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
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.

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

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

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

.DATA:
15:  .long 8

.CODE2:
23:  ret
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenges of static binary rewriting

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

.CODE2:
23:  ret

.DATA:
15:  .long  8
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
</tbody>
</table>
Challenges of static binary rewriting

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

.DATA:
15: .long 8

.CODE2:
23: ret
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td>.CODE2 - .DATA</td>
</tr>
</tbody>
</table>

Note: .DATA - .CODE2 - .DATA
Challenges of static binary rewriting

```assembly
.CODE1:

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

 DataService:
15: .long 8

.CODE2:
23: ret

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
</tr>
</tbody>
</table>
Challenges of static binary rewriting

**.CODE1:**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
</tr>
<tr>
<td>mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
</tr>
<tr>
<td>add rax, rbx</td>
<td>rax</td>
<td>23</td>
</tr>
<tr>
<td>jmp rax</td>
<td>jmp</td>
<td>.CODE2</td>
</tr>
</tbody>
</table>

**.DATA:**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>.long 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**.CODE2:**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>.CODE2 - .DATA</td>
</tr>
<tr>
<td></td>
<td>.CODE2</td>
</tr>
</tbody>
</table>
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
```
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
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>add rax, rbx</td>
<td>rax</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>jmp rax</td>
<td>jmp</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>ret</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td>.CODE2-.DATA</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
<td>.CODE2</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>jmp</td>
<td>.CODE2</td>
<td>-</td>
</tr>
<tr>
<td>23: ret</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **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

---

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td>.CODE2 - .DATA</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
<td>.CODE2</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>jmp</td>
<td></td>
<td>.CODE2</td>
</tr>
<tr>
<td>23: ret</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

- **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:**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td>.CODE2 - .DATA</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
<td>.CODE2</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>jmp</td>
<td>.CODE2</td>
<td>-</td>
</tr>
<tr>
<td>15: or [rax], al</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>17: add [rax], al</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>19: add [rax], al</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>21: add [rax], al</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
<td>.CODE2</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>jmp</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>23: ret</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

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

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
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Var</th>
<th>Val</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea  rax, [rip+8]</td>
<td>rax</td>
<td>15</td>
<td>.DATA</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>rbx</td>
<td>8</td>
<td>.CODE2 - .DATA</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>rax</td>
<td>23</td>
<td>.CODE2</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>jmp</td>
<td>.CODE2</td>
<td>-</td>
</tr>
<tr>
<td>23: ret</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **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).

RetroWrite, e9patch, and datalog disassembly (the version before we reported the issue) all fail on a similar case.
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 \texttt{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
7:     hlt
10:    hlt
13:    hlt
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**

<table>
<thead>
<tr>
<th>Initial Rewriting</th>
<th>New Rewriting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: lea rax, [rip+8]</td>
<td>90: [AFL trampoline]</td>
</tr>
<tr>
<td>7: mov rbx, [rax]</td>
<td>0: jmp 90</td>
</tr>
<tr>
<td>10: add rax, rbx</td>
<td>7: hlt</td>
</tr>
<tr>
<td>13: jmp rax</td>
<td>10: hlt</td>
</tr>
<tr>
<td>15: .long 8</td>
<td>13: hlt</td>
</tr>
<tr>
<td>23: ret</td>
<td>15: .long 8</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Initial Rewriting</th>
<th>90: [AFL trampoline]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: <code>lea rax, [rip+8]</code></td>
<td>0: <code>jmp 90</code></td>
</tr>
<tr>
<td>7: <code>mov rbx, [rax]</code></td>
<td>100: <code>lea rax, [rip-92]</code></td>
</tr>
<tr>
<td>10: <code>add rax, rbx</code></td>
<td>7: <code>hlt</code></td>
</tr>
<tr>
<td>13: <code>jmp rax</code></td>
<td>107: <code>mov rbx, [rax]</code></td>
</tr>
<tr>
<td>15: <code>.long 8</code></td>
<td>10: <code>hlt</code></td>
</tr>
<tr>
<td>23: <code>ret</code></td>
<td>110: <code>add rax, rbx</code></td>
</tr>
</tbody>
</table>

