Binvariants:

Enhancing Fuzzing of Closed-source Binary Executables via Register-level Likely Invariants

Zao Yang
University of Utah

Stefan Nagy
University of Utah

FUTURES^3
LAB FUTURE TECHNOLOGY FOR USABLE, RELIABLE, & EFFICIENT SECURITY OF SOFTWARE & SYSTEMS

Software Security: Vetting Closed-source Code

Government

Telecommunications

Finance

- Among today's **most highly targeted software** by attackers
- Opaque internals impedes **third-party static code analysis**
- **Solution:** finding bugs through **dynamic analysis** (e.g., fuzzing)

Securing Binaries: Coverage-guided Fuzzing

- **Key idea:** drive testing by mutating inputs that reach **new code regions**

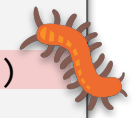
```
(Abridged source code of TCPDump's print-arp.c)

L1  static u_char ezero = [6];
L2  ndp((ndo, "%s", str(tpa(ap))));
L3  if (memcmp(ezero, tha(ap), ap->hln) != 0)
L4    ndp((ndo,"(%s)",str(tha(ap),la,ap->hln)));
```

Input	CodeCov	Outcome	Data State
01	L1 – L4	Saved	ap->hln = 3
02	L1 – L4	Discard	ap->hln = 3

Securing Binaries: Coverage-guided Fuzzing

- **Key idea:** drive testing by mutating inputs that reach **new code regions**

```
(Abridged source code of TCPDump's print-arp.c)

L1  static u_char ezero = [6];
L2  ndp((ndo, "%s", str(tpa(ap))));
L3  if (memcmp(ezero, tha(ap), ap->hln) != 0)
L4    ndp((ndo,"(%s)",str(tha(ap),la,ap->hln)));
```



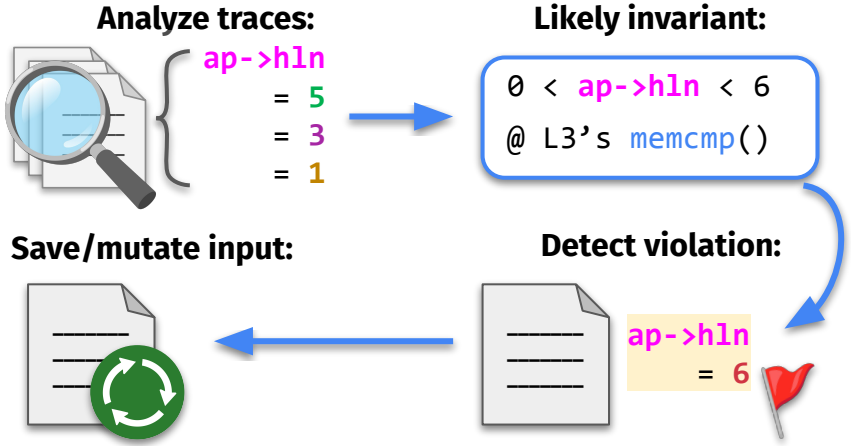
Input	CodeCov	Outcome	Data State
01	L1 – L4	Saved	ap->hln = 3
02	L1 – L4	Discard	ap->hln = 3
...			
99	L1 – L4	Discard	ap->hln = 6

Limitation: focusing strictly on **coverage-level feedback** often **misses data-level state changes** that precede bugs!

Beyond Code Coverage: Likely Data Invariants

- **Key idea:** additionally mutate inputs that reveal **novel data-level states**
  - Detect **violations of expected properties** captured over pre-fuzzing program traces

