Format

- Welcome
- Discuss Confidentiality
- Issue Clearing (if required)
- Get Presence
- Meaningful Update & Follow-Up
- Review
- Presentation
- Feedback

REVelry RedQueen

Fuzzing w/ Input-to-State Correspondence

About Me | Jay Warne

Currently

- DARPA research
  - Side-channels
  - Processors
  - Hypervisors
  - Rowhammer Style Attacks
  - Program Analysis Methods
- Project Work
  - PO for Videographic Data Analysis
  - Routine Software Engineering
- Advisory
  - Product Development
  - Product Direction
  - Expert Reviewer

Previously

- Ran a Security Ops Team
- Occasional RE & Red Team
- Field Forensic Analysis & Tool Deployment

Things I Like

- Alpine Ski Racing
- Ski/ Alpine Mountaineering
- Ice/ Rock Climbing
- Surfing
- Backpacking

RedQueen – tl;dr

- Code coverage
- Input-to-State correspondence
- Uses those values to explore branches that wouldn’t have otherwise been reached

That is build into kAFL

If you just want to use it,
grab kAFL from IntelLabs

What We Will Cover

- Common Fuzzing Roadblocks
- High-level overview of recent research
  - Symbolic Execution
  - Taint-based Fuzzing
  - Patching-based Fuzzing
  - AFL Family of Fuzzers
- RedQueen’s Solution: Input-to-State Correspondence
  - What it solves
  - Where it fails
- Results
- Implementation Details Q&A

Common Fuzzing Roadblocks

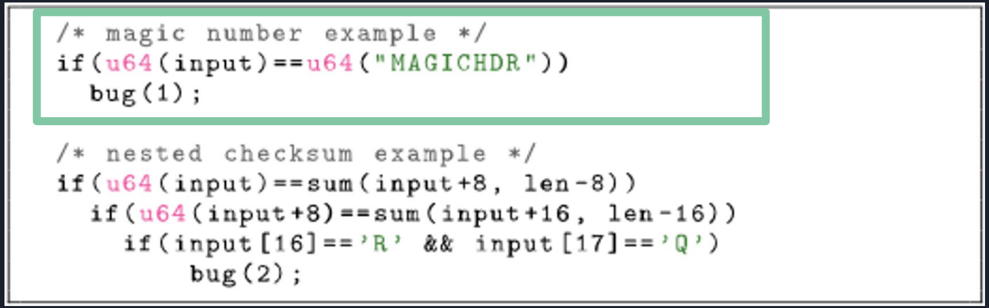
Common Fuzzing Roadblocks

Common Fuzzing Roadblocks

1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data

Common Fuzzing Roadblocks

1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data

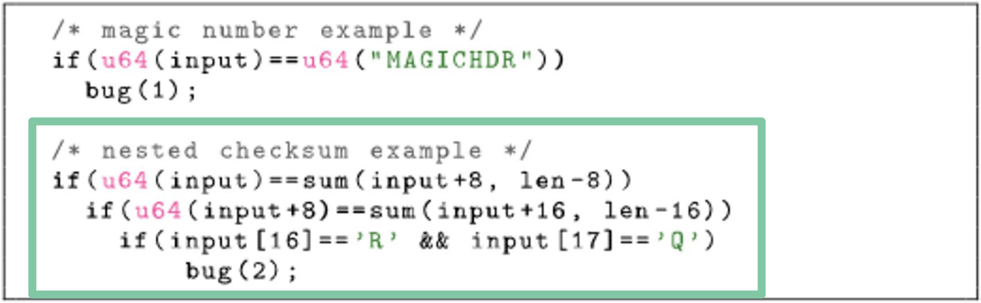


```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);

/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]==’R’ && input[17]==’Q’)
        bug(2);
```

Common Fuzzing Roadblocks

1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);

/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
        bug(2);
```

Common Fuzzing Roadblocks

1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data
5) Encoded Data

Common Fuzzing Roadblocks

1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data
5) Encoded Data

Input-to-State fails to solve these cases because the input doesn't doesn't correspond to the state after the "transformation"

Current Approaches

Symbolic Execution

SymbEx assigns abstract values for variables

```
x = <unknown input>
y = 3x+5
If ( y > 11 ) {
    // Block A
} else {
    // Block B
}
```

- X is treated as a symbolic value
- Y is 3(symbVal) + 5

Here we see two branches:

1) $3x+5 > 11$
2) $3x+5 <= 11$

Symbolic Execution

SymbEx assigns abstract values for variables