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**

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**

<table>
<thead>
<tr>
<th>Initial Rewriting</th>
<th>Incremental Rewriting caused by the intentional crash at address 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: <em>lea</em> rax, [rip+8]</td>
<td>90: [AFL trampoline]</td>
</tr>
<tr>
<td>7: <em>mov</em> rbx, [rax]</td>
<td>0: <em>jmp</em> 90</td>
</tr>
<tr>
<td>10: <em>add</em> rax, rbx</td>
<td>10: <em>hlt</em></td>
</tr>
<tr>
<td>13: <em>jmp</em> rax</td>
<td>107: <em>mov</em> rbx, [rax]</td>
</tr>
<tr>
<td>15: <code>.long</code> 8</td>
<td>10: <em>hlt</em></td>
</tr>
<tr>
<td>23: <em>ret</em></td>
<td>110: <em>add</em> rax, rbx</td>
</tr>
</tbody>
</table>

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**

![Assembly code snippet]

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

```assembly
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**

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>0.0</th>
<th>0: lea rax, [rip+8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>7:  mov rbx, [rax]</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>10: add rax, rbx</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>13: jmp rax</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>15: .long 8</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>23: ret</td>
<td></td>
</tr>
</tbody>
</table>
How we handle the motivation case: **Stochastic Rewriting**

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Initial Rewriting</th>
<th>Stochastic Rewriting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>lea rax, [rip+8]</td>
<td>jmp 90</td>
</tr>
<tr>
<td>0.0</td>
<td>mov rbx, [rax]</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>add rax, rbx</td>
<td>hlt</td>
</tr>
<tr>
<td>0.3</td>
<td>jmp rax</td>
<td>hlt</td>
</tr>
<tr>
<td>0.1</td>
<td>.long 8</td>
<td>hlt</td>
</tr>
<tr>
<td>90: [AFL trampoline]</td>
<td>lea rax, [rip-92]</td>
<td></td>
</tr>
<tr>
<td>100:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>113:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How we handle the motivation case: **Stochastic Rewriting**

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Initial Rewriting</th>
<th>Stochastic Rewriting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0: <strong>lea</strong> rax, [rip+8]</td>
<td>0.0 0: <strong>jmp</strong> 90</td>
</tr>
<tr>
<td>0.0</td>
<td>7: <strong>mov</strong> rbx, [rax]</td>
<td>0.0 7: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.0</td>
<td>10: <strong>add</strong> rax, rbx</td>
<td>0.0 10: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.0</td>
<td>13: <strong>jmp</strong> rax</td>
<td>0.0 13: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.3</td>
<td>15: <strong>.long</strong> 8</td>
<td>0.3 15: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.1</td>
<td><strong>ret</strong></td>
<td>0.1 23: <strong>hlt</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>90: [AFL trampoline]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: <strong>lea</strong> rax, [rip-92]</td>
<td>107: <strong>mov</strong> rbx, [rax]</td>
</tr>
<tr>
<td>110: <strong>add</strong> rax, rbx</td>
<td>113: <strong>jmp</strong> rax</td>
</tr>
</tbody>
</table>

90: [AFL trampoline]
How we handle the motivation case: **Stochastic Rewriting**

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Initial Rewriting</th>
<th>Binary Cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>lea rax, [rip+8]</td>
<td>jmp 90</td>
</tr>
<tr>
<td>0.0</td>
<td>mov rbx, [rax]</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>add rax, rbx</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>jmp rax</td>
<td>hlt</td>
</tr>
<tr>
<td>0.3</td>
<td>long 8</td>
<td>hlt</td>
</tr>
<tr>
<td>0.1</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>jmp 90</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>long 8</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

