13:   hlt              110:  add rax, rbx
0.3  15:   hlt              113:  jmp rax
0.1  23:   hlt
```

How we handle the motivation case: Stochastic Rewriting



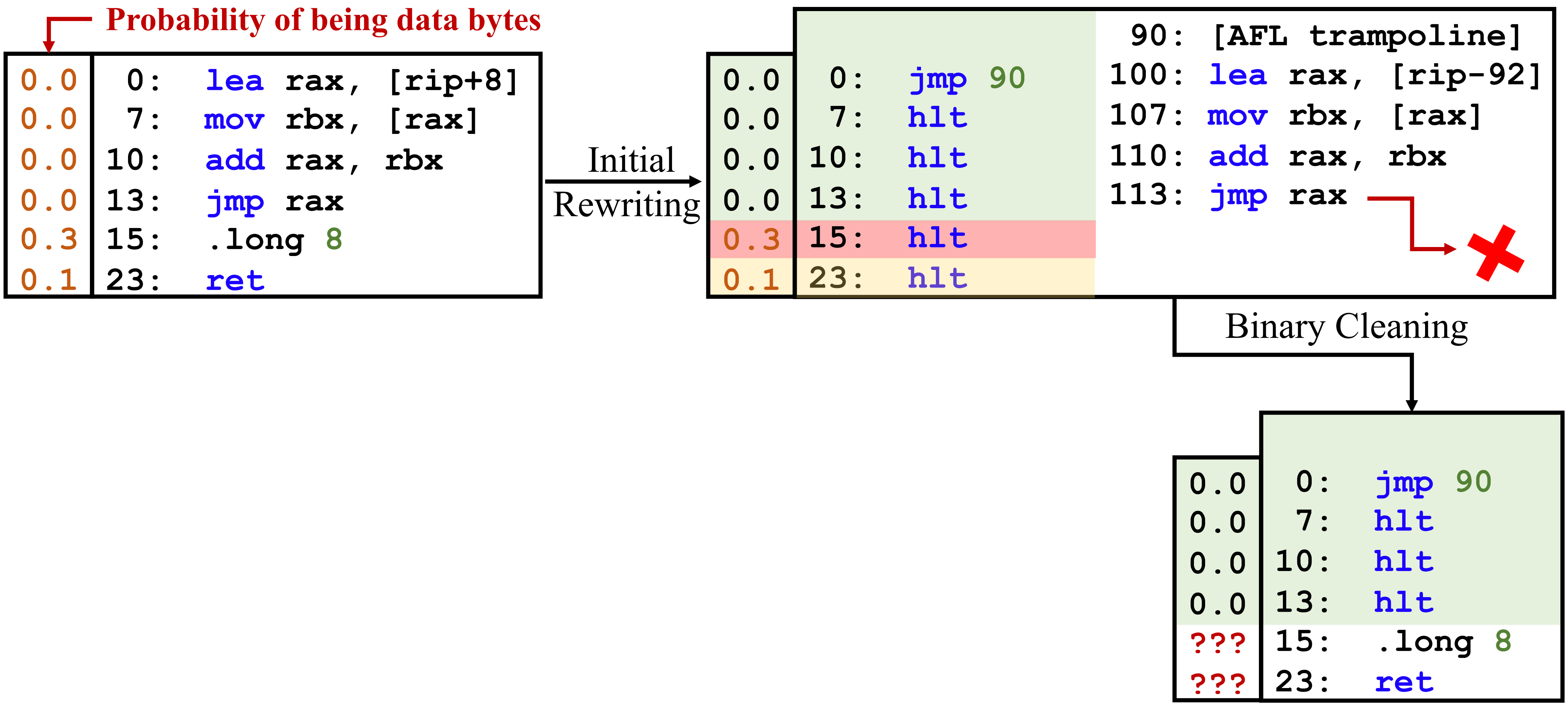
Probability of being data bytes

```
0.0   0:   lea rax, [rip+8]
0.0   7:   mov rbx, [rax]
0.0  10:   add rax, rbx
0.0  13:   jmp rax
0.3  15:   .long 8
0.1  23:   ret
```

Initial Rewriting

```
                         90:  [AFL trampoline]
0.0   0:   jmp 90       100:  lea rax, [rip-92]
0.0   7:   hlt          107:  mov rbx, [rax]
0.0  10:   hlt          110:  add rax, rbx
0.0  13:   hlt          113:  jmp rax
0.3  15:   hlt
0.1  23:   hlt
```

How we handle the motivation case: Stochastic Rewriting



Probability of being data bytes

```
0.0    0:  lea rax, [rip+8]
0.0    7:  mov rbx, [rax]
0.0   10:  add rax, rbx
0.0   13:  jmp rax
0.3   15:  .long 8
0.1   23:  ret
```

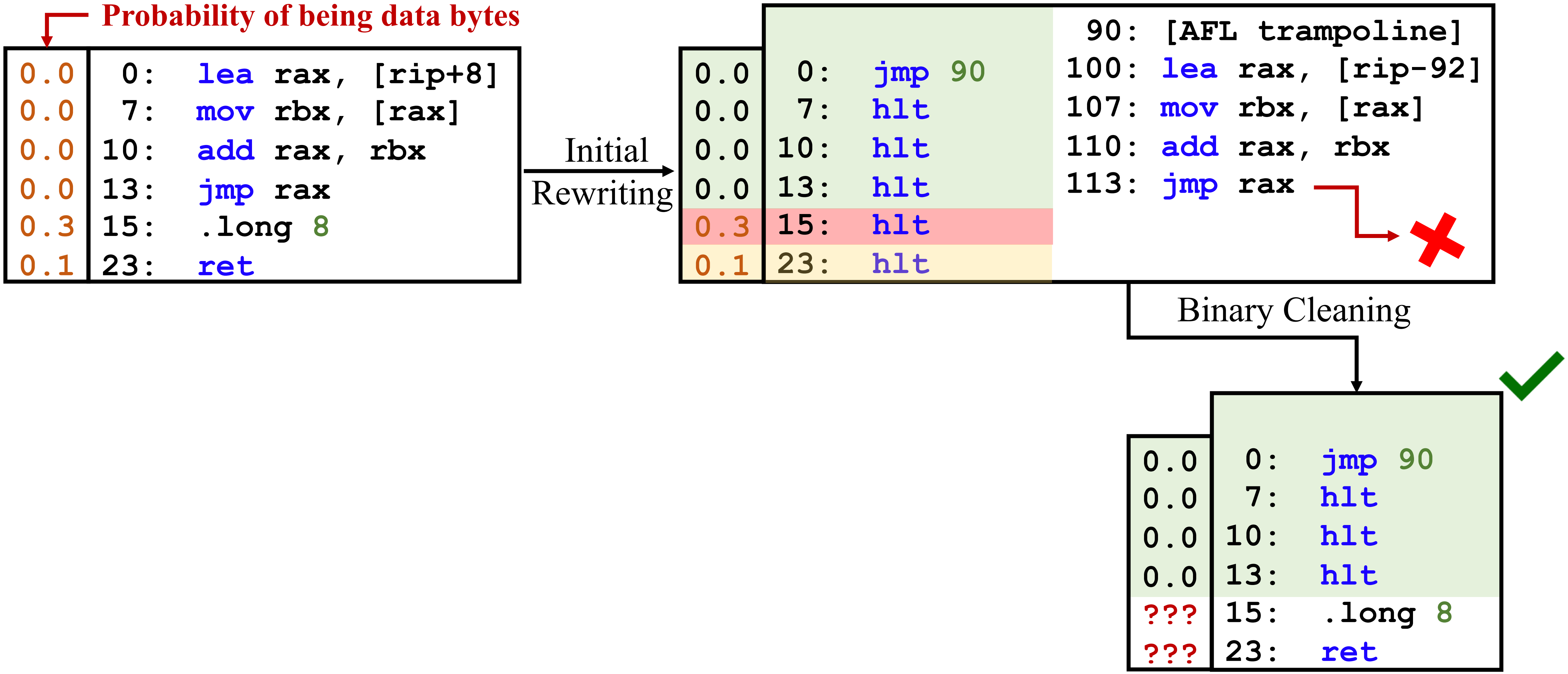
Initial Rewriting

```
0.0    0:  jmp 90        90:  [AFL trampoline]
0.0    7:  hlt          100:  lea rax, [rip-92]
0.0   10:  hlt          107:  mov rbx, [rax]
0.0   13:  hlt          110:  add rax, rbx
0.3   15:  hlt          113:  jmp rax
0.1   23:  hlt
```

Binary Cleaning

```
0.0    0:  jmp 90
0.0    7:  hlt
0.0   10:  hlt
0.0   13:  hlt
???   15:  .long 8
???   23:  ret
```

How we handle the motivation case: Stochastic Rewriting



Probability of being data bytes

```
0.0    0:   lea rax, [rip+8]
0.0    7:   mov rbx, [rax]
0.0   10:   add rax, rbx
0.0   13:   jmp rax
0.3   15:   .long 8
0.1   23:   ret
```

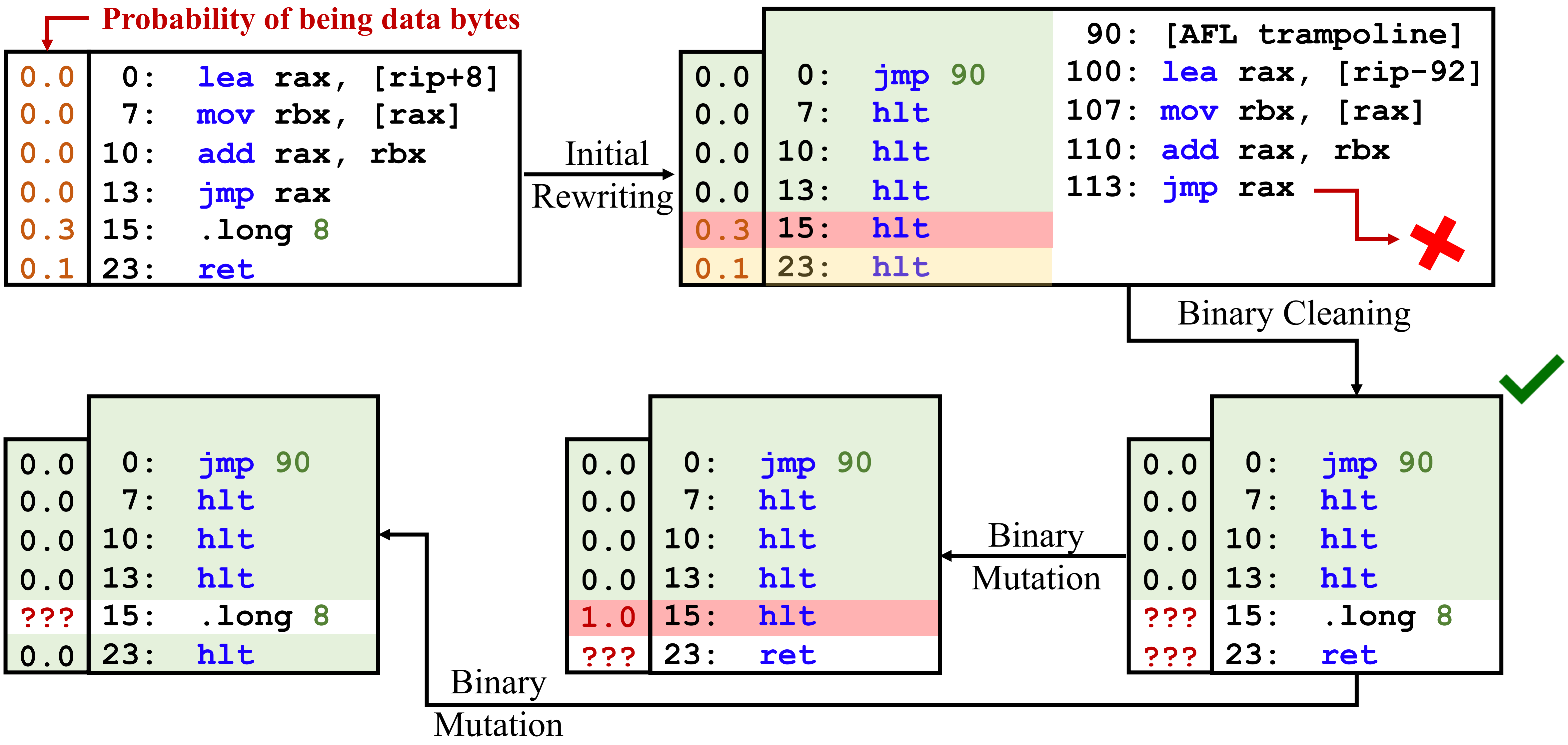
Initial Rewriting

```
0.0    0:   jmp 90          90:  [AFL trampoline]
0.0    7:   hlt            100:  lea rax, [rip-92]
0.0   10:   hlt            107:  mov rbx, [rax]
0.0   13:   hlt            110:  add rax, rbx
0.3   15:   hlt            113:  jmp rax
0.1   23:   hlt
```

Binary Cleaning

