[1] list université PARIS-SACLAY
USC University of Southern California [2]

Fine-Grained Coverage-Based Fuzzing

Wei-Cheng Wu [1,2] *Bernard Nongpoh* [1] *Marwan Nour* [1]
Michaël Marcozzi [1] *Sébastien Bardin* [1] *Christophe Hauser* [2]

to appear in
ACM Transactions On Software Engineering and Methodology

This work has been mainly carried out by...

Wei-Cheng Wu

Ph.D. student

(also @ USC in Los Angeles)

Dr. Benard Nongpoh

Postdoc

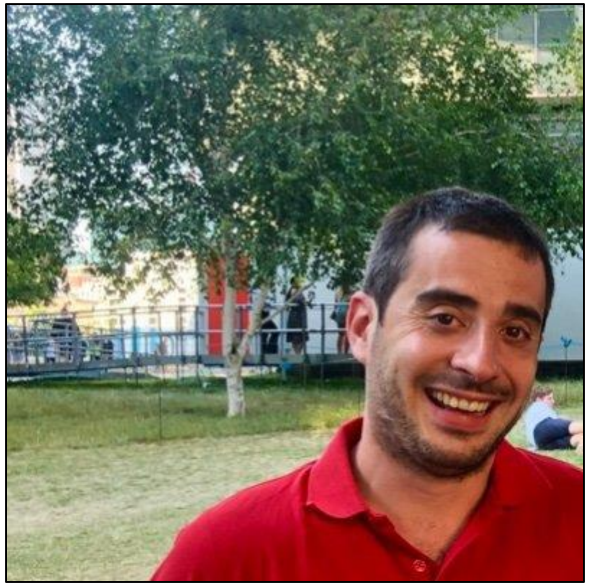
(now @ Qualcomm)

Marwan Nour

M.Sc. Intern

(from Ecole Polytechnique)

About me // Dr. Michaël Marcozzi



- **Permanent researcher** @ CEA LIST, Université Paris-Saclay
- My research group focus on **software analysis for security**
- **Invited lecturer** @ ENSTA, Institut Polytechnique de Paris

Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

Fuzzing [1/2]

Fuzzing a program (for security) is…

1. Feed program with massive number of <u>automatically generated inputs</u>
2. <u>Trigger</u> so observable <u>failures</u> (e.g. crashes)
3. Analyse failures to <u>reveal</u> program <u>vulnerabilities</u> to fix or exploit

buffer-overflow.c

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[10];
    strcpy(password_buffer, password);
    if (strcmp(password_buffer, "dumbledore") == 0)
        auth_flag = 1;
    if (strcmp(password_buffer, "gandalf") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    AFL_INIT_ARGV();
    if (argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    } else if (check_authentication(argv[1]))
    {
        /* Code doing sensitive actions here */
        printf("\nSensitive actions done.\n");
    }
    else printf("\nAccess Denied.\n");
}
```

Fuzzing [1/2]

Fuzzing a program (for security) is…

1. Feed program with massive number of automatically generated inputs
2. Trigger so observable failures (e.g. crashes)
3. Analyse failures to reveal program vulnerabilities to fix or exploit

buffer-overflow.c

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[10];
    strcpy(password_buffer, password);
    if (strcmp(password_buffer, "dumbledore") == 0)
        auth_flag = 1;
    if (strcmp(password_buffer, "gandalf") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    AFL_INIT_ARGV();
    if (argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    } else if (check_authentication(argv[1]))
    {
        /* Code doing sensitive actions here */
        printf("\nSensitive actions done.\n");
    }
    else printf("\nAccess Denied.\n");
}
```

Fuzzing [1/2]

Fuzzing a program (for security) is…

1. Feed program with massive number of automatically generated inputs
2. Trigger so observable failures (e.g. crashes)
3. Analyse failures to reveal program vulnerabilities to fix or exploit

```
./buffer-overflow.exe 11111111111111111111111111111111111111111
Segmentation fault
```