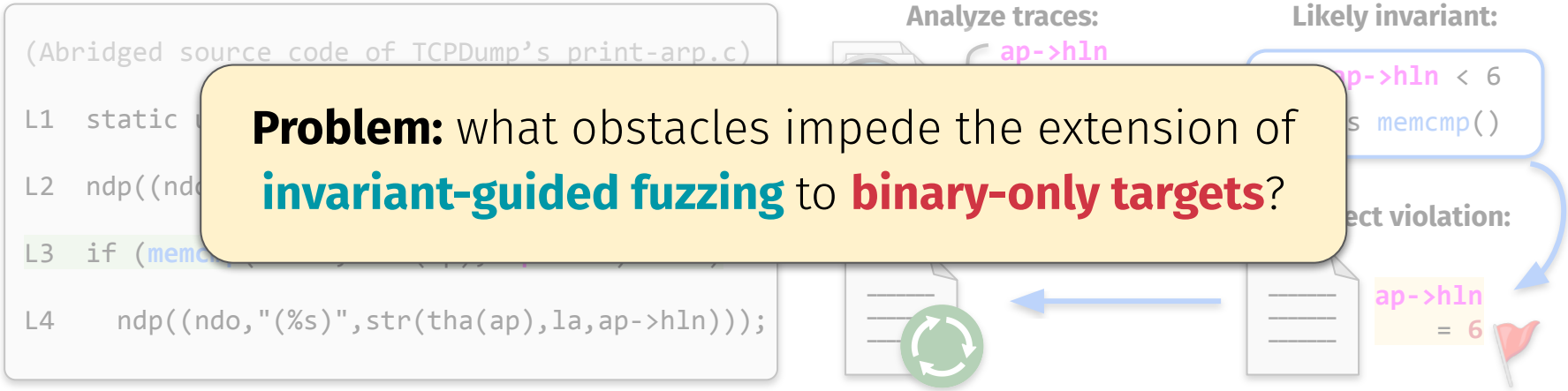
```
(Abridged source code of TCPDump's print-arp.c)

L1  static u_char ezero = [6];

L2  ndp((ndo, "%s", str(tpa(ap))));

L3  if (memcmp(ezero, tha(ap), ap->hln) != 0)

L4    ndp((ndo,"(%s)",str(tha(ap),la,ap->hln)));
```



- **Outcome:** discovery of numerous bugs missed by code coverage alone
  - **Source-level approaches:** InvsCov (Fioraldi et al., 2021), Halo (Huang et al., 2024)

Beyond Code Coverage: Likely Data Invariants

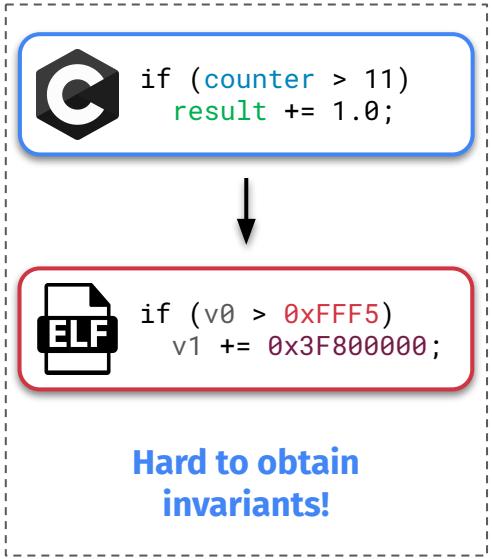
- **Key idea:** additionally mutate inputs that reveal **novel data-level states**
  - Detect **violations of expected properties** captured over pre-fuzzing program traces



Problem: what obstacles impede the extension of **invariant-guided fuzzing** to **binary-only targets**?

- **Outcome:** discovery of numerous bugs missed by code coverage alone
  - **Source-level approaches:** InvsCov (Fioraldi et al., 2021), Halo (Huang et al., 2024)

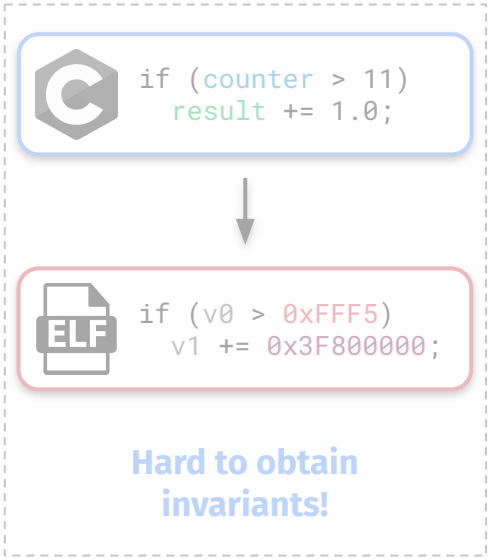
Challenges: Invariant-guided Fuzzing for Binaries