```
0.0    0:   jmp 90
0.0    7:   hlt
0.0   10:   hlt
0.0   13:   hlt
???   15:   .long 8
???   23:   ret
```

How we handle the motivation case: Stochastic Rewriting



Probability of being data bytes

```
0.0   0:  lea rax, [rip+8]
0.0   7:  mov rbx, [rax]
0.0  10:  add rax, rbx
0.0  13:  jmp rax
0.3  15:  .long 8
0.1  23:  ret
```

Initial Rewriting

```
0.0   0:  jmp 90          90:  [AFL trampoline]
0.0   7:  hlt            100:  lea rax, [rip-92]
0.0  10:  hlt            107:  mov rbx, [rax]
0.0  13:  hlt            110:  add rax, rbx
0.3  15:  hlt            113:  jmp rax
0.1  23:  hlt
```

Binary Cleaning

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
???  15:  .long 8
???  23:  ret
```

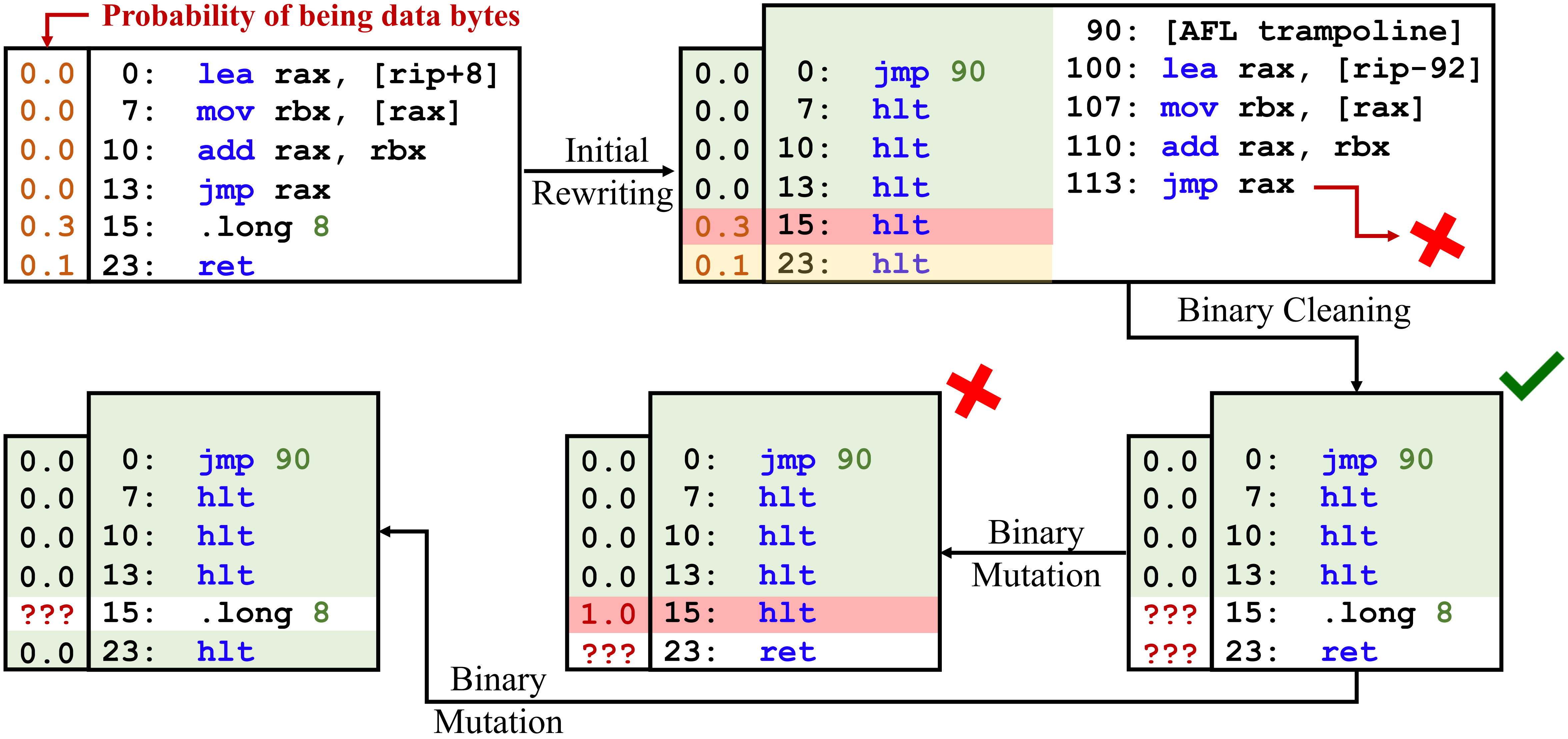
Binary Mutation

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
1.0  15:  hlt
???  23:  ret
```

Binary Mutation

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
???  15:  .long 8
0.0  23:  hlt
```

How we handle the motivation case: **Stochastic Rewriting**



Probability of being data bytes

```
0.0   0:   lea rax, [rip+8]
0.0   7:   mov rbx, [rax]
0.0  10:   add rax, rbx
0.0  13:   jmp rax
0.3  15:   .long 8
0.1  23:   ret
```

Initial Rewriting

```
0.0   0:   jmp 90          90:  [AFL trampoline]
0.0   7:   hlt            100:  lea rax, [rip-92]
0.0  10:   hlt            107:  mov rbx, [rax]
0.0  13:   hlt            110:  add rax, rbx
0.3  15:   hlt            113:  jmp rax
0.1  23:   hlt
```

Binary Cleaning

```
0.0   0:   jmp 90
0.0   7:   hlt
0.0  10:   hlt
0.0  13:   hlt
???  15:   .long 8
???  23:   ret
```

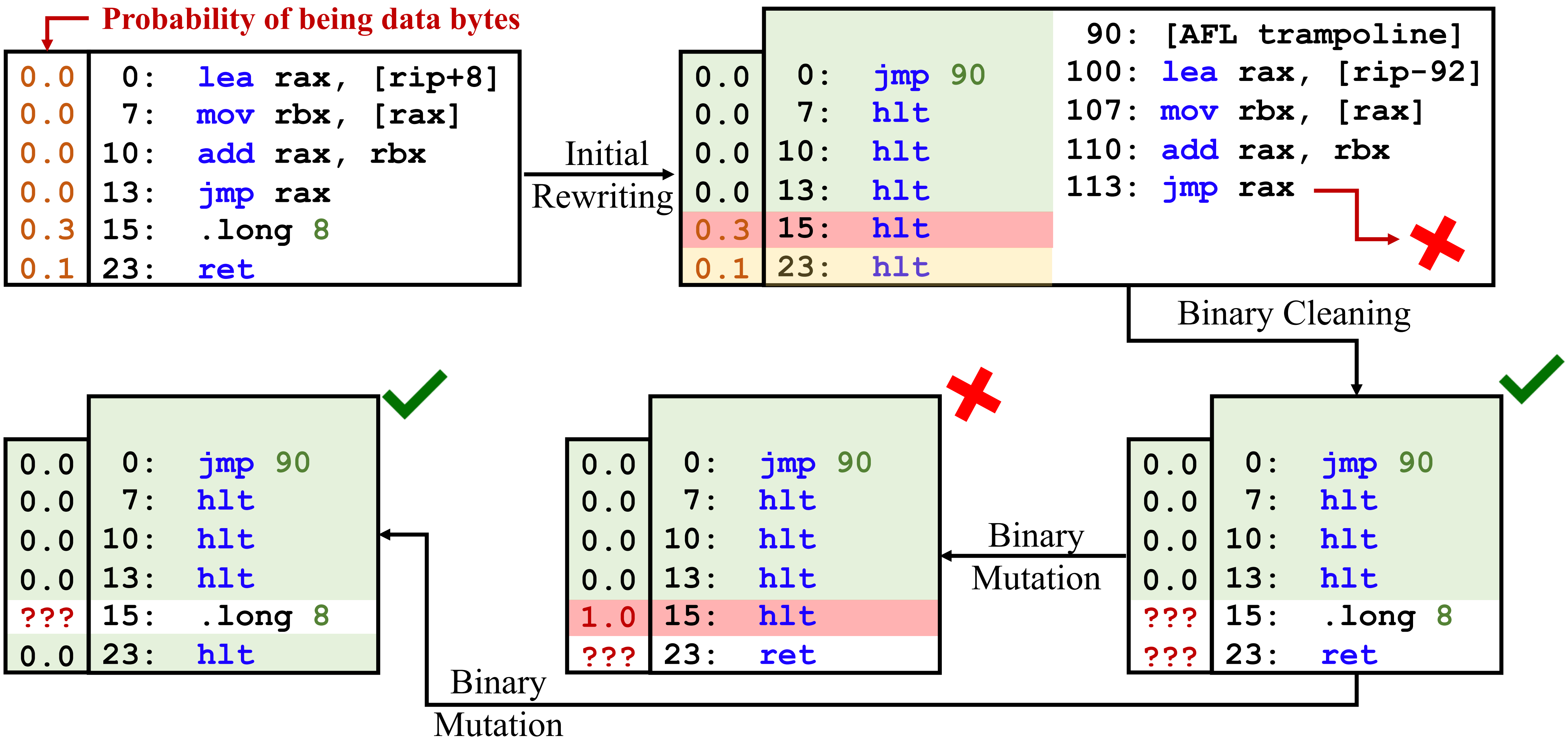
Binary Mutation

```
0.0   0:   jmp 90
0.0   7:   hlt
0.0  10:   hlt
0.0  13:   hlt
1.0  15:   hlt
???  23:   ret
```

Binary Mutation

```
0.0   0:   jmp 90
0.0   7:   hlt
0.0  10:   hlt
0.0  13:   hlt
???  15:   .long 8
0.0  23:   hlt
```

How we handle the motivation case: **Stochastic Rewriting**



Probability of being data bytes

```
0.0   0:  lea rax, [rip+8]
0.0   7:  mov rbx, [rax]
0.0  10:  add rax, rbx
0.0  13:  jmp rax
0.3  15:  .long 8
0.1  23:  ret
```

Initial Rewriting

```
0.0   0:  jmp 90          90:  [AFL trampoline]
0.0   7:  hlt            100:  lea rax, [rip-92]
0.0  10:  hlt            107:  mov rbx, [rax]
0.0  13:  hlt            110:  add rax, rbx
0.3  15:  hlt            113:  jmp rax
0.1  23:  hlt
```

Binary Cleaning

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
???  15:  .long 8
???  23:  ret
```

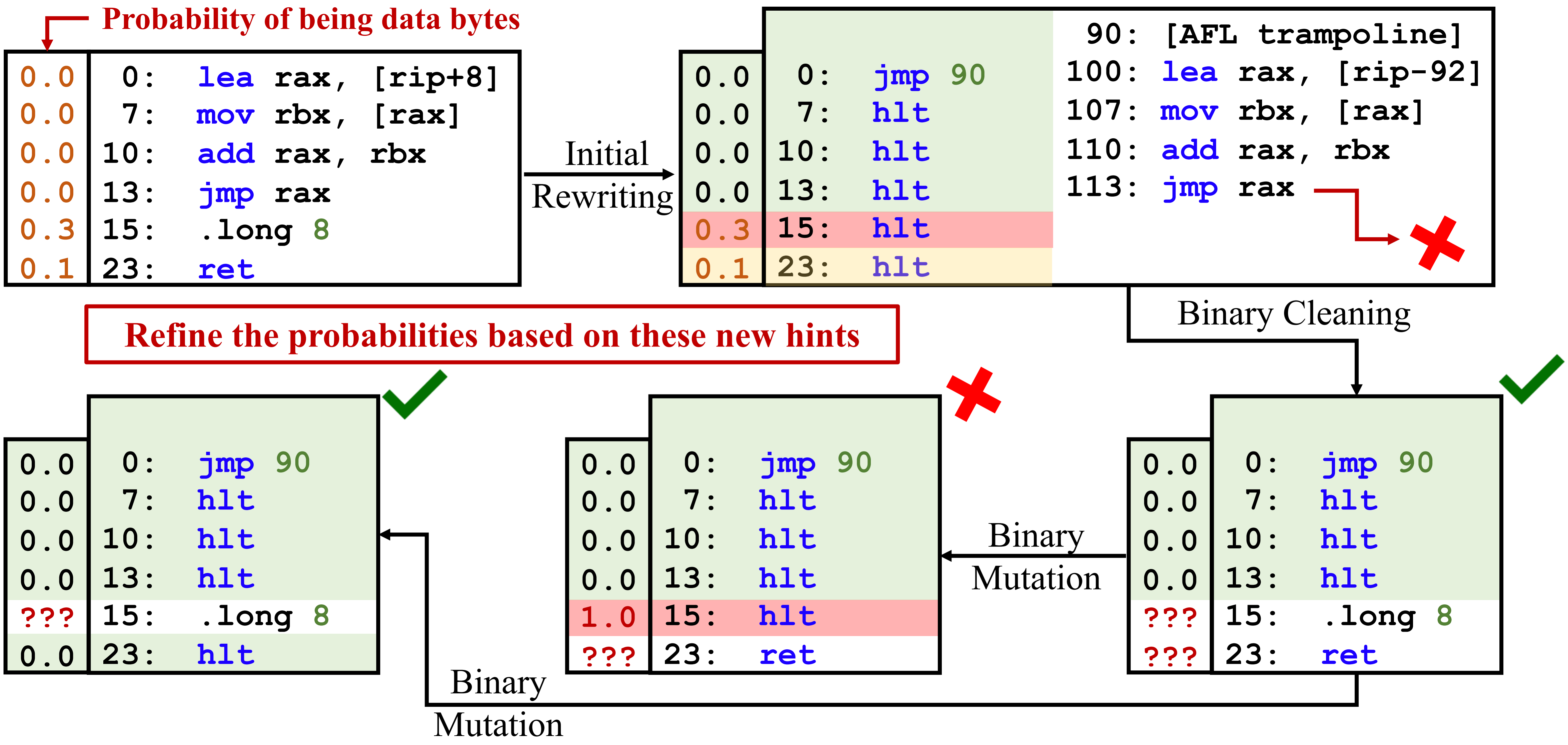
Binary Mutation

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
1.0  15:  hlt
???  23:  ret
```