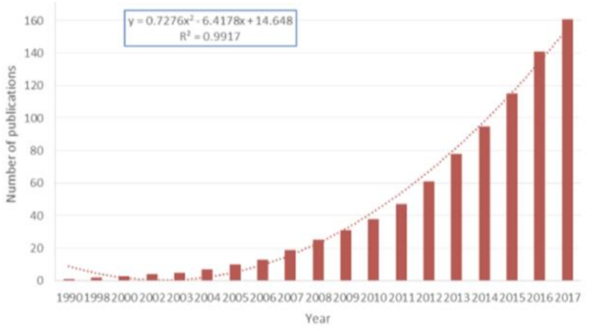
buffer-overflow.c

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[10];
    strcpy(password_buffer, password);
    if (strcmp(password_buffer, "dumbledore") == 0)
        auth_flag = 1;
    if (strcmp(password_buffer, "gandalf") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    AFL_INIT_ARGV();
    if (argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    } else if (check_authentication(argv[1]))
    {
        /* Code doing sensitive actions here */
        printf("\nSensitive actions done.\n");
    }
    else printf("\nAccess Denied.\n");
}
```

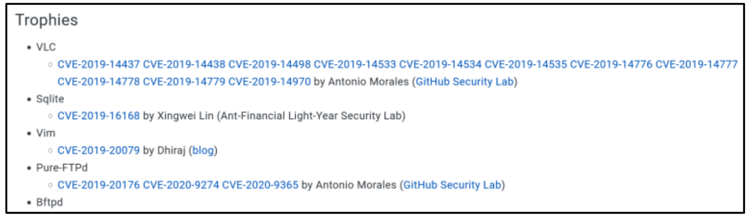
Fuzzing [2/2]

Fuzzing is **popular** (**why?** easy to understand/use, scalable, effective?)…

- Many recent research papers on improving fuzzers
- “At Google, fuzzing has uncovered tens of thousands of bugs” [Metzman et al., 2021]
- Fuzzers have found many CVE vulnerabilities in real programs



Number of fuzzing papers/year [Liang et al., 2018]



Some 2019 CVEs found by AFL++ fuzzer [AFL++ website]

Coverage-based fuzzing [1/3]

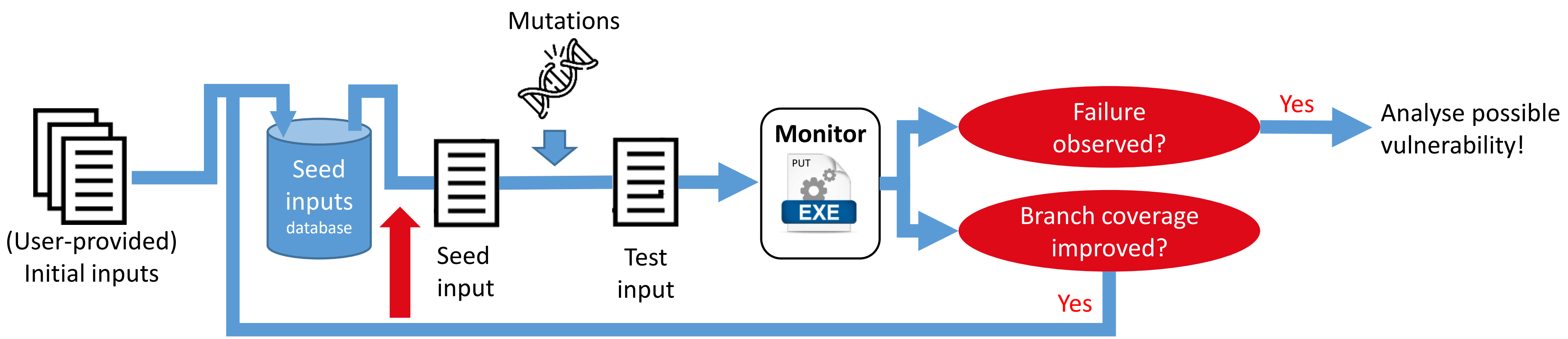
Many fuzzers use **branch coverage to guide** input generation…

- New inputs are generated by mutating the former inputs that improved branch coverage
- The rationale of this heuristic is…
  - The inputs that improved branch coverage uncovered *new interesting program behaviours*
  - Mutating these inputs should *explore these new behaviours even more*