**Stochastic Rewriting**

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

**Initial Rewriting**

- 0: `jmp 90`
- 7: `hlt`
- 10: `hlt`
- 13: `hlt`

**Binary Cleaning**

- 0: `jmp 90`
- 7: `hlt`
- 10: `hlt`
- 13: `hlt`
- 15: `.long 8`
- 23: `ret`
STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting

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

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Initial Rewriting</th>
<th>Binary Cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 0: <strong>lea</strong> rax, [rip+8]</td>
<td>0.0 0: <strong>jmp</strong> 90</td>
<td>0.0 0: <strong>jmp</strong> 90</td>
</tr>
<tr>
<td>0.0 7: <strong>mov</strong> rbx, [rax]</td>
<td>0.0 7: <strong>hlt</strong></td>
<td>0.0 7: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.0 10: <strong>add</strong> rax, rbx</td>
<td>0.0 10: <strong>hlt</strong></td>
<td>0.0 10: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.0 13: <strong>jmp</strong> rax</td>
<td>0.0 13: <strong>hlt</strong></td>
<td>0.0 13: <strong>hlt</strong></td>
</tr>
<tr>
<td>0.3 15: <strong>.long</strong> 8</td>
<td>0.3 15: <strong>hlt</strong></td>
<td>??? ??? 15: <strong>.long</strong> 8</td>
</tr>
<tr>
<td>0.1 23: <strong>ret</strong></td>
<td>0.1 23: <strong>hlt</strong></td>
<td>??? ??? 23: <strong>ret</strong></td>
</tr>
</tbody>
</table>

90: [AFL trampoline]
How we handle the motivation case: **Stochastic Rewriting**

### Probability of being data bytes

<table>
<thead>
<tr>
<th>Probability</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>lea rax, [rip+8]</td>
</tr>
<tr>
<td>0.0</td>
<td>mov rbx, [rax]</td>
</tr>
<tr>
<td>0.0</td>
<td>add rax, rbx</td>
</tr>
<tr>
<td>0.3</td>
<td>jmp rax</td>
</tr>
<tr>
<td>0.1</td>
<td>.long 8</td>
</tr>
<tr>
<td>0.0</td>
<td>jmp 90</td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
</tr>
<tr>
<td>0.3</td>
<td>hlt</td>
</tr>
<tr>
<td>0.1</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>jmp 90</td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
</tr>
<tr>
<td>0.0</td>
<td>hlt</td>
</tr>
<tr>
<td>0.1</td>
<td>hlt</td>
</tr>
</tbody>
</table>

### Initial Rewriting

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

### Binary Cleaning

- **90**: [AFL trampoline]
- **100**: lea rax, [rip-92]
- **107**: mov rbx, [rax]
- **110**: add rax, rbx
- **113**: jmp rax

### Binary Mutation

- **0**: jmp 90
- **7**: hlt
- **10**: hlt
- **13**: hlt
- **15**: hlt
- **23**: ret

- **0**: jmp 90
- **7**: hlt
- **10**: hlt
- **13**: hlt
- **15**: hlt
- **23**: ret

- **0**: jmp 90
- **7**: hlt
- **10**: hlt
- **13**: hlt
- **15**: .long 8
- **23**: ret
How we handle the motivation case: **Stochastic Rewriting**

![Stochastic Rewriting Diagram]

**Probability of being data bytes**

<table>
<thead>
<tr>
<th>Probability</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.3</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>lea rax, [rip+8]</td>
<td>mov rbx, [rax]</td>
<td>add rax, rbx</td>
<td>jmp rax</td>
<td>.long 8</td>
<td>ret</td>
</tr>
</tbody>
</table>

**Initial Rewriting**

- 0: jmp 90
- 7: hlt
- 10: hlt
- 13: hlt
- 15: .long 8
- 23: hlt

**Binary Cleaning**

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

**Binary Mutation**

- 1:0 15: hlt
- 0: jmp 90
- 7: hlt
- 10: hlt
- 13: hlt
- 15: .long 8
- 23: hlt
- 0: jmp 90
- 7: hlt
- 10: hlt
- 13: hlt
- 15: .long 8
- 23: hlt
How we handle the motivation case: **Stochastic Rewriting**

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Binary Cleaning</th>
<th>Binary Mutation</th>
<th>Binary Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 0: lea rax, [rip+8]</td>
<td>0.0 0: jmp 90</td>
<td>1.0 15: hlt</td>
<td>0.0 0: jmp 90</td>
</tr>
<tr>
<td>0.0 7: mov rbx, [rax]</td>
<td>100: lea rax, [rip-92]</td>
<td>0.0 7: hlt</td>
<td>0.0 0: jmp 90</td>
</tr>
<tr>