```
x = <unknown input>
y = 3x+5
If ( y > 11 ) {
    // Block A
} else {
    // Block B
}
```

- X is treated as a symbolic value
- Y is 3(symbVal) + 5

Here we see two branches:

1) x > 2
2) x <= 2

Symbolic Execution

SymbEx assigns abstract values for variables

```
x = <unknown input>
y = 3x+5
If ( y > 11 ) {
    // Block A
} else {
    // Block B
}
```

Struggles with

- Path Explosion
- Large Targets (libraries)
- Memory Operations
- Arrays
- Environment

Taint Tracking

- Effectively Data Flow Analysis

Taint Tracking

- Effectively Data Flow Analysis
- Data from untrusted sources

Taint Tracking

- Effectively Data Flow Analysis
- Data from untrusted sources

Sorta mediocre on it’s own… Left up to the implementation

- Undertainting
- Overtainting

Patching-based Fuzzing

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

All inputs are stored here

Inputs are:

- Taken off the queue
- Fuzzed for a while
- Returned to the queue

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

Any coverage produced by an input is recorded here.

If found found, the input is added to the queue

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

One input might have numerous stages of mutators applied to it:

- Deterministic
- Havoc
- Splicing

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

One input might have numerous stages of mutators applied to it:

Applied once

- Deterministic
- Havoc
- Splicing

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

Multiple random mutations at the same time in random places

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

One input might have numerous stages of mutators applied to it:

- Deterministic
- Havoc
- Splicing

Binary Only Fuzzing

- Feedback required for efficiency
- AFL Family Fuzzers

AFL Family Makeup:

1) Queue
2) Bitmap
3) Mutators

One input might have numerous stages of mutators applied to it:

- Deterministic
- Havoc
- Splicing

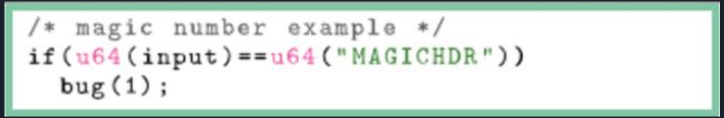
Two inputs are combined at a random position

RedQueen: Input-to-State Correspondence

Common Fuzzing Roadblocks

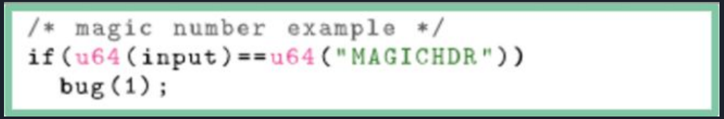
1) Magic Numbers
2) Checksum Tests
3) Hash Maps
4) Compressed Data

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

Magic Bytes



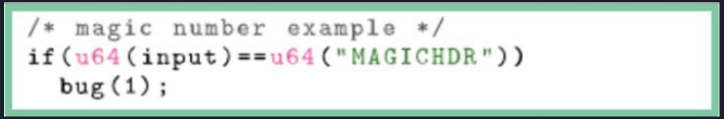
```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

How does RedQueen handle these?

Values from the program state often correspond with the input, so:

- Hook all compares instructions and perform a single trace run
- If a comparison w/ arguments found → create a custom mutation

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

How does RedQueen handle these?

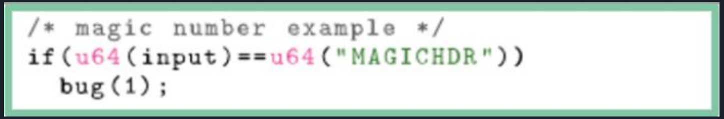
Values from the program state often correspond with the input, so:

- Hook all compares instructions and perform a single trace run
- If a comparison w/ arguments found → create a custom mutation