```
if (input > 5) { // Decision point
    // THEN branch
} else {
    // ELSE branch
}
```

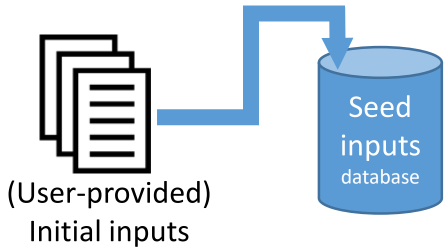
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



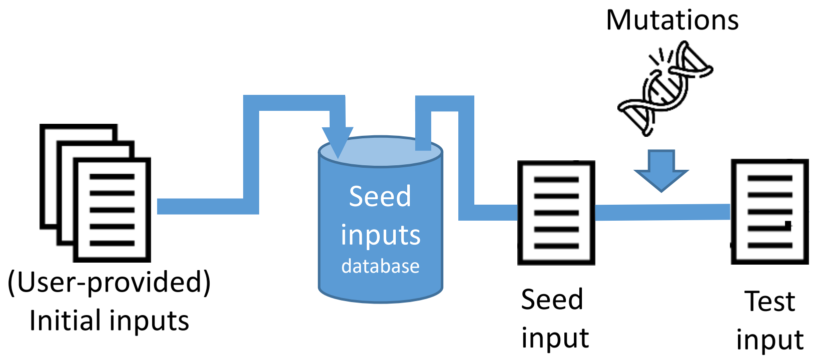
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



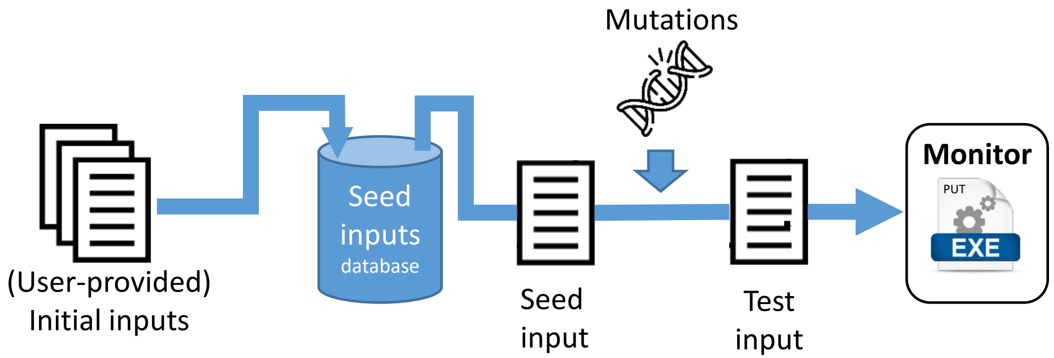
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



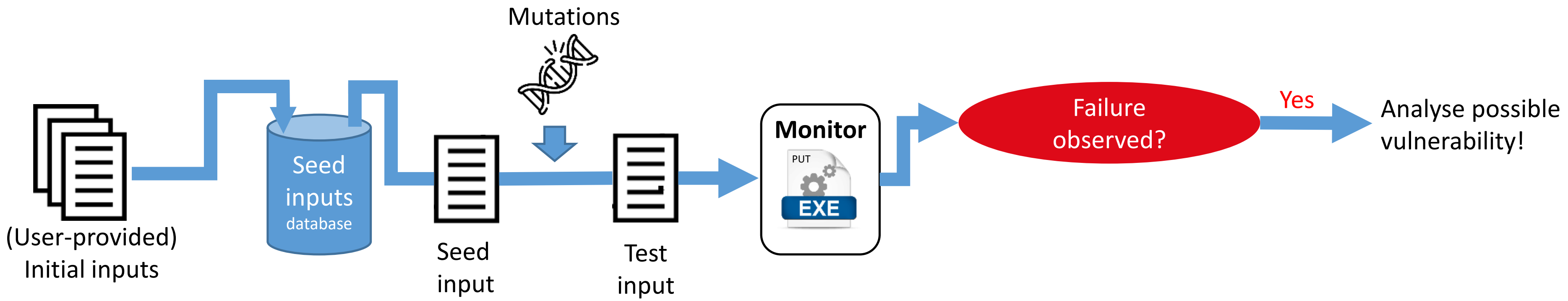
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



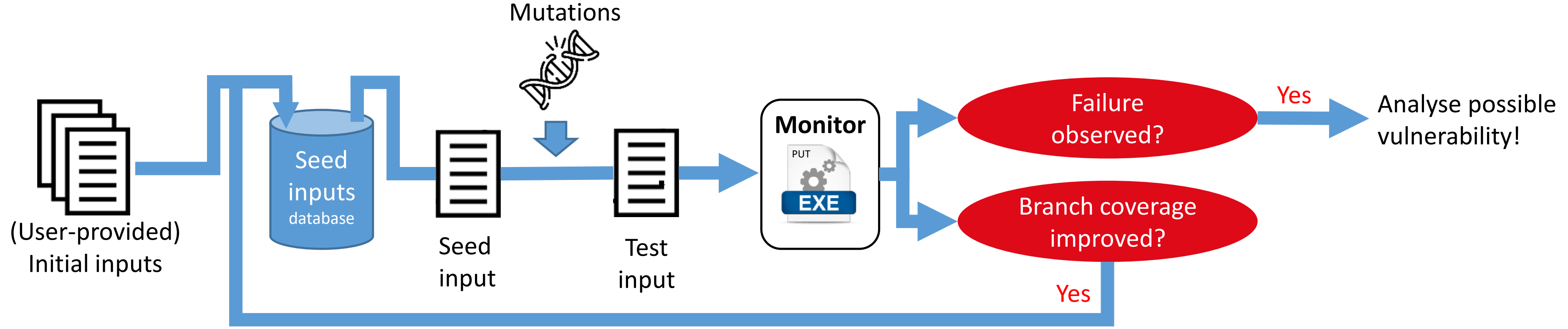
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