1. Invariant Inference

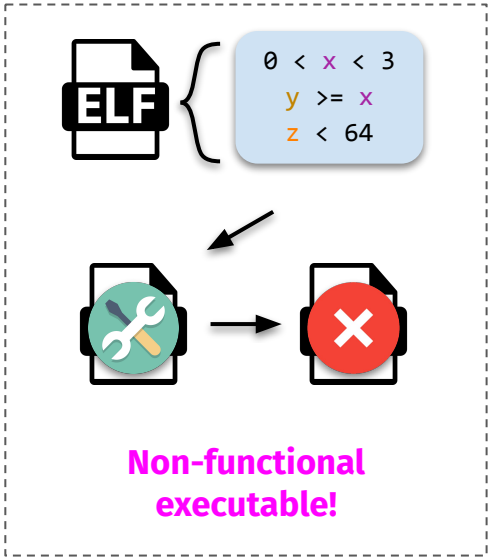
(difficulty of introspection)

Challenges: Invariant-guided Fuzzing for Binaries



1. Invariant Inference

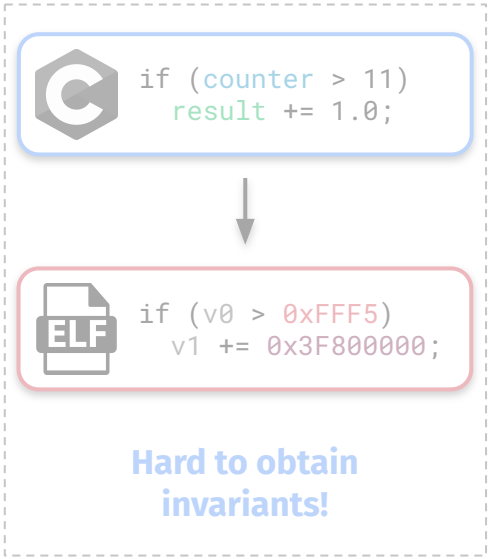
(difficulty of introspection)



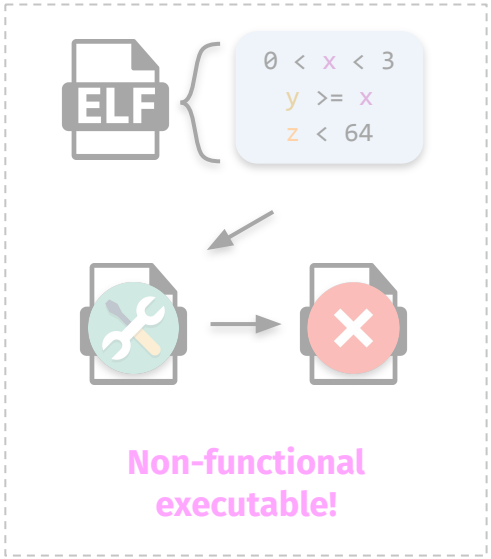
2. Invariant Injection

(avoid breaking the code)

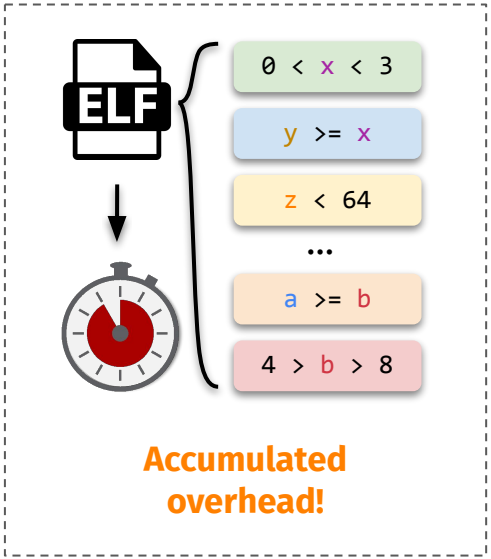
Challenges: Invariant-guided Fuzzing for Binaries