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



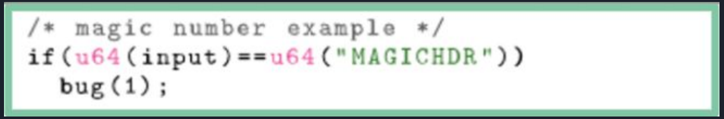
```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

- Hooks
  - Compares
  - Jump Table Offsets
  - Calls
- Perform a single run with hooks in these locations

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

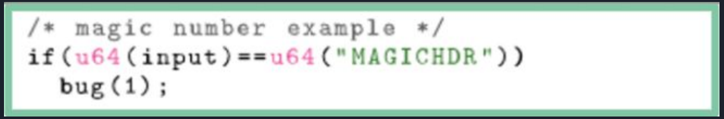
Can’t differentiate between compares

Consider >, <, and == : How might one test for those without knowing the flags?

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

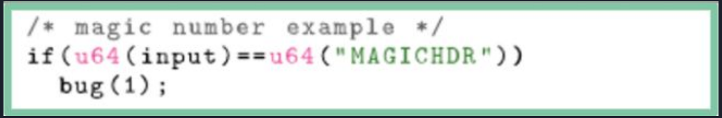
Inputs often get processed at some point; maybe before the compare

- Zero/Sign Extend
- Endianness
- C-String
- Size of Pointer
- ASCII

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

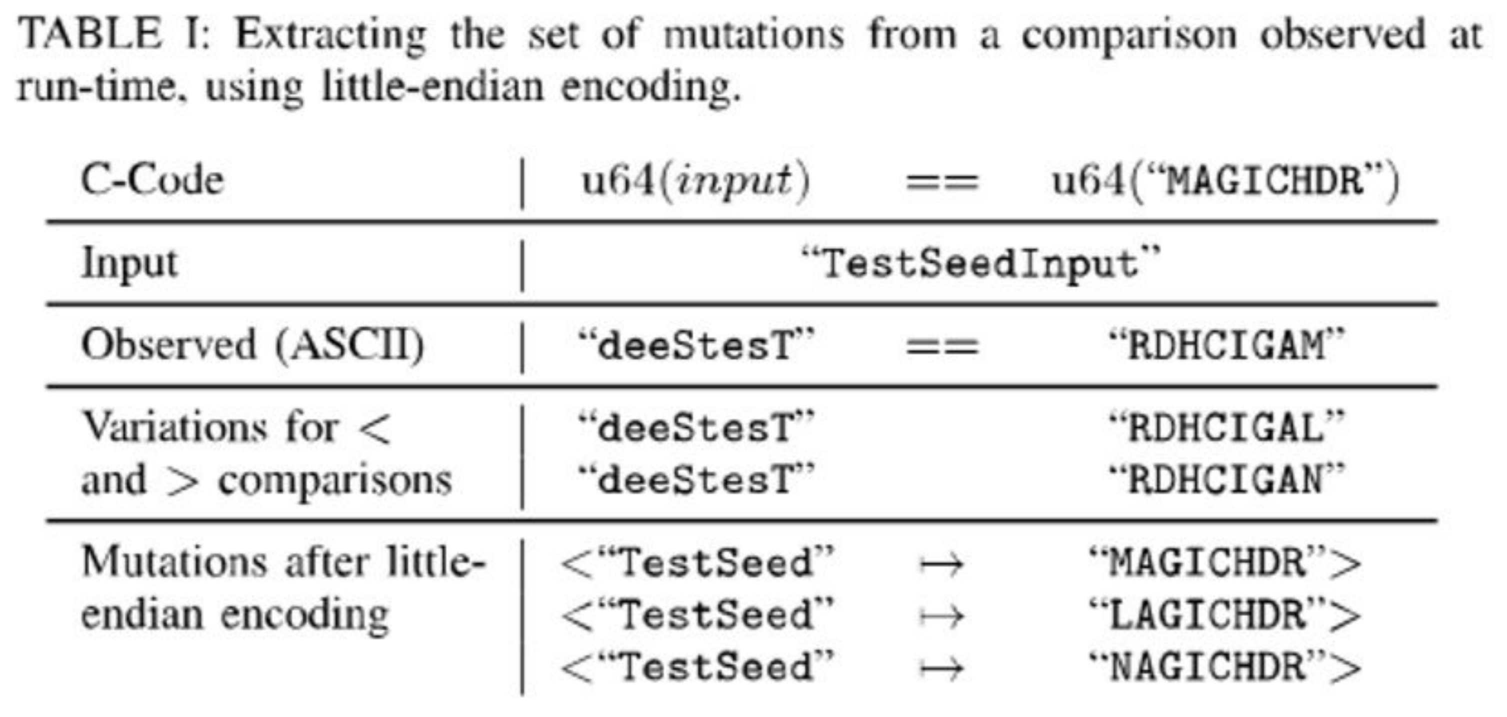
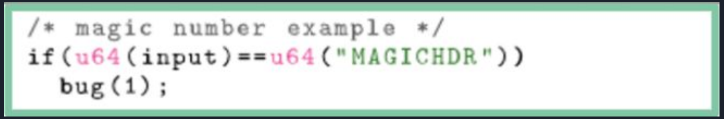


TABLE I: Extracting the set of mutations from a comparison observed at run-time, using little-endian encoding.

C-Code	u64(*input*)	==	u64("MAGICHDR")
Input	"TestSeedInput"		
Observed (ASCII)	"deeStesT"	==	"RDHCIGAM"
Variations for < and > comparisons	"deeStesT" "deeStesT"		"RDHCIGAL" "RDHCIGAN"
Mutations after little-endian encoding	<"TestSeed" <"TestSeed" <"TestSeed"	↦ ↦ ↦	"MAGICHDR"> "LAGICHDR"> "NAGICHDR">

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

Only the portion of a string subject to a compare is replaced with mutations, and all at once

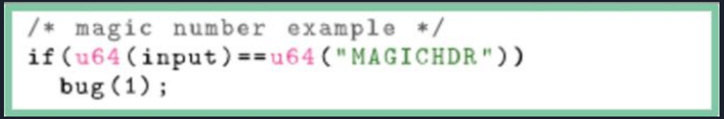
This speeds up the process by:

- Reducing the number of candidate positions
- Not requiring further modification and hooking of the target

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes

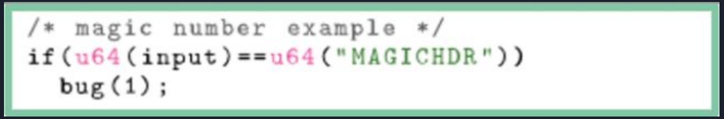