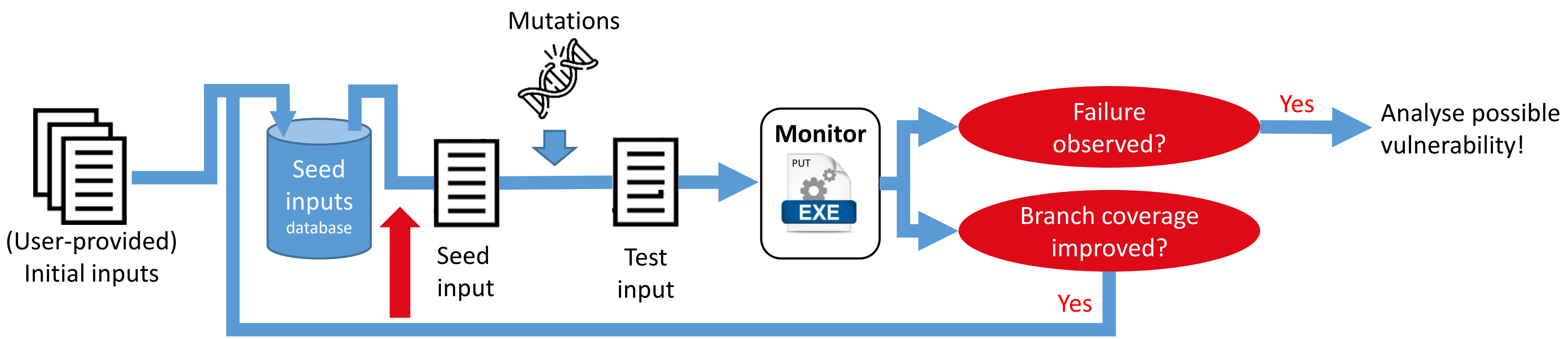
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



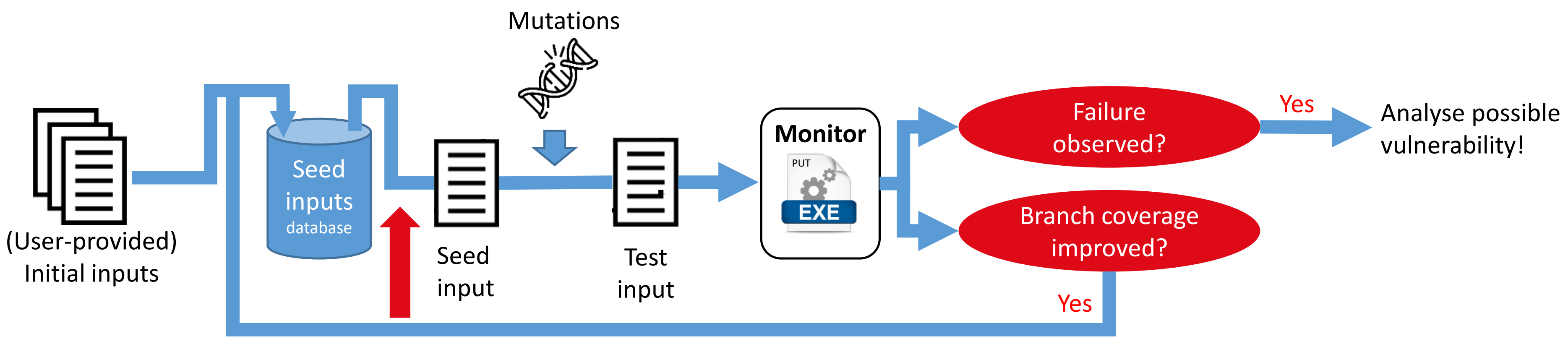
Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



Coverage-based fuzzing [2/3]

More precisely, coverage-based **fuzzers** implement the following **loop**…



The loop terminates when the fuzzing budget is over!

Coverage-based fuzzing [3/3]

Yet, the fuzzing loop alone requires **a high budget to find bugs** in “**difficult**” **branches**…

- A branch in fuzzed code is “difficult” when only activated by tiny fraction of inputs