Binary Mutation

```
0.0   0:  jmp 90
0.0   7:  hlt
0.0  10:  hlt
0.0  13:  hlt
???  15:  .long 8
0.0  23:  hlt
```

How we handle the motivation case: **Stochastic Rewriting**



Probability of being data bytes

```
0.0 |  0:   lea rax, [rip+8]
0.0 |  7:   mov rbx, [rax]
0.0 | 10:   add rax, rbx
0.0 | 13:   jmp rax
0.3 | 15:   .long 8
0.1 | 23:   ret
```

Initial Rewriting

```
0.0 |  0:   jmp 90     |  90:  [AFL trampoline]
0.0 |  7:   hlt        | 100:  lea rax, [rip-92]
0.0 | 10:   hlt        | 107:  mov rbx, [rax]
0.0 | 13:   hlt        | 110:  add rax, rbx
0.3 | 15:   hlt        | 113:  jmp rax
0.1 | 23:   hlt        |
```

Binary Cleaning

```
0.0 |  0:   jmp 90
0.0 |  7:   hlt
0.0 | 10:   hlt
0.0 | 13:   hlt
??? | 15:   .long 8
??? | 23:   ret
```

Binary Mutation

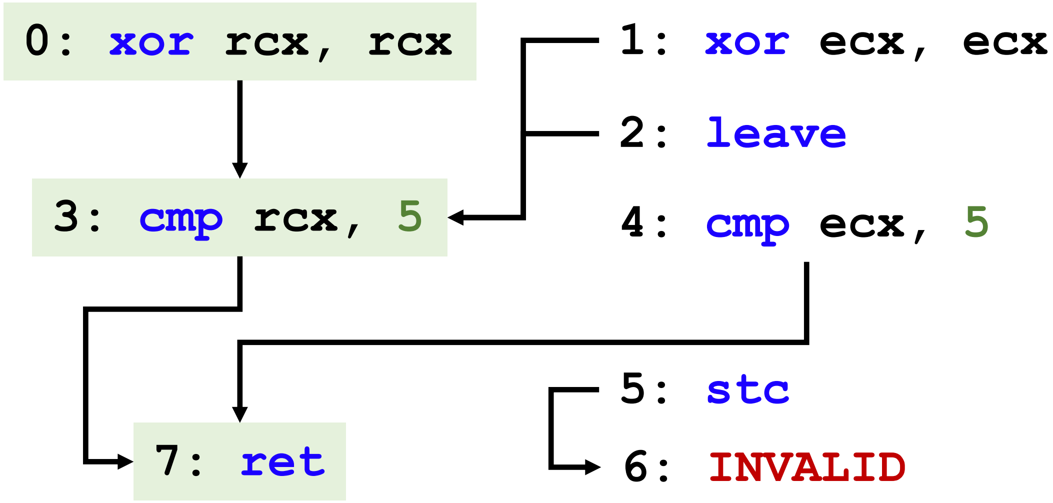
```
0.0 |  0:   jmp 90
0.0 |  7:   hlt
0.0 | 10:   hlt
0.0 | 13:   hlt
1.0 | 15:   hlt
??? | 23:   ret
```

Binary Mutation

```
0.0 |  0:   jmp 90
0.0 |  7:   hlt
0.0 | 10:   hlt
0.0 | 13:   hlt
??? | 15:   .long 8
0.0 | 23:   hlt
```

Refine the probabilities based on these new hints

Universal Control-flow Graph (*UCFG*) and Probability Analysis



`0: xor rcx, rcx`

`1: xor ecx, ecx`

`2: leave`

`3: cmp rcx, 5`

`4: cmp ecx, 5`

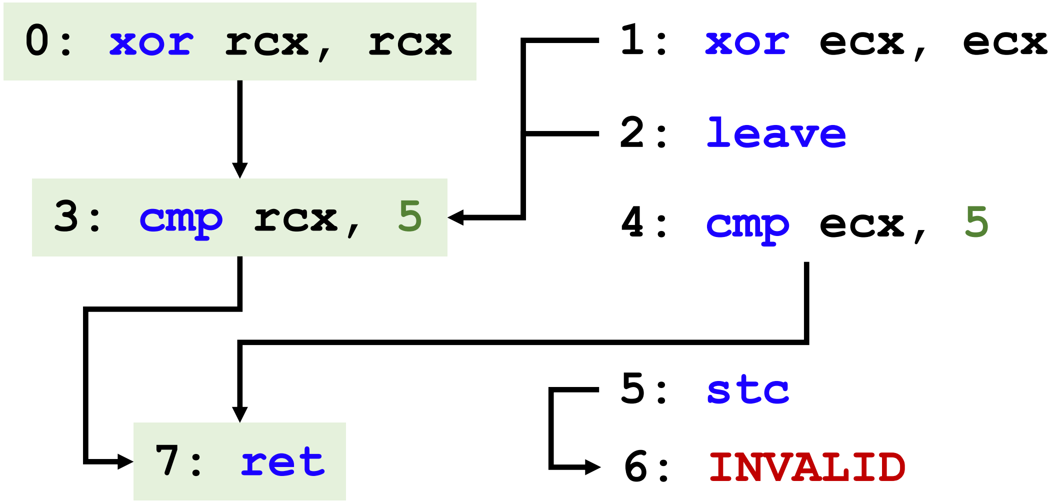
`5: stc`

`6: INVALID`

`7: ret`

Addr	Byte	[Len]	Decoded Instruction
0	48	[3]	`xor rcx, rcx`
1	31	[2]	`xor ecx, ecx`
2	c9	[1]	`leave`
3	48	[4]	`cmp rcx, 5`
4	83	[3]	`cmp ecx, 5`
5	f9	[1]	`stc`
6	05	[0]	`INVALID`
7	c3	[1]	`ret`

Universal Control-flow Graph (*UCFG*) and Probability Analysis



`0: xor rcx, rcx`

`1: xor ecx, ecx`

`2: leave`

`3: cmp rcx, 5`

`4: cmp ecx, 5`

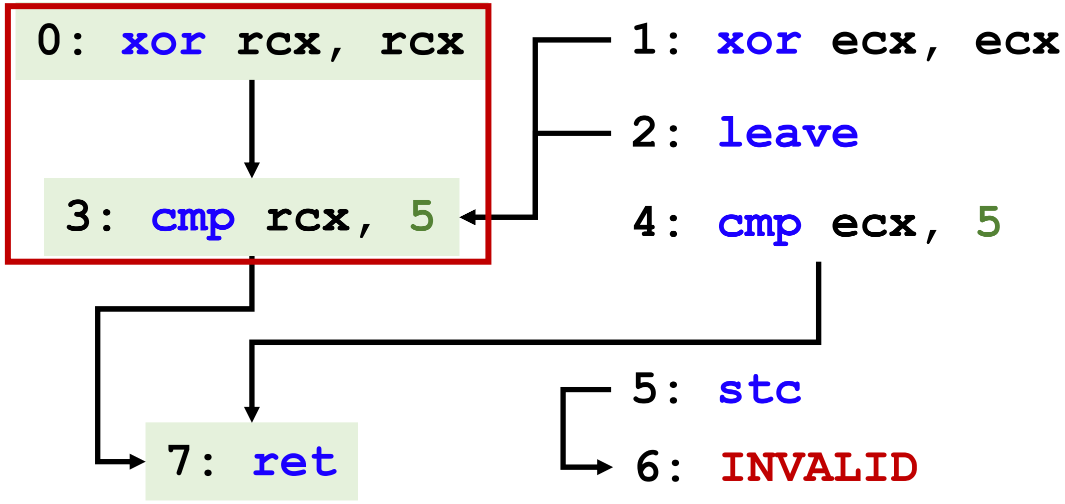
`5: stc`

`6: INVALID`

`7: ret`

Addr	Byte	[Len]	Decoded Instruction
0	48	[3]	xor rcx, rcx
1	31	[2]	xor ecx, ecx
2	c9	[1]	leave
3	48	[4]	cmp rcx, 5
4	83	[3]	cmp ecx, 5
5	f9	[1]	stc
6	05	[0]	INVALID
7	c3	[1]	ret

Universal Control-flow Graph (*UCFG*) and Probability Analysis



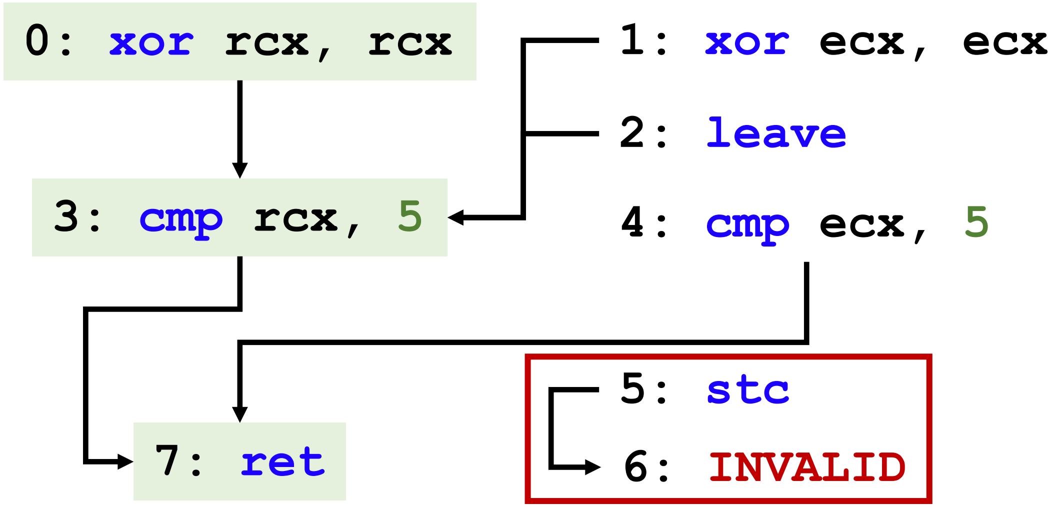
```
0: xor rcx, rcx
3: cmp rcx, 5
7: ret
1: xor ecx, ecx
2: leave
4: cmp ecx, 5
5: stc
6: INVALID
```

Addr	Byte	[Len]	Decoded Instruction
0	48	[3]	`xor rcx, rcx`
1	31	[2]	`xor ecx, ecx`
2	c9	[1]	`leave`
3	48	[4]	`cmp rcx, 5`
4	83	[3]	`cmp ecx, 5`
5	f9	[1]	`stc`
6	05	[0]	INVALID
7	c3	[1]	`ret`

If there is a definition-use relation between two addresses, both addresses are likely to be code

- Address ***0*** and address ***3*** have a definition-use relation about register *rcx*.

Universal Control-flow Graph (*UCFG*) and Probability Analysis



0: xor rcx, rcx

1: xor ecx, ecx

2: leave

3: cmp rcx, 5

4: cmp ecx, 5

5: stc

6: INVALID

7: ret

Addr	Byte	[Len]	Decoded Instruction
0 :	48	[3]	xor rcx, rcx
1 :	31	[2]	xor ecx, ecx
2 :	c9	[1]	leave
3 :	48	[4]	cmp rcx, 5
4 :	83	[3]	cmp ecx, 5
5 :	f9	[1]	stc
6 :	05	[0]	INVALID
7 :	c3	[1]	ret

If there is a definition-use relation between two addresses, both addresses are likely to be code

- Address ***0*** and address ***3*** have a definition-use relation about register *rcx*.

The control flow cannot reach invalid instructions or data

- Address ***5*** cannot be a valid instruction boundary as it leads to an ***invalid*** instruction.

Error Diagnosis: Delta Debugging

Error Diagnosis: Delta Debugging

- Stochastic Rewriting needs to locate and repair the crashs inducing rewriting errors.

- Delta Debugging
  - A binary-search like debugging technique
  - Check whether the unintentional crash can be reproduced with part of uncertain addresses patched

Error Diagnosis: Delta Debugging

- Stochastic Rewriting needs to locate and repair the crashs inducing rewriting errors.