```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

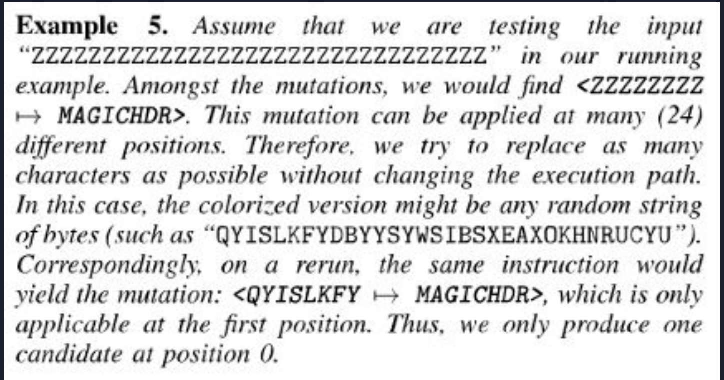
Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

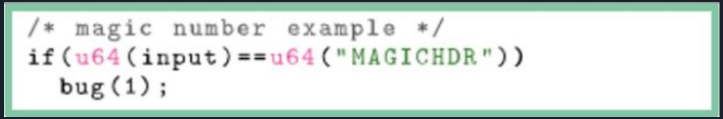


Example 5. Assume that we are testing the input “ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ” in our running example. Amongst the mutations, we would find <ZZZZZZZZ ↦ MAGICHDR>. This mutation can be applied at many (24) different positions. Therefore, we try to replace as many characters as possible without changing the execution path. In this case, the colorized version might be any random string of bytes (such as “QYISLKFYDBYYSYWSIBSXEAXOKHNRUCYU”). Correspondingly, on a rerun, the same instruction would yield the mutation: <QYISLKFY ↦ MAGICHDR>, which is only applicable at the first position. Thus, we only produce one candidate at position 0.

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

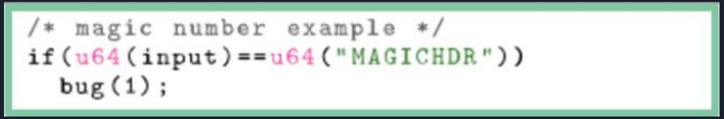
Func takes at least two pointer arguments, the first 128 bytes are extracted:

- First 4-32 bytes a la memcmp
- All bytes up to 0 a la strcmp

Steps:

- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Magic Bytes



```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

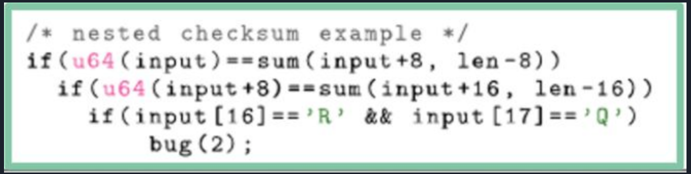
Just lots of non-zero and non-0xff bytes

Sorta just more robust strings – only used during havoc

Steps:

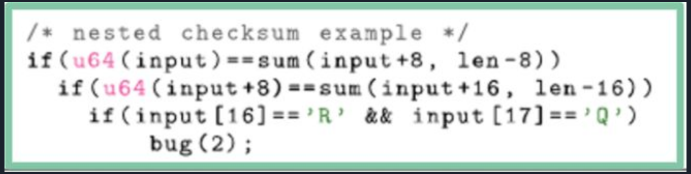
- Tracing
- Variations
- Encodings
- Application
- Colorization
- Strings & Memory
- Input Specific Dictionary

Checksums



```
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]==’R’ && input[17]==’Q’)
        bug(2);
```

Checksums



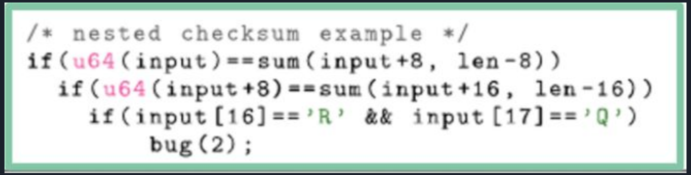
```
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
        bug(2);
```

Identify Comparisons → list of comparisons and values compared in all different colorized versions of the input

Filter comparisons on:

1) Able to find left side of mutation pattern in all inputs using the same encoding
2) No immediate value arguments
3) Pattern changes during colorization

Checksums



```
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
        bug(2);
```

Identify Comparisons → list of comparisons and values compared in all different colorized versions of the input

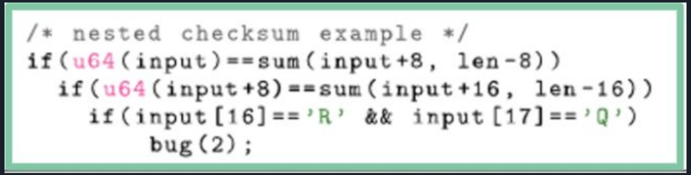
Filter comparisons on:

1) Able to find left side of mutation pattern in all inputs using the same encoding
2) No immediate value arguments
3) Pattern changes during colorization

Drawbacks:

- Might identify a misidentify important compares due to over-approximation

Checksums



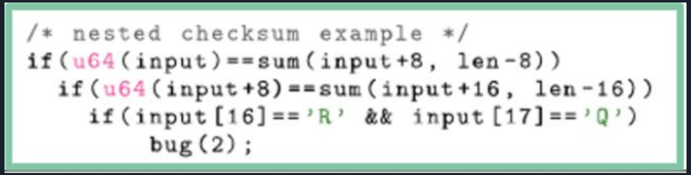
```
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]==’R’ && input[17]==’Q’)
        bug(2);
```

Step 2 is Patching…

Might patch out some bounds check or make paths reachable that really aren’t… blah blah

Checksums

Step 3 is Verification…