```
int main (long long input) {
...
if (input == 666) {
    // Difficult branch
}
```

- Code analyses enable fuzzers to be faster at finding inputs entering difficult branches…
    - (Taint tracking) *Track comparisons* between inputs and constants in fuzzed code (e.g. AFL++ fuzzer)
    - (Symbolic execution) *Derive and solve path constraints* to enter barely covered branches (e.g. Qsym fuzzer)

Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

Fine-grained coverage metrics [1/2]

- Branch coverage is a **shallow metric** of interesting program behaviours
- Fuzzers may thus **ignore inputs** that were interesting to find and mutate
- Software testing researchers have for long proposed **finer-grained metrics**
- **Idea**: guide fuzzers using these **control-flow**, **data-flow** or **mutation metrics**

Fine-grained coverage metrics [2/2]

For example, MCC metric considers subtler variations of program logic…

```
if (engine_speed  > 0 || wheels_speed > 0) {
   // Lock door
} else { … }
```

program

Branch Coverage
cover both branches

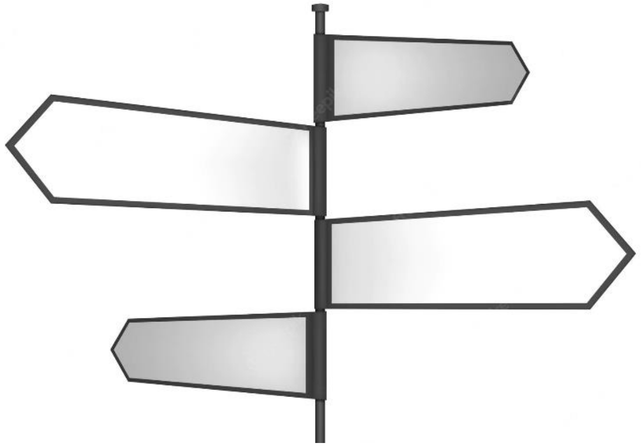
Coverage objective	Satisfying input
Take THEN branch	engine_speed = 5 wheels_speed = 0
Take ELSE branch	engine_speed = 0 wheels_speed = 0

Multiple Condition Coverage (MCC)
cover whole truth table

Coverage objective	Satisfying input
true || true	engine_speed = 5 wheels_speed = 5
true || false	engine_speed = 5 wheels_speed = 0
false || true	engine_speed = 0 wheels_speed = 5
false || false	engine_speed = 0 wheels_speed = 0

State of the art

- **Early research exists** for a **specific** fine-grained metric in a **specific** fuzzer
- Yet, **no clear and general idea of** what practical **impact** is
- **Huge effort needed** to support all fine-grained metrics in all legacy fuzzers



Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

Challenges of guiding fuzzers with finer-grained metrics

1. **Harness the wild variety** of legacy fuzzers and fine-grained metrics…

   Provide a <u>runtime guidance mechanism</u> that works without modifying legacy fuzzers:

   - *Activate* coverage objectives from most fine-grained metrics *for seed selection*
   - *Trigger* search for inputs that satisfy *difficult fine-grained coverage objectives*

2. **Evaluate impact** of fine-grained metrics **over legacy fuzzers performance**

Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

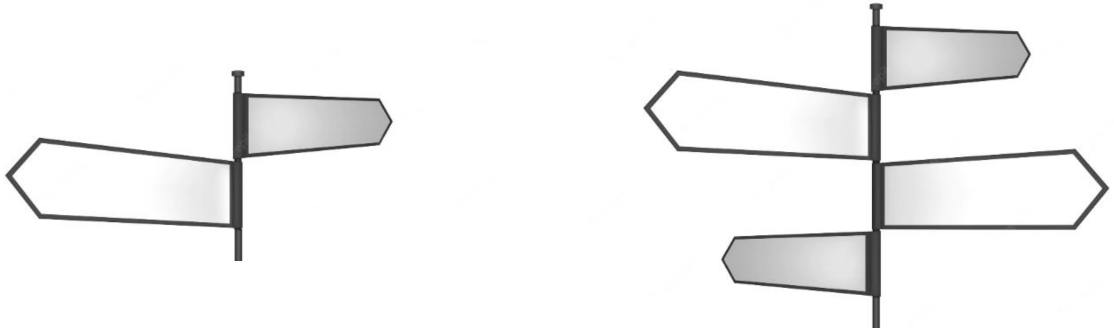
Principle [1/3]

We guide legacy (branch) fuzzers by **transforming the fuzzed program**…

- Objectives from most metrics can be made explicit as assertions in the fuzzed code

  [Bardin et al., 2021]

- Thus, we add a no-op branch (**if** guarded by the assertion predicate) for each assertion



Principle [2/3]