<td>0.0 10: add rax, rbx</td>
<td>107: mov rbx, [rax]</td>
<td>0.0 10: hlt</td>
<td>0.0 0: jmp 90</td>
</tr>
<tr>
<td>0.0 13: jmp rax</td>
<td>110: add rax, rbx</td>
<td>0.0 13: hlt</td>
<td>0.0 0: jmp 90</td>
</tr>
<tr>
<td>0.3 15: .long 8</td>
<td>113: jmp rax</td>
<td>0.3 15: hlt</td>
<td>?? 15: .long 8</td>
</tr>
<tr>
<td>0.1 23: ret</td>
<td>0.1 23: hlt</td>
<td>0.1 23: hlt</td>
<td>?? 23: ret</td>
</tr>
</tbody>
</table>
How we handle the motivation case: **Stochastic Rewriting**

Initial Rewriting

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>lea rax, [rip+8]</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>mov rbx, [rax]</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>add rax, rbx</code></td>
<td>10:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>jmp rax</code></td>
<td>13:</td>
</tr>
<tr>
<td>0.3</td>
<td><code>.long 8</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.1</td>
<td><code>ret</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

Refine the probabilities based on these new hints

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>jmp 90</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.3</td>
<td><code>hlt</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

Binary Cleaning

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>jmp 90</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.3</td>
<td><code>hlt</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

Binary Mutation

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>jmp 90</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

Binary Mutation

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>jmp 90</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

Binary Cleaning

<table>
<thead>
<tr>
<th>Probability of being data bytes</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><code>jmp 90</code></td>
<td>0:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>7:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>15:</td>
</tr>
<tr>
<td>0.0</td>
<td><code>hlt</code></td>
<td>23:</td>
</tr>
</tbody>
</table>

**STOCHFUZZ**: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting
Universal Control-flow Graph (\textit{UCFG}) and Probability Analysis
Universal Control-flow Graph (UCFG) and Probability Analysis

<table>
<thead>
<tr>
<th>Addr</th>
<th>Byte</th>
<th>[Len]</th>
<th>Decoded Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>[3]</td>
<td>xor rcx, rcx</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>[2]</td>
<td>xor ecx, ecx</td>
</tr>
<tr>
<td>2</td>
<td>c9</td>
<td>[1]</td>
<td>leave</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>[4]</td>
<td>cmp rcx, 5</td>
</tr>
<tr>
<td>4</td>
<td>83</td>
<td>[3]</td>
<td>cmp ecx, 5</td>
</tr>
<tr>
<td>5</td>
<td>f9</td>
<td>[1]</td>
<td>stc</td>
</tr>
<tr>
<td>6</td>
<td>05</td>
<td>[0]</td>
<td>INVALID</td>
</tr>
<tr>
<td>7</td>
<td>c3</td>
<td>[1]</td>
<td>ret</td>
</tr>
</tbody>
</table>
Universal Control-flow Graph (UCFG) and Probability Analysis

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

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 crashes 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 crashes 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-Oemu: 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-Oemu:* 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 \((\text{free} \text{type}2)\)

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

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

```c
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*)

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

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

**AFL Instrumentation**

```c
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:**

**Probabilistic Program Analysis:**
STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting
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