```
/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]==’R’ && input[17]==’Q’)
        bug(2);
```

Still same as Magic Byte approach…

Results

Results

LAVA-M is the test set that we are looking at… Synthetic bugs inserted in hard to reach places in GNU coreutils

It’s a standard test set

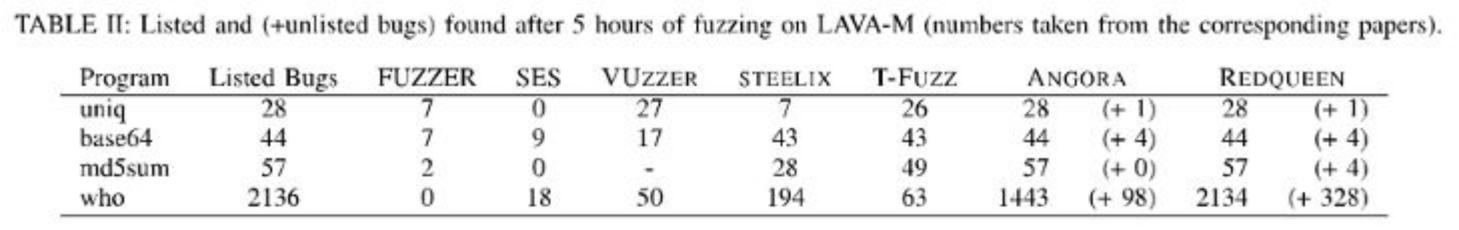


TABLE II: Listed and (+unlisted bugs) found after 5 hours of fuzzing on LAVA-M (numbers taken from the corresponding papers).

Program	Listed Bugs	FUZZER	SES	VUZZER	STEELIX	T-FUZZ	ANGORA		REDQUEEN	
uniq	28	7	0	27	7	26	28	(+ 1)	28	(+ 1)
base64	44	7	9	17	43	43	44	(+ 4)	44	(+ 4)
md5sum	57	2	0	-	28	49	57	(+ 0)	57	(+ 4)
who	2136	0	18	50	194	63	1443	(+ 98)	2134	(+ 328)

Results

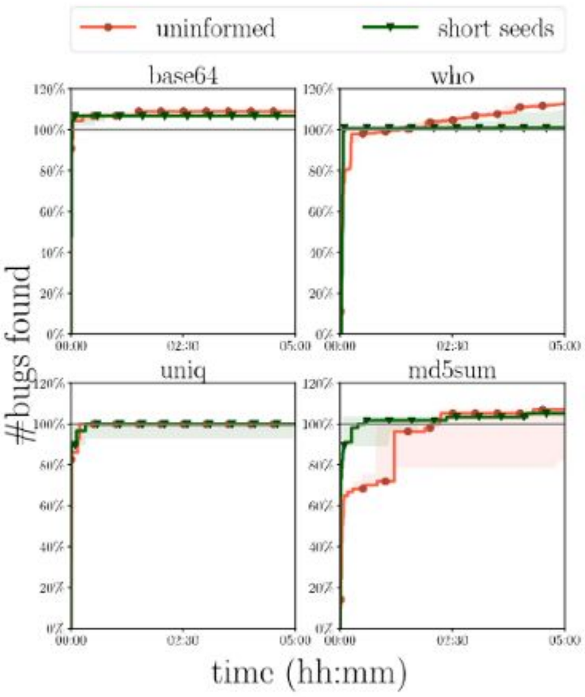


Fig. 1: Evaluating LAVA-M on informed and uninformed seeds.

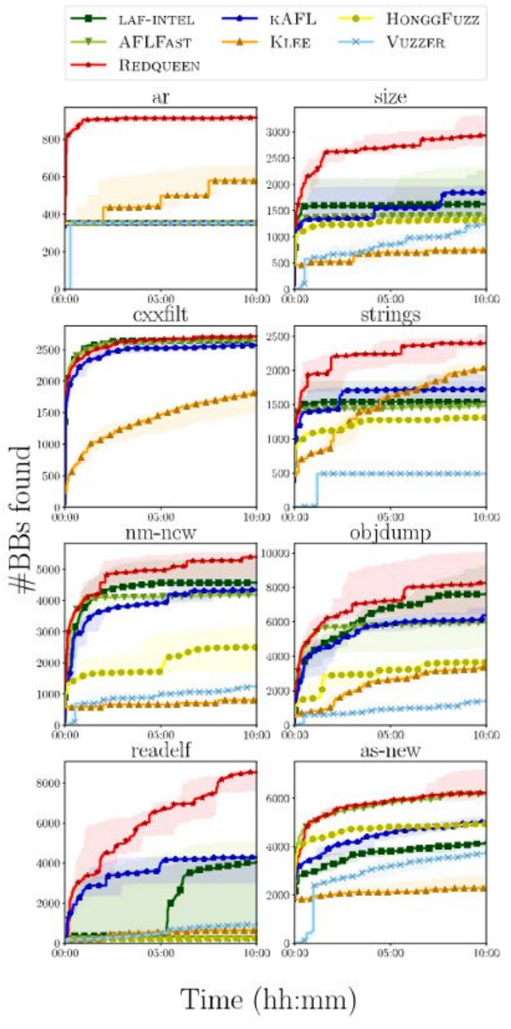


Fig. 2: The coverage (in basic blocks) produced by various tools over 5 runs at 10 h each on the binutils. Displayed are the median and the 60 % intervals.

Implementation Details