```
if (engine_speed  > 0 || wheels_speed > 0) {
   // Lock door
} else { … }
```

program

+

Multiple Condition Coverage (MCC)

cover whole truth table

Coverage objective	Satisfying input
true || true	engine_speed = 5 wheels_speed = 5
true || false	engine_speed = 5 wheels_speed = 0
false || true	engine_speed = 0 wheels_speed = 5
false || false	engine_speed = 0 wheels_speed = 0

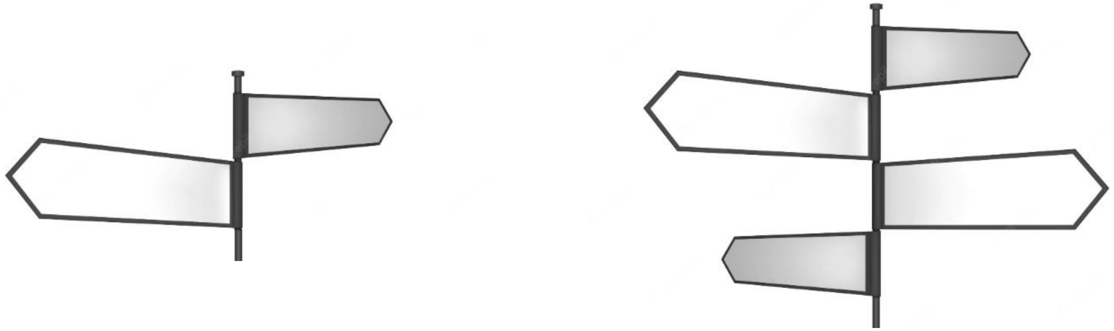
```
if (engine_speed  > 0 && wheels_speed > 0) {}
if (engine_speed  > 0 && wheels_speed <= 0) {}
if (engine_speed  <= 0 && wheels_speed > 0) {}
if (engine_speed  <= 0 && wheels_speed <= 0) {}