1. Invariant Inference
(difficulty of introspection)

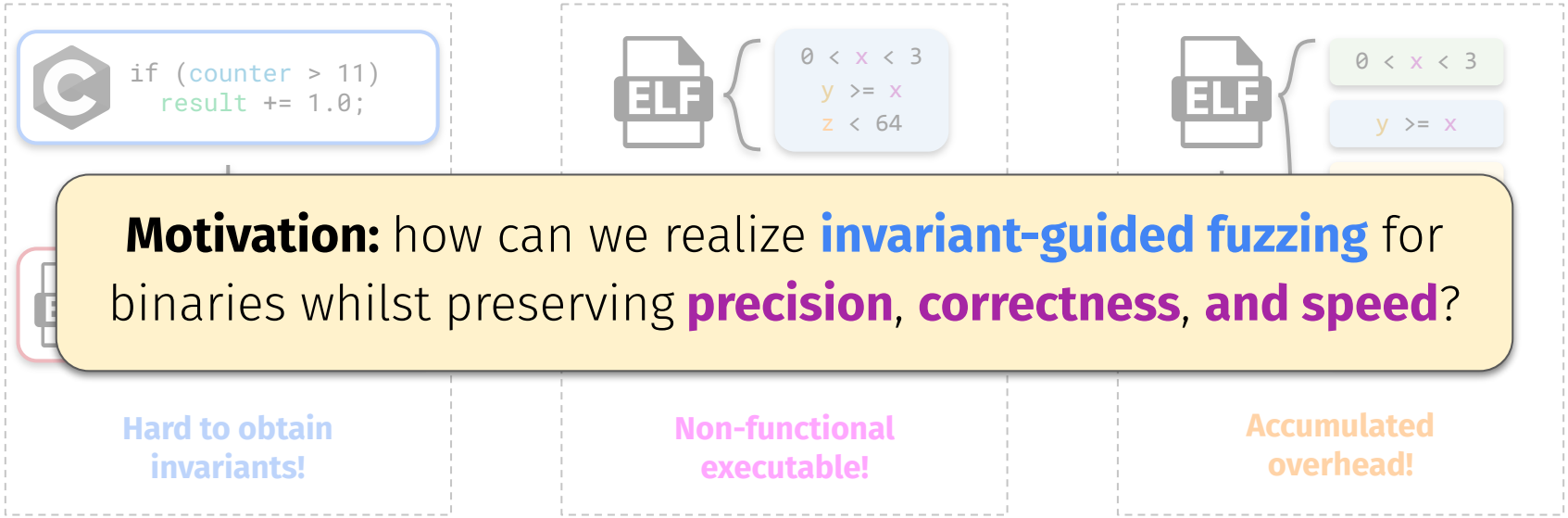


2. Invariant Injection
(avoid breaking the code)



3. Upholding Speed
(eliminate noise/bloat)

Challenges: Invariant-guided Fuzzing for Binaries



Motivation: how can we realize **invariant-guided fuzzing** for binaries whilst preserving **precision**, **correctness**, **and speed**?

1. Invariant Inference
(difficulty of introspection)

2. Invariant Injection
(avoid breaking the code)

3. Upholding Speed
(eliminate noise/bloat)

Our Solution: Register-level Likely Invariants

- **Intuition:** program data-level state is ultimately reflected in **CPU registers**
  - On x86-64: `rax–rdx`, `r8–r15`, `rsi`, `rdi`

```
L1  static u_char ezero = [6];

L2  ndp((ndo, "%s", str(tpa(ap))));

L3  if (memcmp(ezero, tha(ap), ap->hln) != 0)

L4   ndp((ndo,"(%s)",str(tha(ap),la,ap->hln)));
```

```
0x35be6    mov    rdx,  rbx
0x35be9    mov    byte [rsp+0x10],  bl
0x35bed    add    r14,  rbp
0x35bf0    mov    rsi,  r14
0x35bf3    call   memcmp
0x35bf8    movzx  ecx,  byte [rsp+0x10]
```

Our Solution: Register-level Likely Invariants

- **Intuition:** program data-level state is ultimately reflected in **CPU registers**
  - On x86-64: `rax–rdx`, `r8–r15`, `rsi`, `rdi`

```
L1  static u_char ezero = [6];
L2  ndp((ndo, "%s", str(tpa(ap))));
L3  if (memcmp(ezero, tha(ap), ap->hln) != 0)
L4   ndp((ndo,"(%s)",str(tha(ap),la,ap->hln)));
```

```
0x35be6    mov    rdx,  rbx              // ap->hln
0x35be9    mov    byte [rsp+0x10],  bl
0x35bed    add    r14,  rbp
0x35bf0    mov    rsi,  r14
0x35bf3    call   memcmp
0x35bf8    movzx  ecx,  byte [rsp+0x10]
```