- Delta Debugging
  - A binary-search like debugging technique
  - Check whether the unintentional crash can be reproduced with part of uncertain addresses patched

We also address a number of practical challenges

- Rewriting optimization (e.g., removing flag register saving)
- Supporting stack unwinding (e.g., exception handling in C++)
- Reducing process set up cost
- Safeguarding non-crashing rewriting errors
- Handling overlapping rewriting

Evaluation

Evaluation

Benchmark:

- Google Fuzzer Test Suite (Google FTS)
- Google Fuzzer Test Suite w/ inlined data
- Fuzzing benchmark from RetroWrite

Baselines:

- *E9patch*: static binary rewriting [PLDI'20]
- *Datalog Disassembly*: static binary rewriting [USENIX Security'20]
- *RetroWrite*: static binary rewriting [S&P'20]
- *PTFuzzer*: hardware-assisted fuzzing [IEEE Access'18]
- *AFL-Qemu*: dynamic binary translation
- *AFL-GCC*: compiler-based instrumentation
- *AFL-Clang-fast*: compiler-based instrumentation

Evaluation

Benchmark:

- Google Fuzzer Test Suite (Google FTS)
- Google Fuzzer Test Suite w/ inlined data
- Fuzzing benchmark from RetroWrite

Baselines:

- *E9patch*: static binary rewriting [PLDI'20]
- *Datalog Disassembly*: static binary rewriting [USENIX Security'20]
- *RetroWrite*: static binary rewriting [S&P'20]
- *PTFuzzer*: hardware-assisted fuzzing [IEEE Access'18]
- *AFL-Qemu*: dynamic binary translation
- *AFL-GCC*: compiler-based instrumentation
- *AFL-Clang-fast*: compiler-based instrumentation

Evaluation

Benchmark:

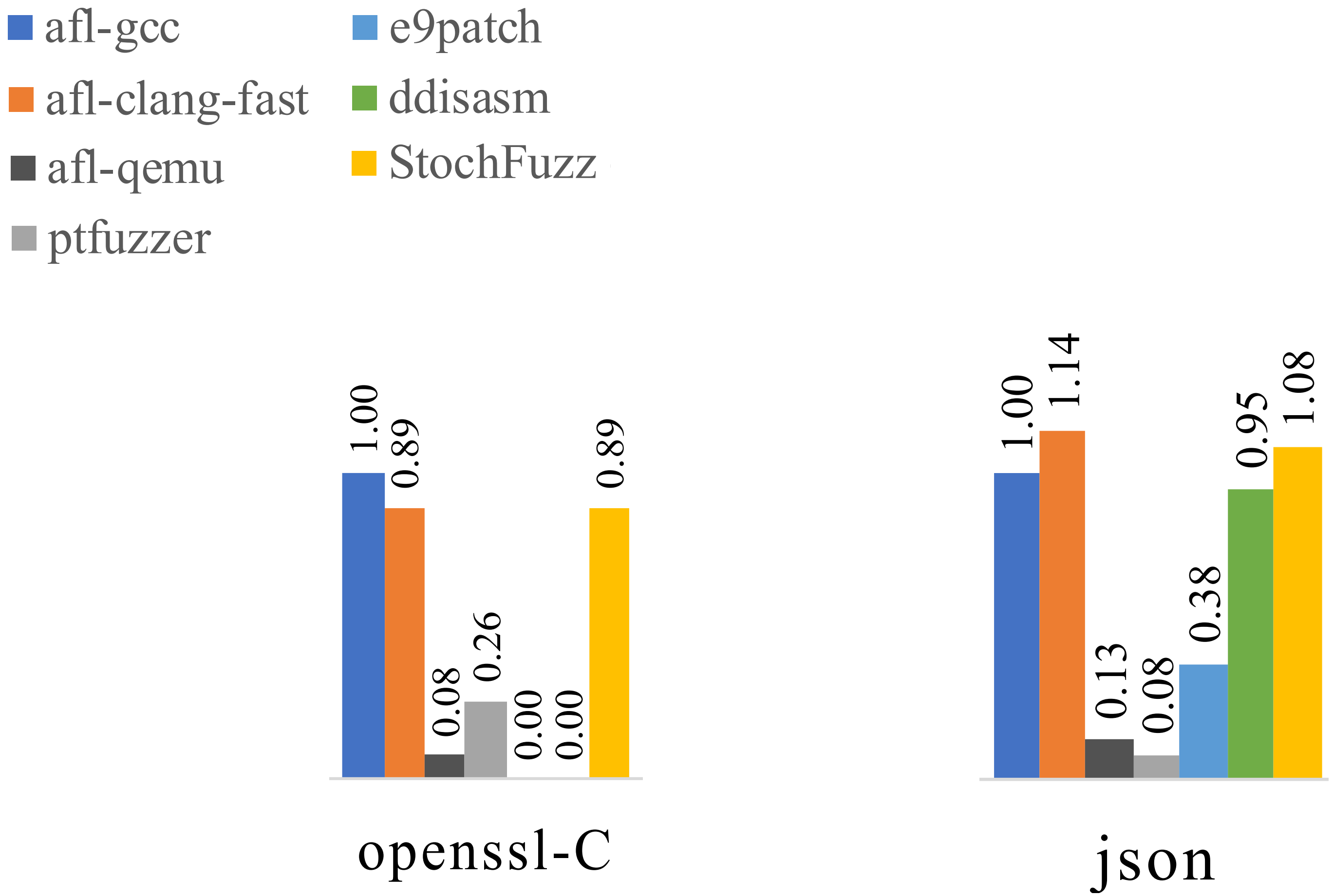
- Google Fuzzer Test Suite (Google FTS)
- Google Fuzzer Test Suite w/ inlined data
- Fuzzing benchmark from RetroWrite

Baselines:

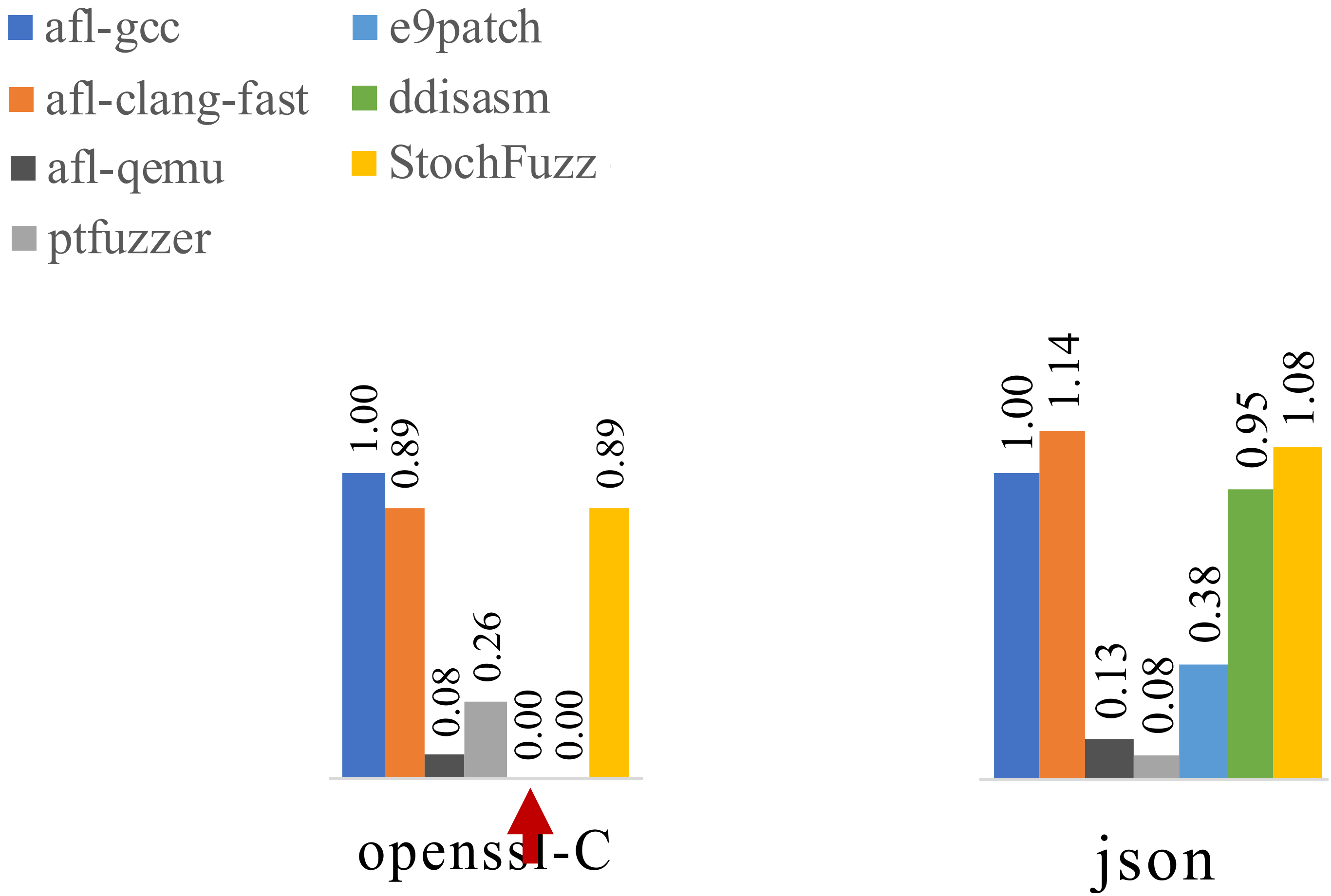
- *E9patch*: static binary rewriting [PLDI’20]
- *Datalog Disassembly*: static binary rewriting [USENIX Security’20]
- *RetroWrite*: static binary rewriting [S&P’20]
- *PTFuzzer*: hardware-assisted fuzzing [IEEE Access’18]
- *AFL-Qemu*: dynamic binary translation
- *AFL-GCC*: compiler-based instrumentation
- *AFL-Clang-fast*: compiler-based instrumentation

Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

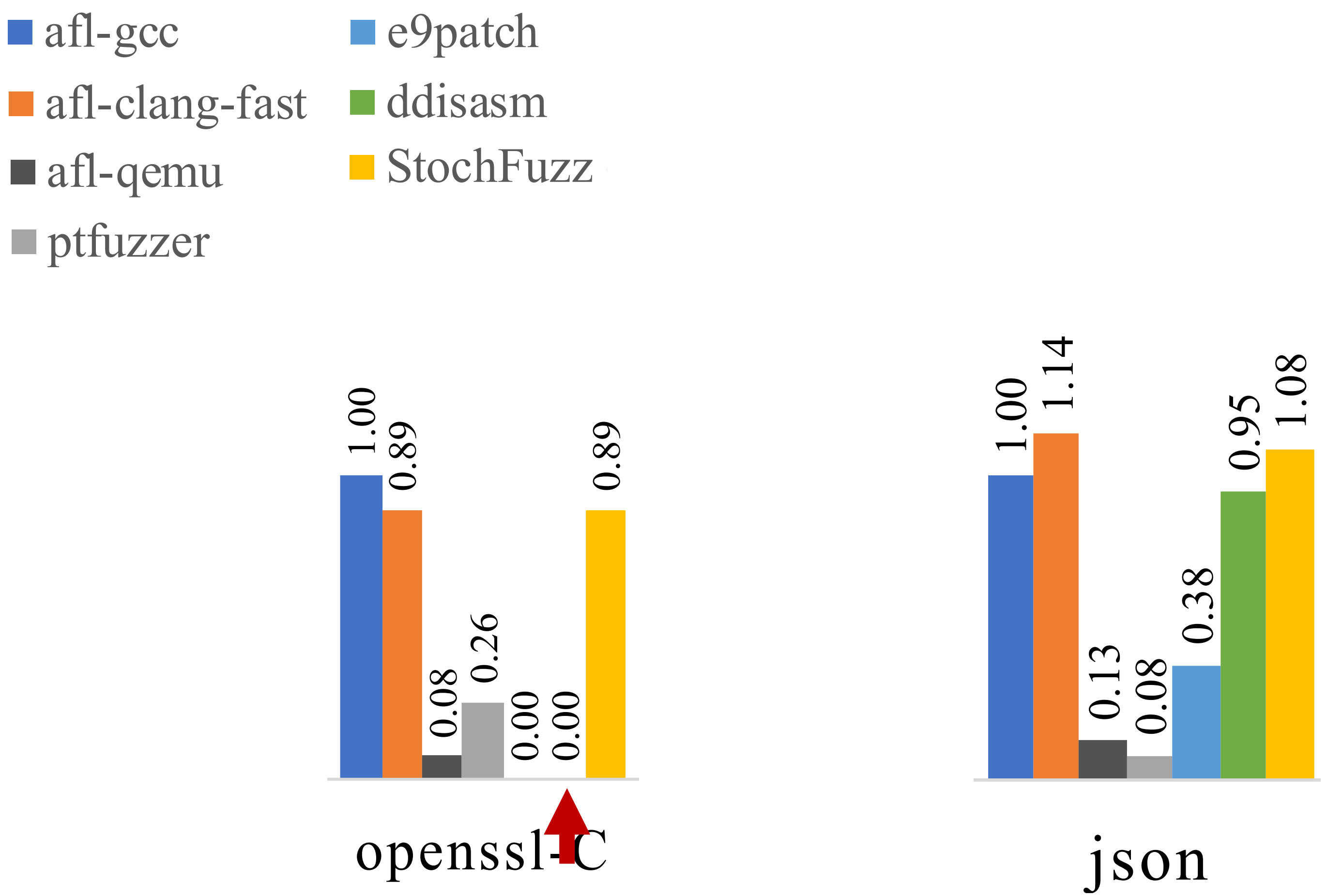
Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)



Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

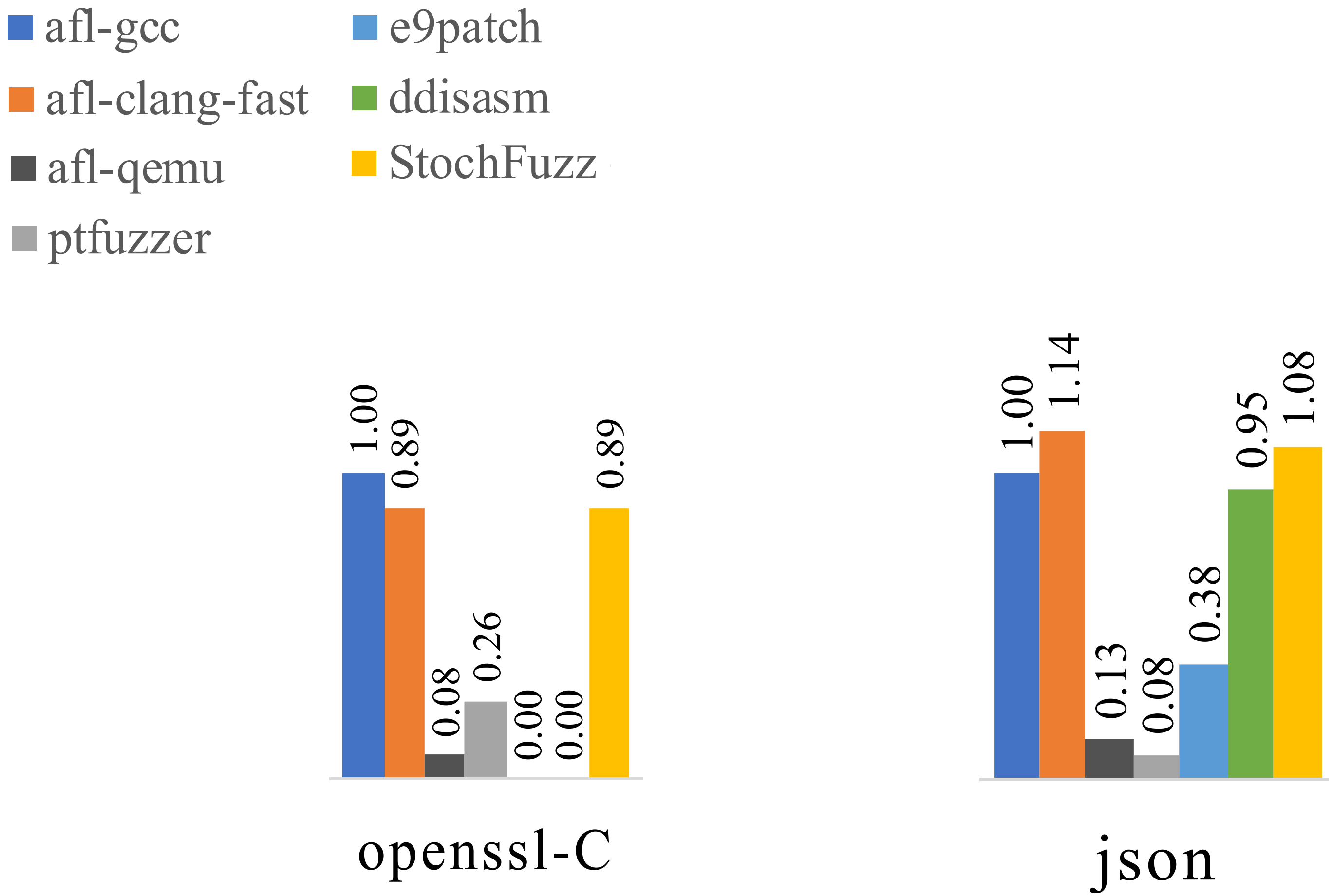


Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)



Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5−37.5%*** of the programs, while StochFuzz succeeds on all the ***24*** programs.
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.
- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***

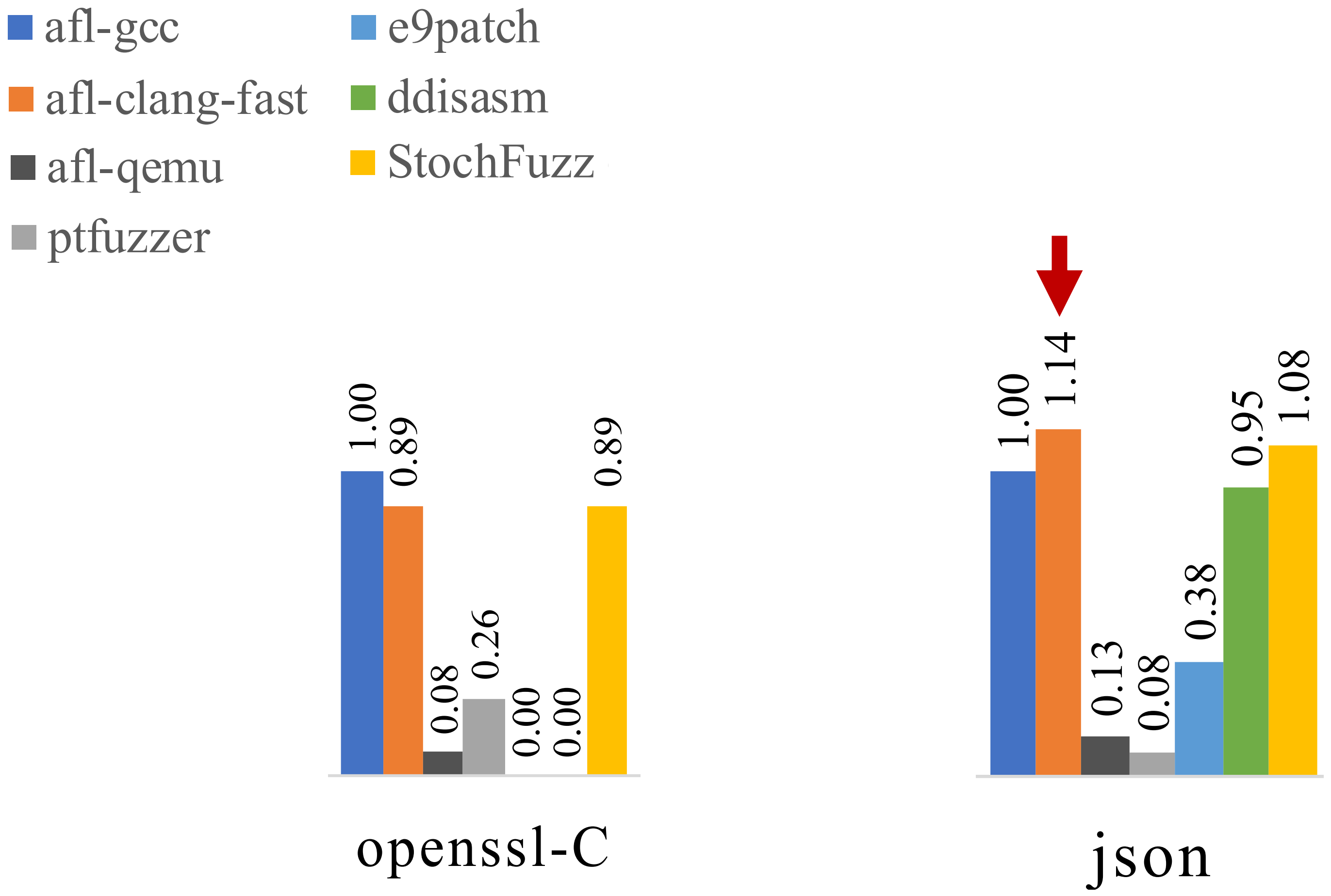


Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5−37.5%*** of the programs, while StochFuzz succeeds on all the ***24*** programs.

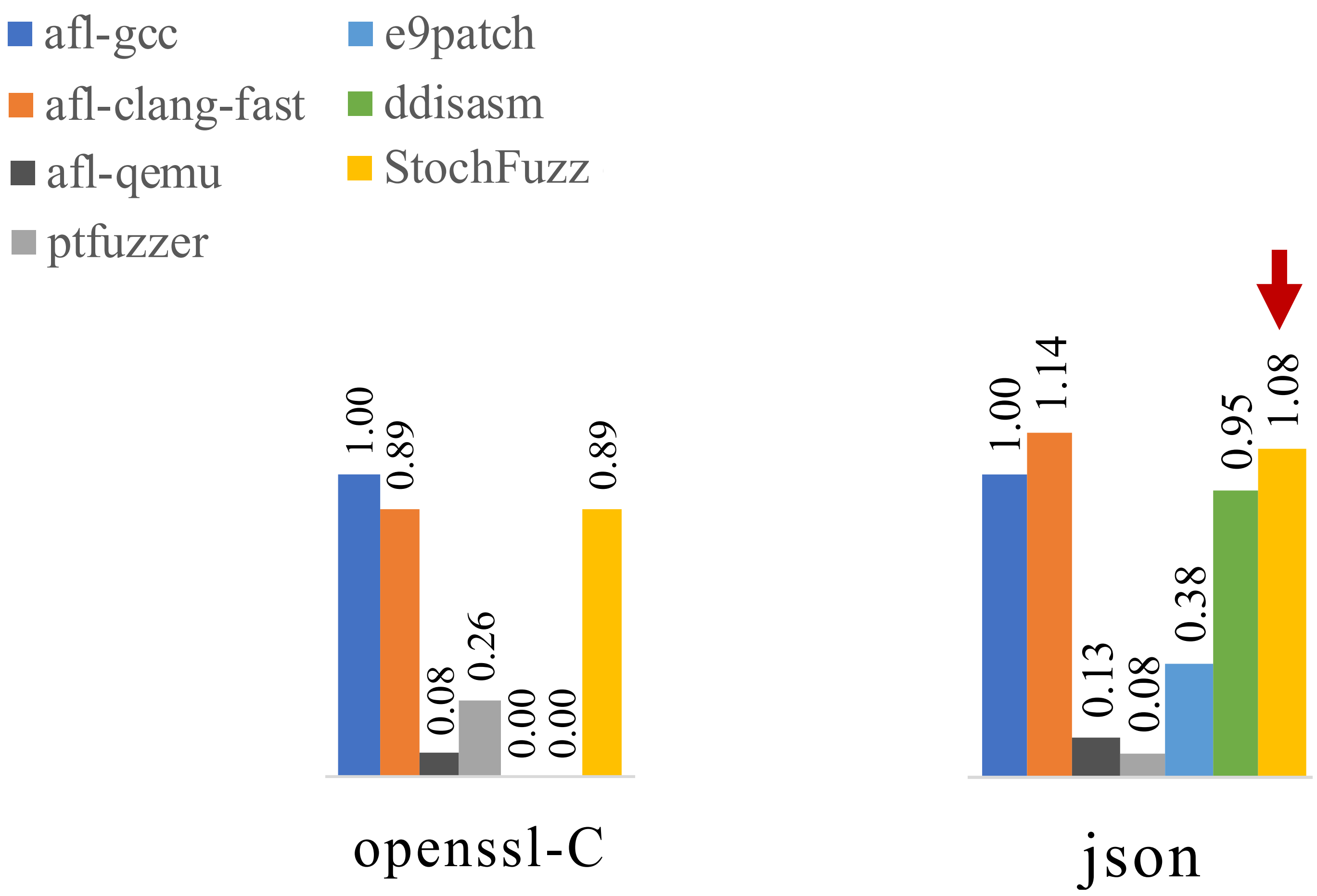
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.

- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***



Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5–37.5%*** of the programs, while StochFuzz succeeds on all the ***24*** programs.
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.
- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***

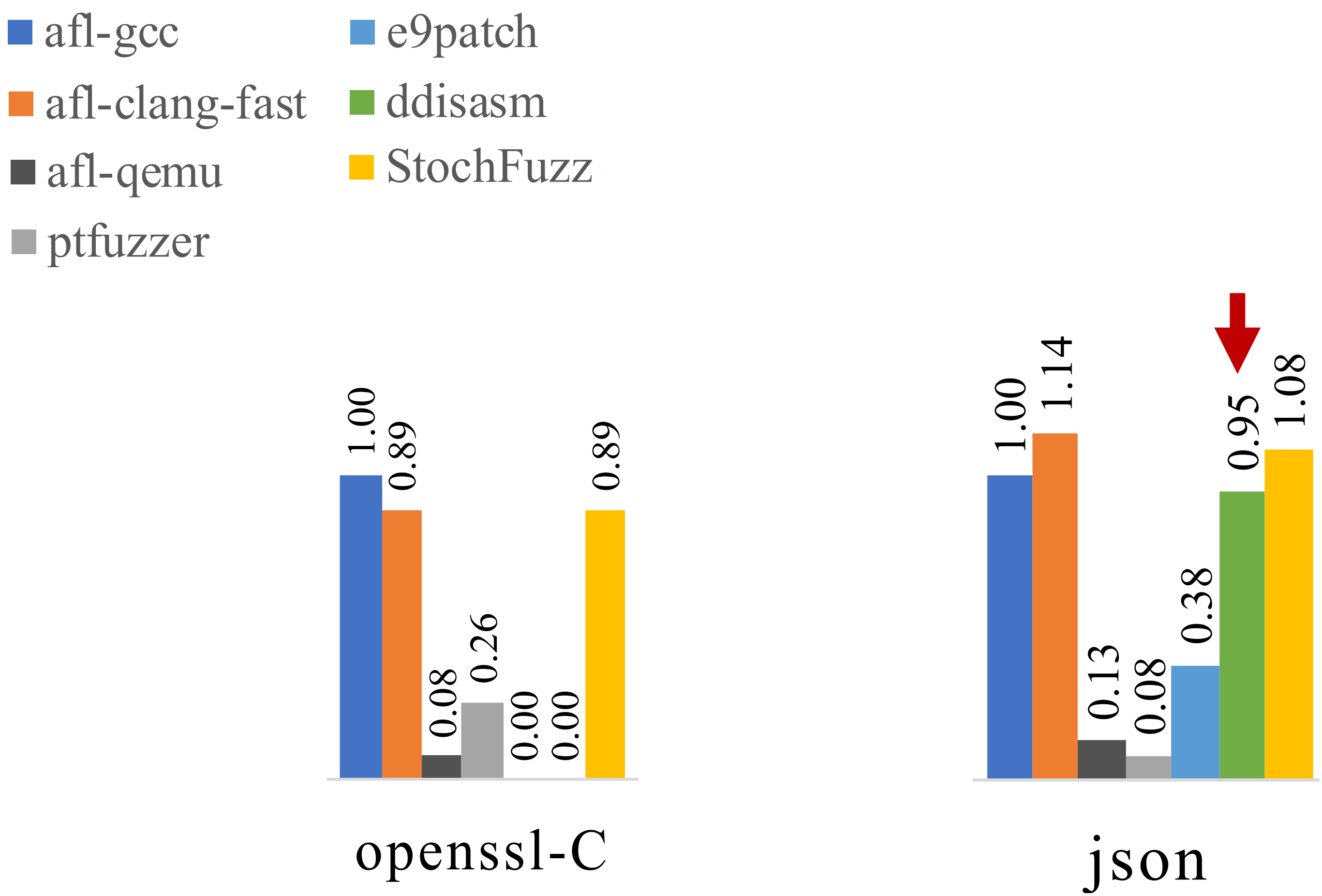


Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5−37.5%*** of the programs, while StochFuzz succeeds on all the ***24*** programs.

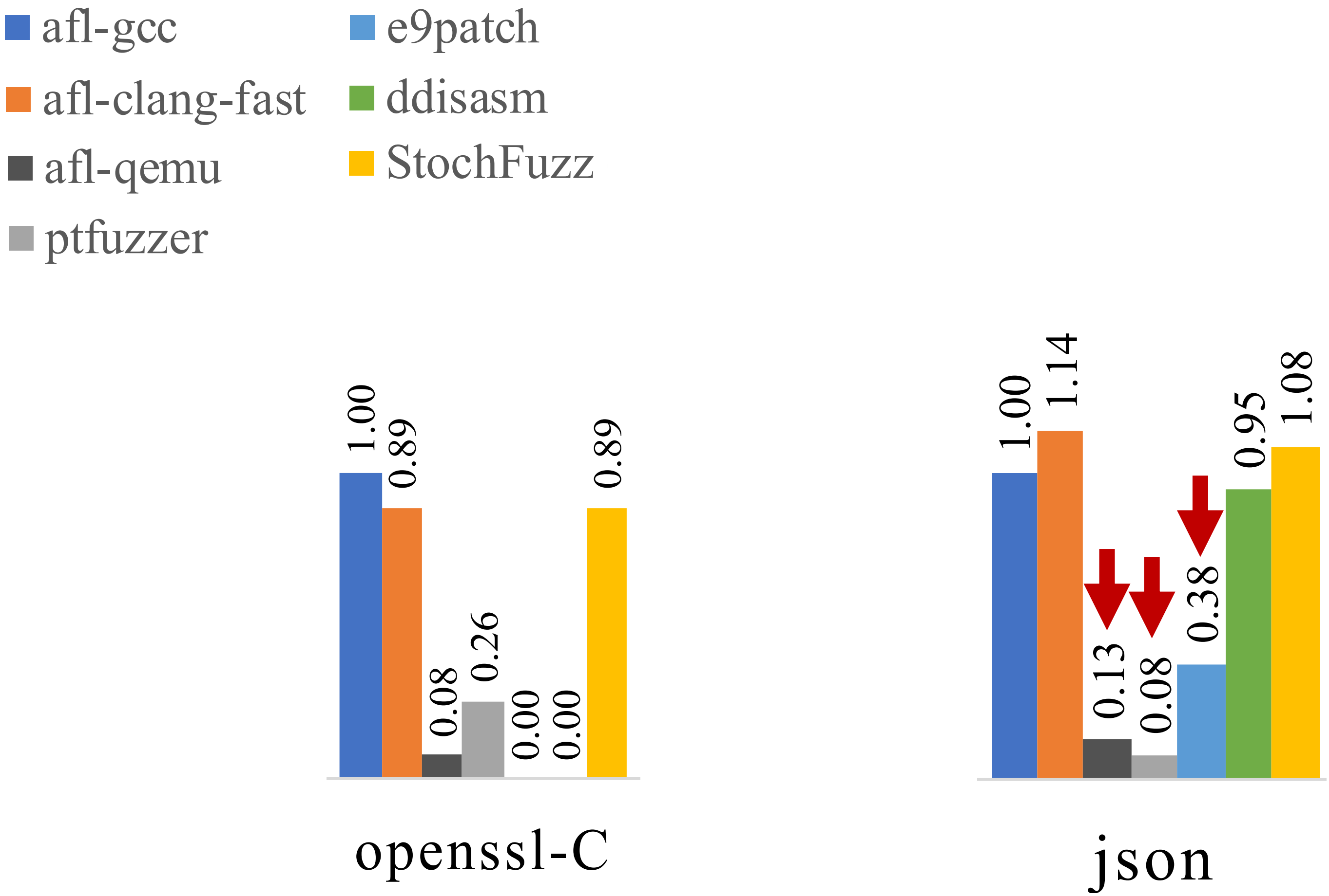
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.

- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***



Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5−37.5%*** of the programs, while StochFuzz succeeds on all the ***24*** programs.
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.
- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***



Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5–37.5%*** of the programs, while StochFuzz succeeds on all the *24* programs.

- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.

- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***

• *AFL-GCC*:	124.1 million
• ***AFL-Clang-fast*:**	**138.1 million**
• *AFL-Qemu*:	16.0 million
• *PTFuzzer*:	24.4 million
• *E9patch*:	23.8 million
• *Datalog Disassembly*:	98.7 million
• ***STOCHFUZZ*:**	**129.3 million**

Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5–37.5%*** of the programs, while StochFuzz succeeds on all the *24* programs.
- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.
- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***

- *AFL-GCC*: 124.1 million
- ***AFL-Clang-fast*: 138.1 million**
- *AFL-Qemu*: 16.0 million
- *PTFuzzer*: 24.4 million
- *E9patch*: 23.8 million
- *Datalog Disassembly*: 98.7 million
- ***STOCHFUZZ*: 129.3 million**

Evaluation: Fuzzing Efficiency on Google FTS (in 24 hours)

- Existing static rewriting techniques (e9patch and datalog disasm) fail on ***12.5–37.5%*** of the programs, while StochFuzz succeeds on all the *24* programs.

- Compared with *afl-clang-fast*, the IR-based instrumentation, StochFuzz only has **11.77%** slowdown on average.

- Other tools have relatively higher overhead.
  - AFL-Qemu: ***88.71%***
  - PTFuzzer: ***75.81%***

AFL-GCC:	124.1 million
***AFL-Clang-fast*:**	**138.1 million**
AFL-Qemu:	16.0 million
PTFuzzer:	24.4 million
E9patch:	23.8 million
Datalog Disassembly:	98.7 million
***STOCHFUZZ*:**	**129.3 million**

Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

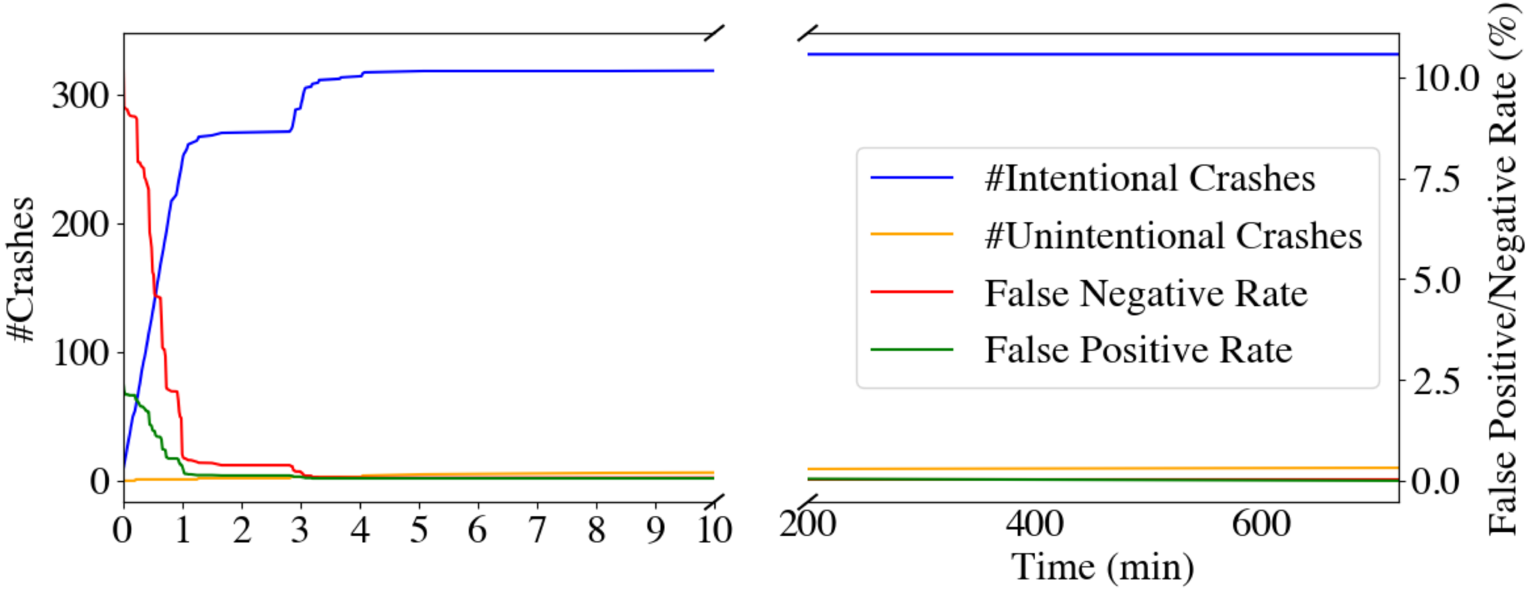
We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

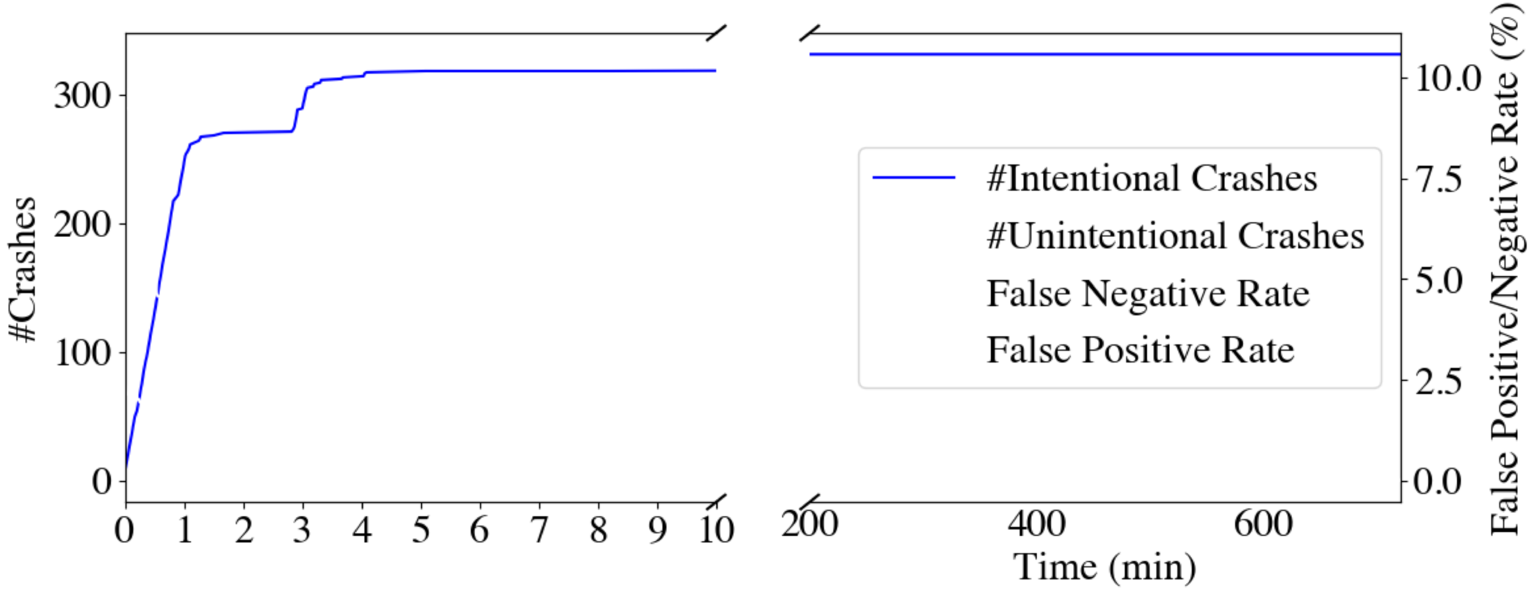
We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.



Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

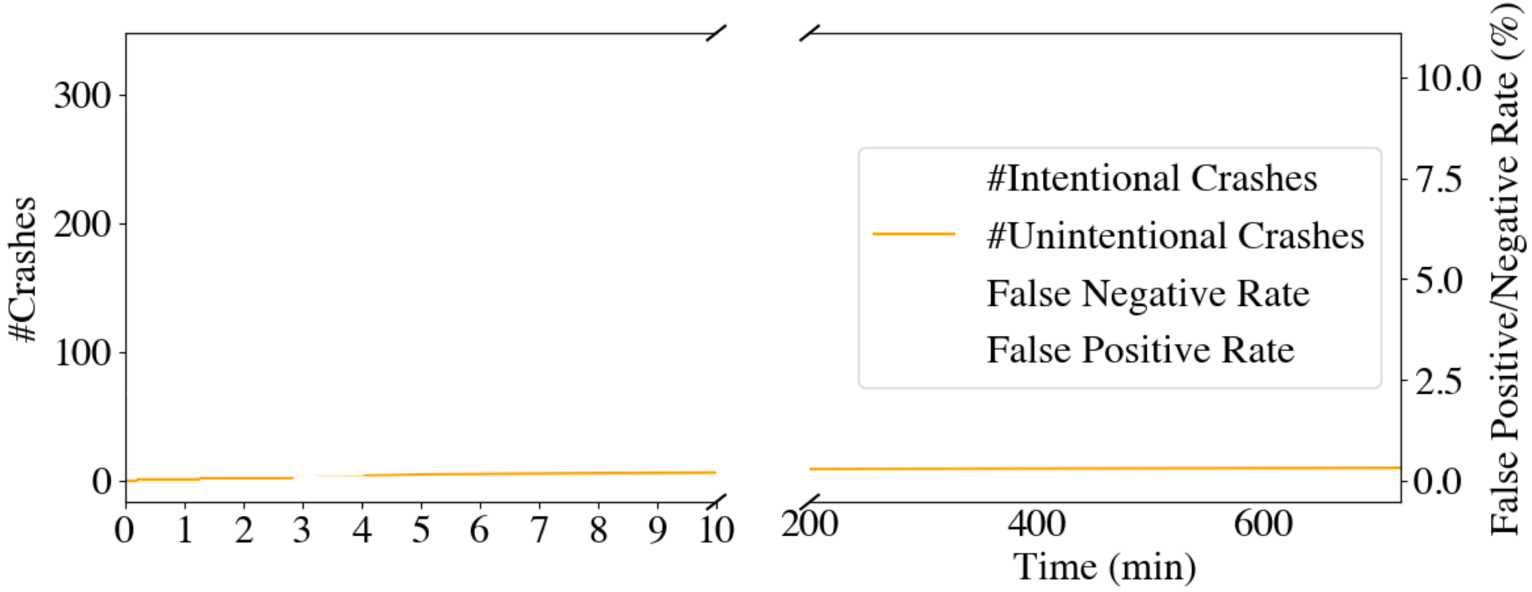
We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.



Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

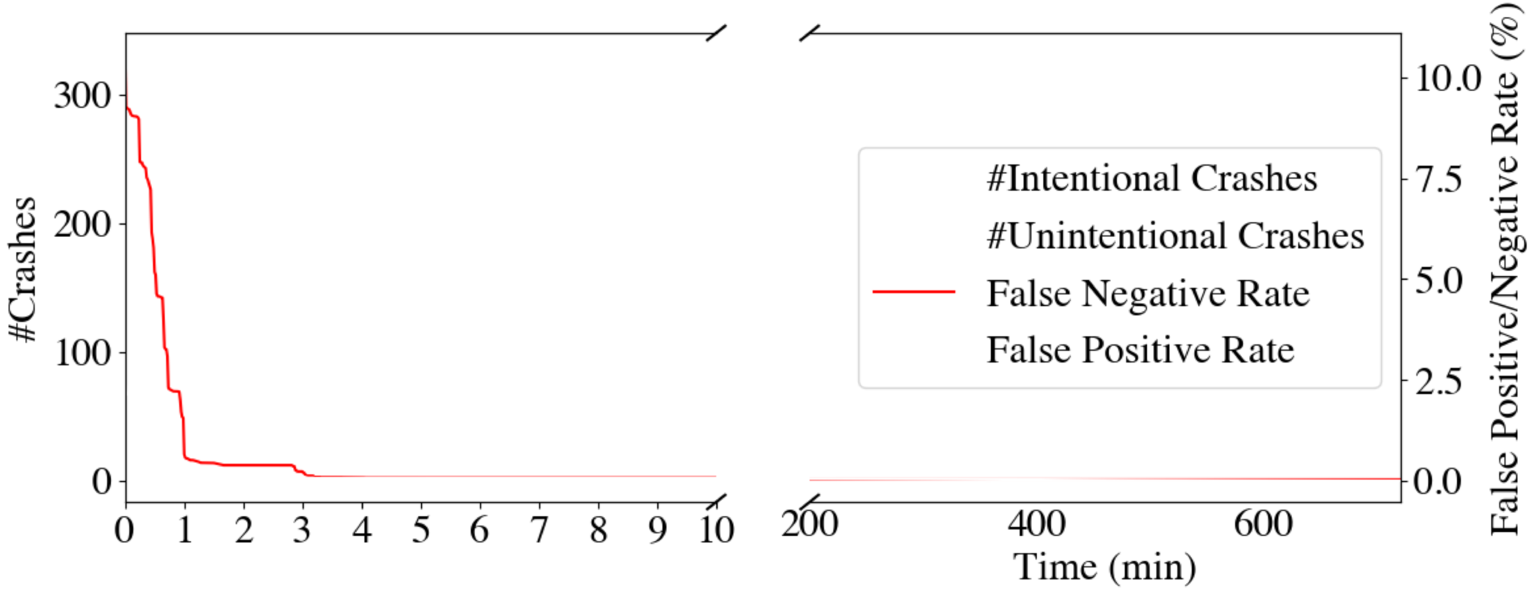
We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.



Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

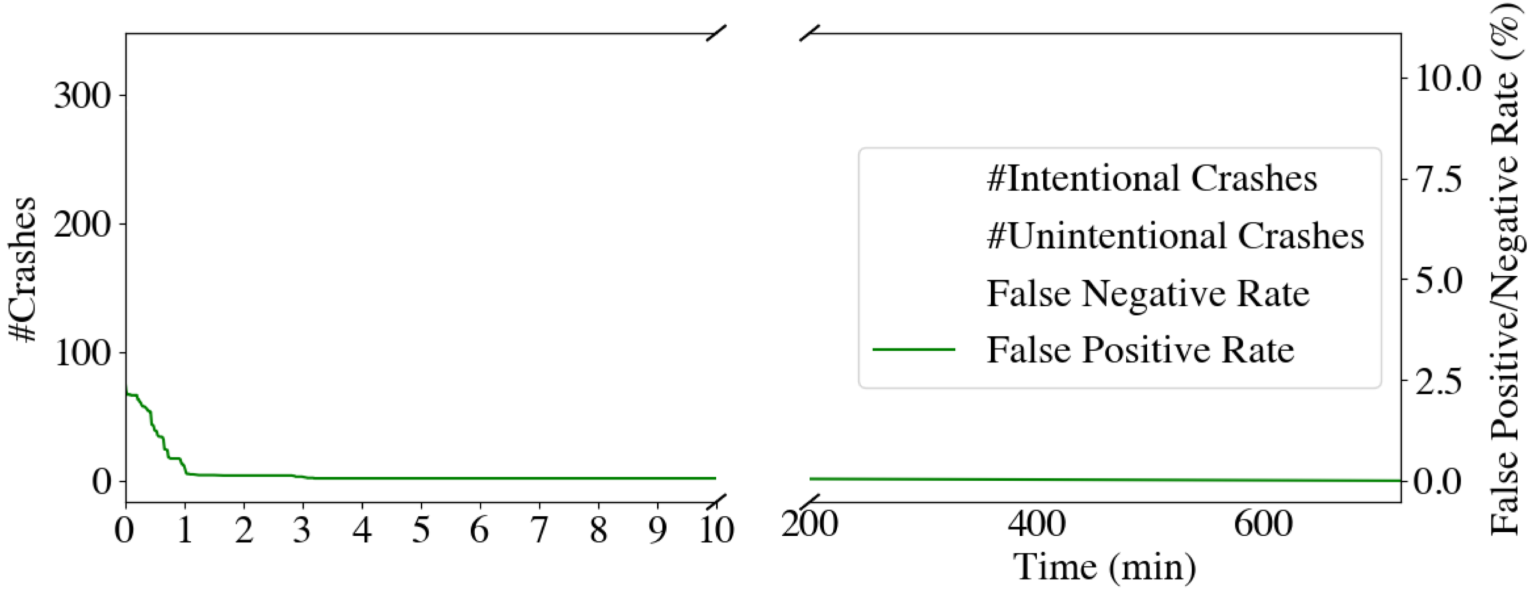
We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.



Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

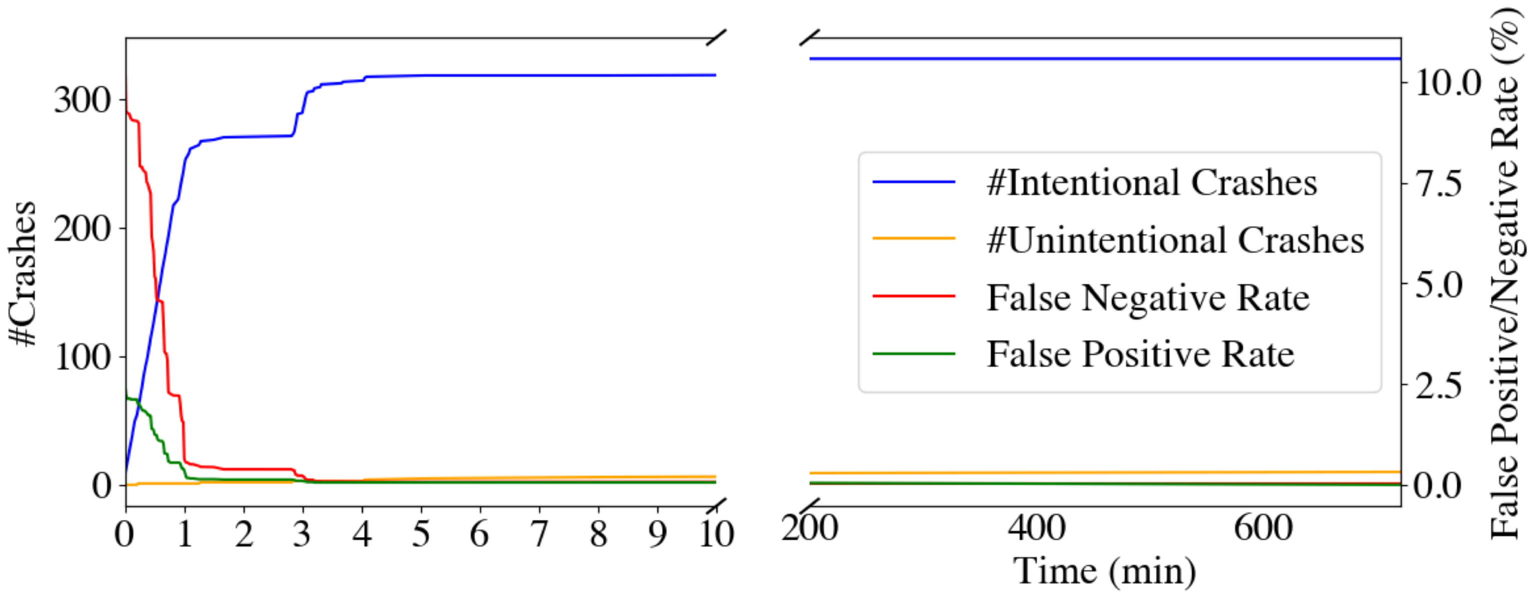
We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.



Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

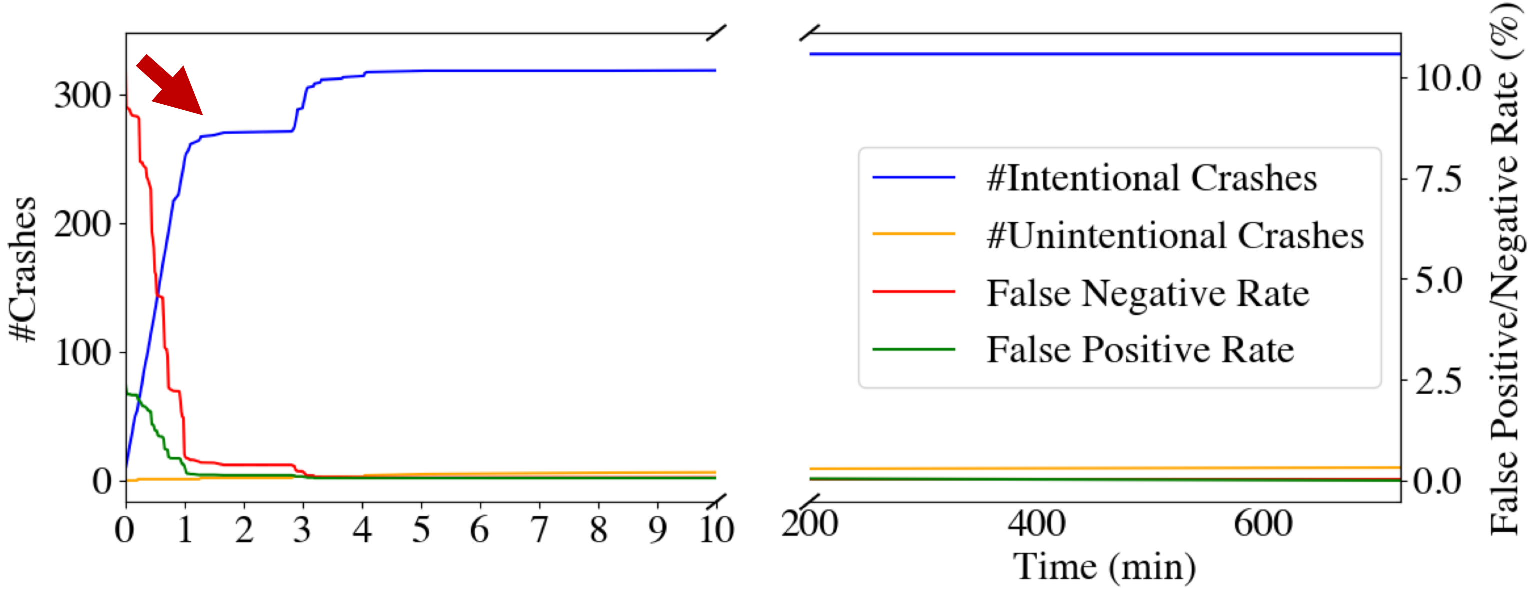


Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

- The process almost converges at the first 5 minutes.
- The number of crashes by rewriting errors is very small compared to that of intentional crashes (i.e., most rewriting errors are fixed by observing new coverage, without triggering unintentional crashes).

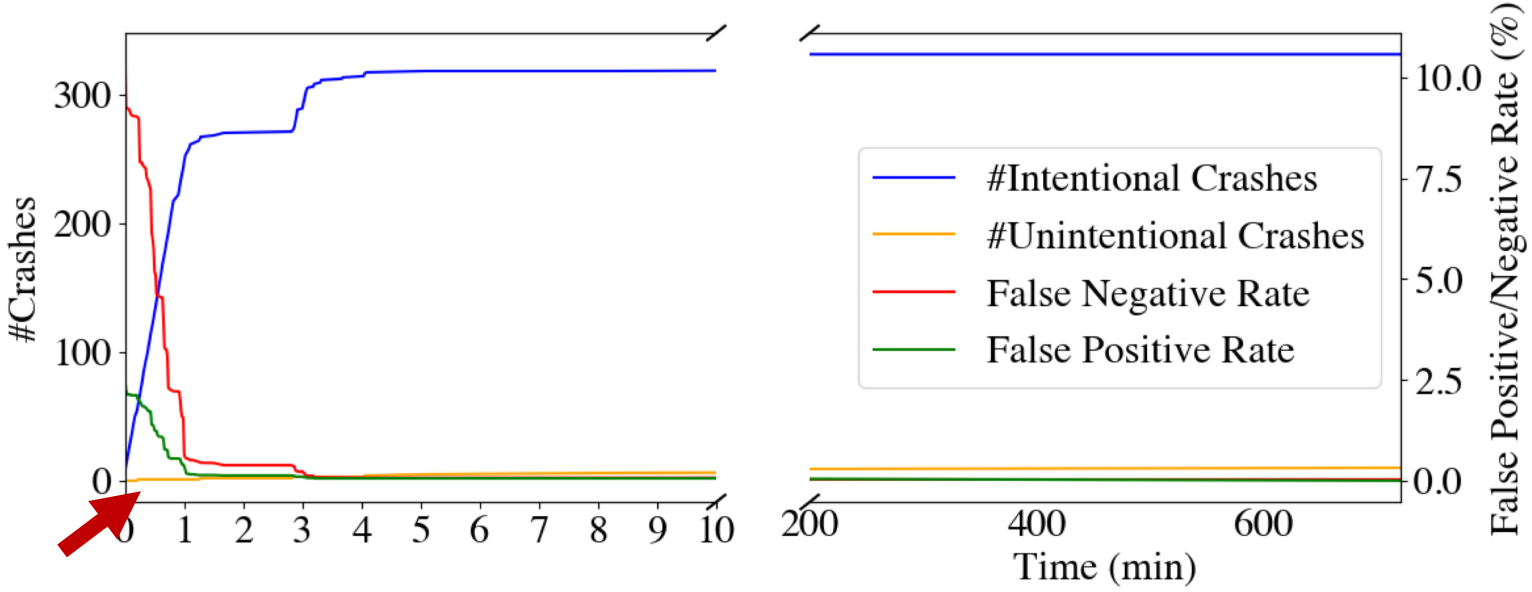


Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

- The process almost converges at the first 5 minutes.
- The number of crashes by rewriting errors is very small compared to that of intentional crashes (i.e., most rewriting errors are fixed by observing new coverage, without triggering unintentional crashes).

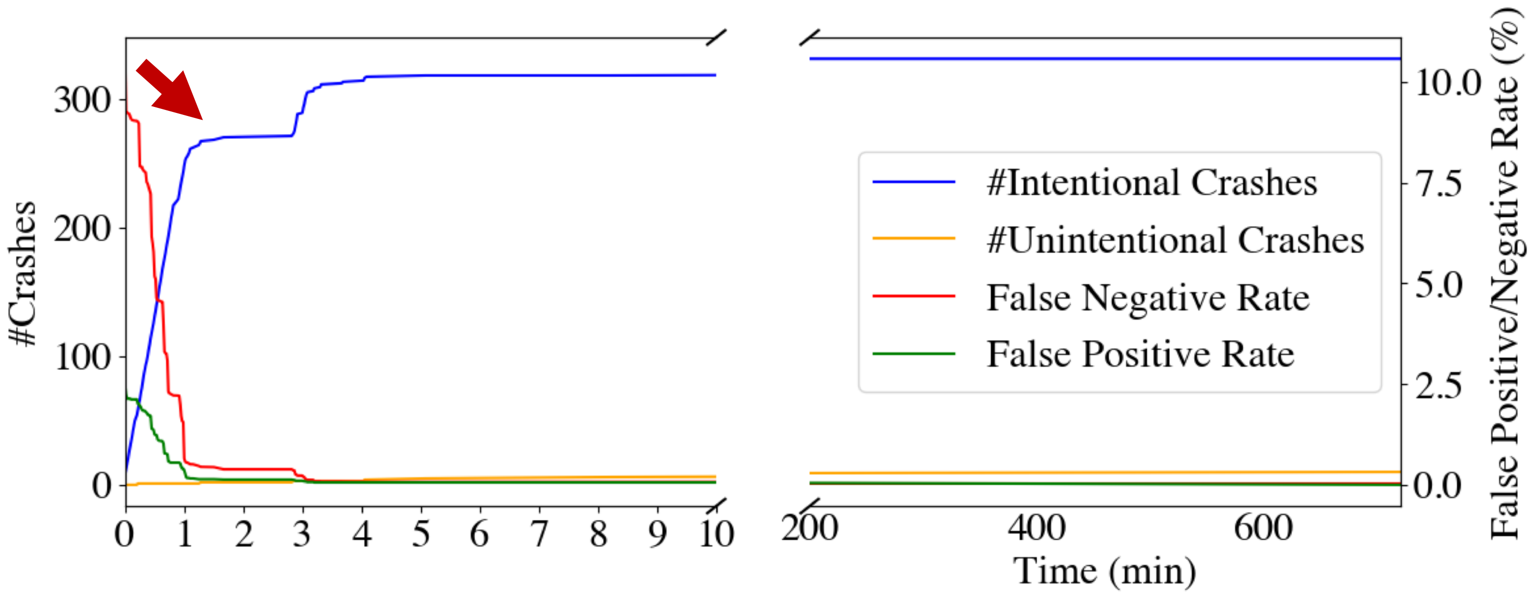


Evaluation: Progress of Incremental and Stochastic Rewriting (*freetype2*)

We hence modify the compilation tool-chain of Google FTS to force *.rodata* sections to be interleaved with *.text* sections.

We study how the numbers of intentional crashes and unintentional crashes, false positives (FPs) (i.e., a data byte is identified as code) and false negatives (FNs) (i.e., a code byte is not identified as code) change over 24-hour fuzzing.

- The process almost converges at the first 5 minutes.
- The number of crashes by rewriting errors is very small compared to that of intentional crashes (i.e., most rewriting errors are fixed by observing new coverage, without triggering unintentional crashes).



Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

AFL Instrumentation

IJON Instrumentation

Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == 'A') {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }

}
```

AFL Instrumentation

IJON Instrumentation

Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == ‘A’) {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }

}
```

AFL Instrumentation

IJON Instrumentation

Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == ‘A’) {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }

}
```

AFL Instrumentation

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == ‘A’) {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }
    ijon_value(x, y);
}
```

IJON Instrumentation

Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == 'A') {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }

}
```

AFL Instrumentation

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == 'A') {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }
    ijon_value(x, y);
}
```

IJON Instrumentation

Evaluation: Collect Other Runtime Feedback Than Coverage (*IJON*)

- *IJON*: state-aware fuzzing [S&P’20]
- We port IJON to support binary-only fuzzing based on AFL-Qemu and STOCHFUZZ
- The same maze experiment
- STOCHFUZZ is **8×** faster than afl-qemu, and only has around **8%** slowdown compared with source-code based IJON

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == ‘A’) {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }

}
```

AFL Instrumentation

```
while (...) {
    afl_coverage();
    char c = input();
    if (c == ‘A’) {
        afl_coverage();
        x = change(x, c);
    } else {
        afl_coverage();
        y = change(y, c);
    }
    ijon_value(x, y);
}
```

IJON Instrumentation

Related Works

Binary Rewriting and Binary-only Fuzzing:

- Flores-Montoya, Antonio, and Eric Schulte. "Datalog disassembly." *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.
- Duck, Gregory J., Xiang Gao, and Abhik Roychoudhury. "Binary rewriting without control flow recovery." *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020.
- Dinesh, Sushant, et al. "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization." *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.
- Zhang, Gen, et al. "Ptfuzz: Guided fuzzing with processor trace feedback." *IEEE Access* 6 (2018): 37302-37313.
- Chen, Yaohui, et al. "Ptrix: Efficient hardware-assisted fuzzing for cots binary." *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 2019.
- S.Schumilo,C.Aschermann,R.Gawlik,S.Schinzel,andT.Holz,“kafl: Hardware-assisted feedback fuzzing for {OS} kernels,” in USENIX Security, 2017, pp. 167–182.

Probabilistic Program Analysis:

- Borges, Mateus, et al. "Iterative distribution-aware sampling for probabilistic symbolic execution." *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015.
- Miller, Kenneth, et al. "Probabilistic disassembly." *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

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.

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.

Thanks!

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.

Thanks!

Github Repo

zhan3299@purdue.edu