if (engine_speed  > 0 || wheels_speed > 0) {
   // Lock door
} else { … }
```

transformed program

Principle [3/3]

When **fuzzing the transformed program** with a legacy (branch) fuzzer…

- …inputs covering the fine-grained objectives will effortlessly <u>be saved as seeds</u>
- …code analyses for difficult branches will help with <u>difficult fine-grained objectives</u>



Practical contributions

We propose a **careful no-op branch insertion tool** for fine-grained metrics…

- …which avoids corrupting the program semantics (side-effects, spurious crashes)
- …which avoids branches being tampered by compiler or fuzzing harness

Simple example of corruption avoidance

```
if (print("a") || graph_ok) {
   // Proceed
} else { /* Error */ }
```

program

Simple example of corruption avoidance

```
if (print("a") || graph_ok) {
   // Proceed
} else { /* Error */ }
```

program

```
if (print("a") && graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && graph_ok) {}
if (!print("a") && !graph_ok) {}

if (print("a") || graph_ok) {
   // Proceed
} else { /* Error */ }
```

transformed program for MCC

Simple example of corruption avoidance

```
if (print("a") || graph_ok) {
  // Proceed
} else { /* Error */ }
```

program

```
if (print("a") && graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && graph_ok) {}
if (!print("a") && !graph_ok) {}

if (print("a") || graph_ok) {
  // Proceed
} else { /* Error */ }
```

transformed program for MCC

Prints "a" 4x more!
(semantic change)

Simple example of corruption avoidance

```
if (print(“a”) || graph_ok) {
   // Proceed
} else { /* Error */ }
```

program

```
if (print(“a”) && graph_ok) {}
if (print(“a”) && !graph_ok) {}
if (!print(“a”) && graph_ok) {}
if (!print(“a”) && !graph_ok) {}

if (print(“a”) || graph_ok) {
   // Proceed
} else { /* Error */ }
```

transformed program for MCC

Prints “a” 4x more!
(semantic change)

```
int temp = print(“a”);
if (temp || graph_ok) {
   // Proceed
} else { /* Error */ }
```

pre-processed program

Simple example of corruption avoidance

```
if (print("a") || graph_ok) {
   // Proceed
} else { /* Error */ }
```

program

```
if (print("a") && graph_ok) {}
if (print("a") && !graph_ok) {}
if (!print("a") && graph_ok) {}
if (!print("a") && !graph_ok) {}

if (print("a") || graph_ok) {
   // Proceed
} else { /* Error */ }
```

transformed program for MCC

Prints "a" 4x more!
(semantic change)

```
int temp = print("a");
if (temp || graph_ok) {
   // Proceed
} else { /* Error */ }
```

pre-processed program

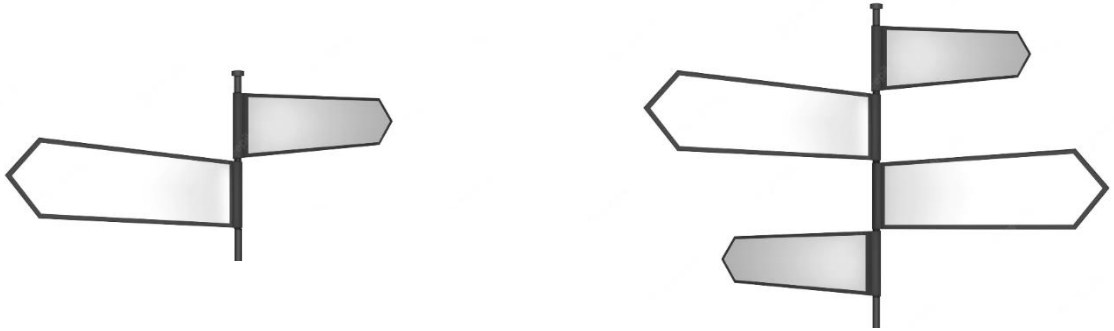
```
int temp = print("a");