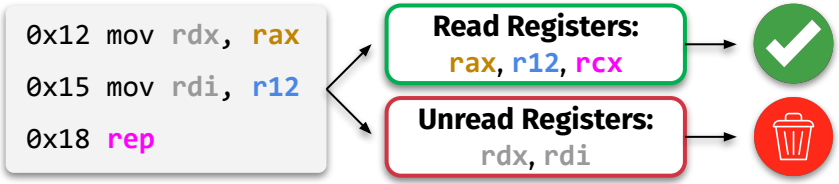
Likely invariant: `ap->hln < 6 @ memcmp()`

Likely invariant: `rbx < 6 @ 0x35be6`

- **Opportunity:** infer source-like invariants **over register-level state abstraction**

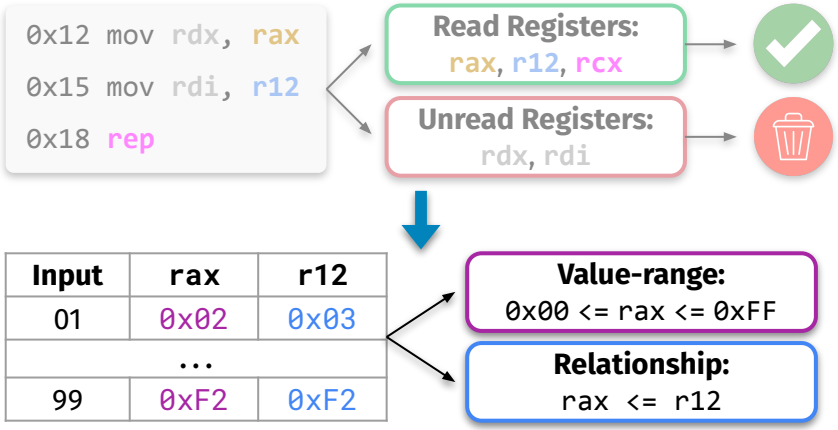
Our Solution: Register-level Likely Invariants

- **Phase 1: capturing likely invariants**
  - Target only **read registers** per block
  - Including **explicit** and **implicit reads**



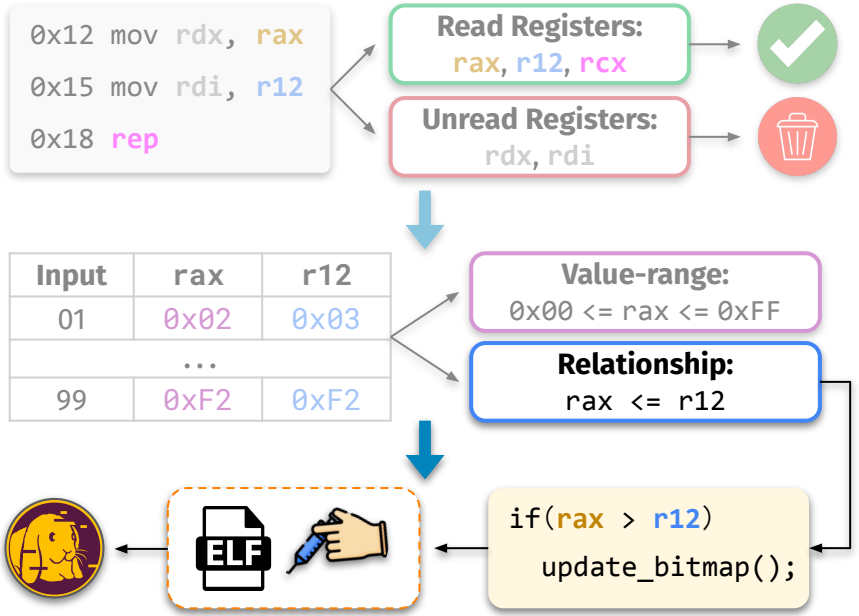
Our Solution: Register-level Likely Invariants

- **Phase 1: capturing likely invariants**
  - Target only **read registers** per block
  - Including **explicit** and **implicit reads**
  - **Value ranges** on **individual** registers
  - **Relationships** among **two registers**



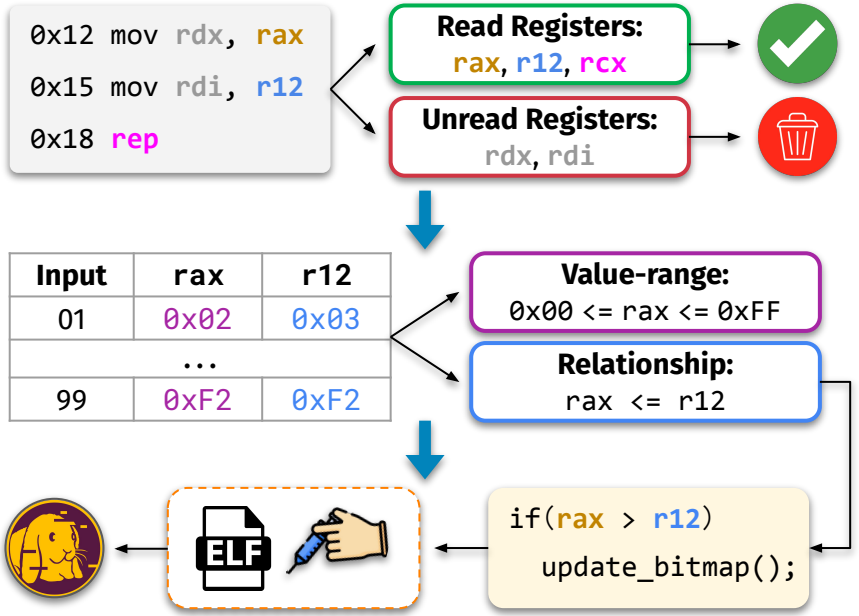
Our Solution: Register-level Likely Invariants

- **Phase 1: capturing likely invariants**
  - Target only **read registers** per block
  - Including **explicit** and **implicit reads**
  - **Value ranges** on **individual** registers
  - **Relationships** among **two registers**
- **Phase 2: injection & binary fuzzing:**
  - Via dynamic binary translation (QEMU)
  - Inlined as **fuzzer-reachable conditions**
  - If violated, trigger fuzzer bitmap update (analogous to new code coverage signal)



Our Solution: Register-level Likely Invariants

- **Phase 1: capturing likely invariants**
  - Target only **read registers** per block
  - Including **explicit** and **implicit reads**
  - **Value ranges** on **individual** registers
  - **Relationships** among **two registers**
- **Phase 2: injection & binary fuzzing:**
  - Via dynamic binary translation (QEMU)
  - Inlined as **fuzzer-reachable conditions**
  - If violated, trigger fuzzer bitmap update (analogous to new code coverage signal)



- **Outcome:** the first invariant-guided fuzzing approach **for binary-only targets**

Evaluation: Overview

- **Evaluation questions:**
  - Can register-level likely invariants **enhance fuzzing's exploration**?
  - Do register-level likely invariants **improve binary-level bug finding**?
  - Are register-level likely invariants **comparable to source-level ones**?
- **Implementation: Binvariants**
  - Built atop AFL++ and QEMU DBT
- **Competing fuzzing approaches:**
  - **CodeCov:** coverage-only binary fuzzing (AFL++ QEMU)
  - **InvsCov:** source-level invariant-guided fuzzing

Evaluation benchmark corpus: **25 binaries**	
Open-source binaries (**18**)	cert-basic, exiv2, gpmf-demo, hdf5, imginfo, jasper, lame, mp4split, opj_decompress, pdfimages, pdftops, pdftotext, readelf, sfconvert, stormlib, tcpdump, tiff2bw, xls2csv
Closed-source binaries (**7**)	cpdf, cuobjdump, IDA Pro, lzturbo, nconvert, nvdisasm, pngout

Evaluation: Code Exploration

- Measured **violated invariants** and **code coverage** vs. coverage-only fuzzing
  - Averaged across **five 48-hour fuzzing campaigns** per binary
  - Coverage collected as basic block coverage via AFL-QEMU-Cov

Metric	**Binvariants' Rel. Total** vs. CodeCov	**Binvariants' Rel. Distinct** vs. CodeCov
Violated likely invariants	**4.10 ×**	**28.11 ×**
Basic block coverage	**1.01 ×**	**1.52 ×**

- **Takeaway:** register-level likely invariants surface **qualitatively different states**
  - Expands binary fuzzing's reach into regions ordinarily untested by coverage-only fuzzing

Evaluation: Bug Discovery

- Measured **total and distinct bugs found** throughout binary-level fuzzing
  - Deduplicated via ASAN (open-source targets) and QEMU-ASAN (closed-source targets)

Competitor	Total Bugs	Distinct Bugs
CodeCov	**137**	**14**
Binvariants	**143**	**20**

- **Takeaways:** register-level likely invariants **enhance binary bug discovery**
  - **46.34%** of mutually-discovered bugs are reached in less time
  - **18 bugs** are traceable back to distinct likely invariant violations

Evaluation: Correspondence to Source-level Invariants

- Compared **register-** vs. **source-level invariants** on 10 open-source targets
  - Measured source-level invariant violations, code coverage, bug discovery

Competitor	Value Range Single–var Invariants	Relationship Two–var Invariants	Total Bugs Found	Distinct Bugs Found
InvsCov	38.25%	52.45%	61	9
Binvariants	63.40%	30.63%	59	7

Binvariants' Rel. Total Coverage	Binvariants' Rel. Distinct Coverage
1.01 ×	2.78 ×

- **Takeaways:** register-level likely invariants **reveal distinct program states**
  - Bias toward single- versus multi-variable invariants exposes **different groups of bugs**
  - Impact on fuzzer queue size (**1–67%**) on-par with source-level invariants' (**mean 62%**)

Case Studies: Binvariants-found Bugs

```
nshorts = R13 / VAL;
while (--nshorts >= 0) *ptr++;
```

```
Invariant:   0x10 ≤ R13 ≤ 0xFFFF
Crash value: R13 = 0x2008FB
```

TCPDump: **unbounded loop counter**

```
dest = base + (int32)RSI;
var = MEM[dest];
```

```
Invariant:   0 ≤ RSI ≤ 0xFF
Crash value: RSI = 0xF292BC7B
```

PNGOUT: **signed offset underflow**

```
size = (int32)RSI * VAL;
memset(buf, 0, size);
```

```
Invariant:   0 ≤ RSI ≤ 0xFFFF
Crash value: RSI = 0x40000032
```

PNGOUT: **unchecked memory write**

```
while (++iter <= RSI)
      MEM[0x914970 +iter*4] = 0;
*MEM[0x916528] = val;
```

```
Invariant:   0 ≤ RSI ≤ 0xFF
Crash value: RSI = 0xFF05
```

NConvert: **pointer overwrite**

Case Studies: Binvariants-found Bugs

```
nshorts = R13 / VAL;
while (--nshorts >= 0) *ptr++;
```

```
size = (int32)RSI * VAL;
memset(buf, 0, size);
```

```
Invariant:   0 ≤ RSI ≤ 0xFFFF
Crash value: RSI = 0x40000032
```

PNGOUT: **unchecked memory write**

```
dest = base + (int32)RSI;
var = MEM[dest];
```

```
MEM[0x914970 +iter*4] = 0;
*MEM[0x916528] = val;
```

```
Invariant:   0 ≤ RSI ≤ 0xFF
Crash value: RSI = 0xFF05
```

NConvert: **pointer overwrite**

Takeaway: register-level likely invariants complement coverage-only binary-level fuzzing by **revealing new states** and **triggering new bugs**.

Conclusion: Why Register-Level Likely Invariants?

- As fuzzing moves beyond coverage-only search, **likely data invariants** expose new states & bugs
- **Problem:** despite its benefits, likely invariants remain restricted to **only open-source targets**
- **Our solution: Register-level Likely Invariants**
  - Infer source-like invariants **directly from CPU registers**
  - Mitigate noise by focusing solely on **read registers**
  - Inject into fuzzing and **flag violations** as new feedback
  - **Outcome:** exposure of **qualitatively different bugs** and **coverage regions** versus coverage-only binary fuzzing

Key Results:

4.1× total and **28×** distinct violations,

1.52× distinct code coverage,

20 bugs missed by coverage alone

Thank you!

github.com/FuturesLab/Binvariants

Contact:
zao.yang@utah.edu
@yangzaocn
@zaoyang.bsky.social

FUTURES³
LAB FUTURE TECHNOLOGY FOR USABLE, RELIABLE, & EFFICIENT SECURITY OF SOFTWARE & SYSTEMS