if (temp && graph_ok) {}
if (temp && !graph_ok) {}
if (!temp && graph_ok) {}
if (!temp && !graph_ok) {}

if (temp || graph_ok) {
   // Proceed
} else { /* Error */ }
```

transformed pre-processed program for MCC

Possible extensions

No-op branches could be **used as a more general guidance mechanism**...

- They could also be guarded by predicates written by human developers…
- …or by predicates computed by static analysers (like fault preconditions)



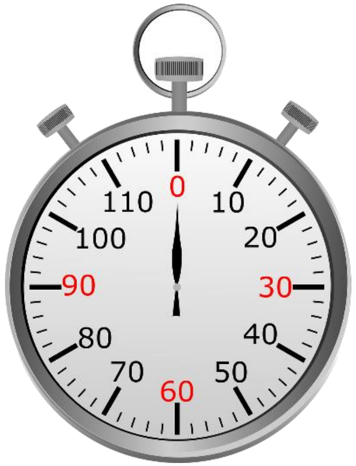
Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

Main evaluation plan

We **evaluate the impact of fine-grained metrics** over fuzzing…

- …by running legacy fuzzers over original programs and transformed versions
- …and comparing throughput, seeds number, covered branches and found bugs

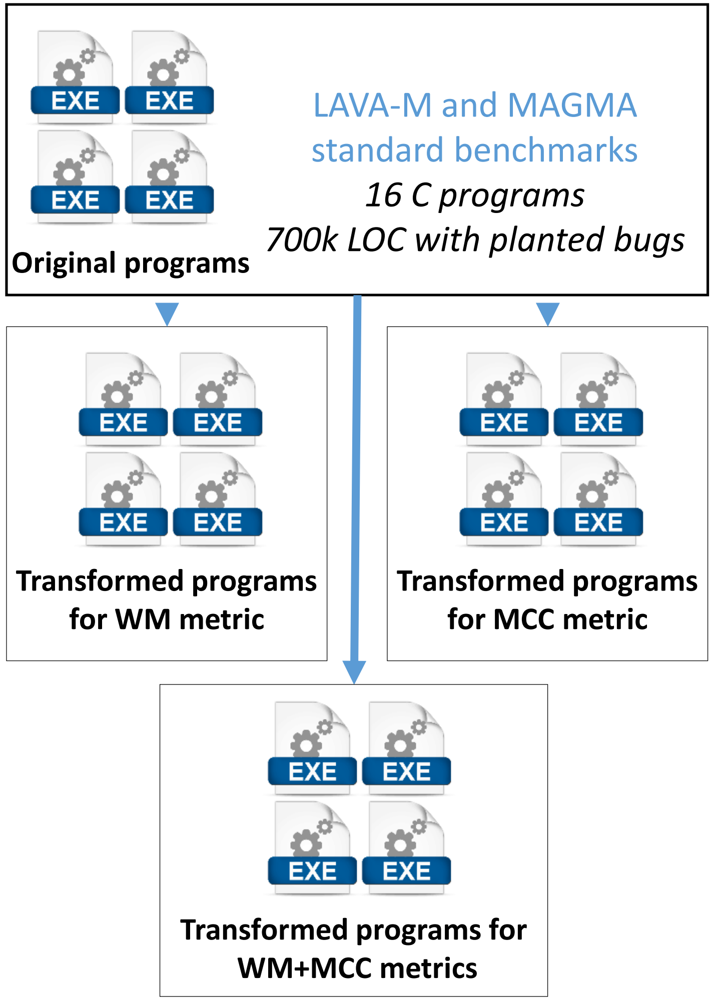


Main experimental setup

Original programs

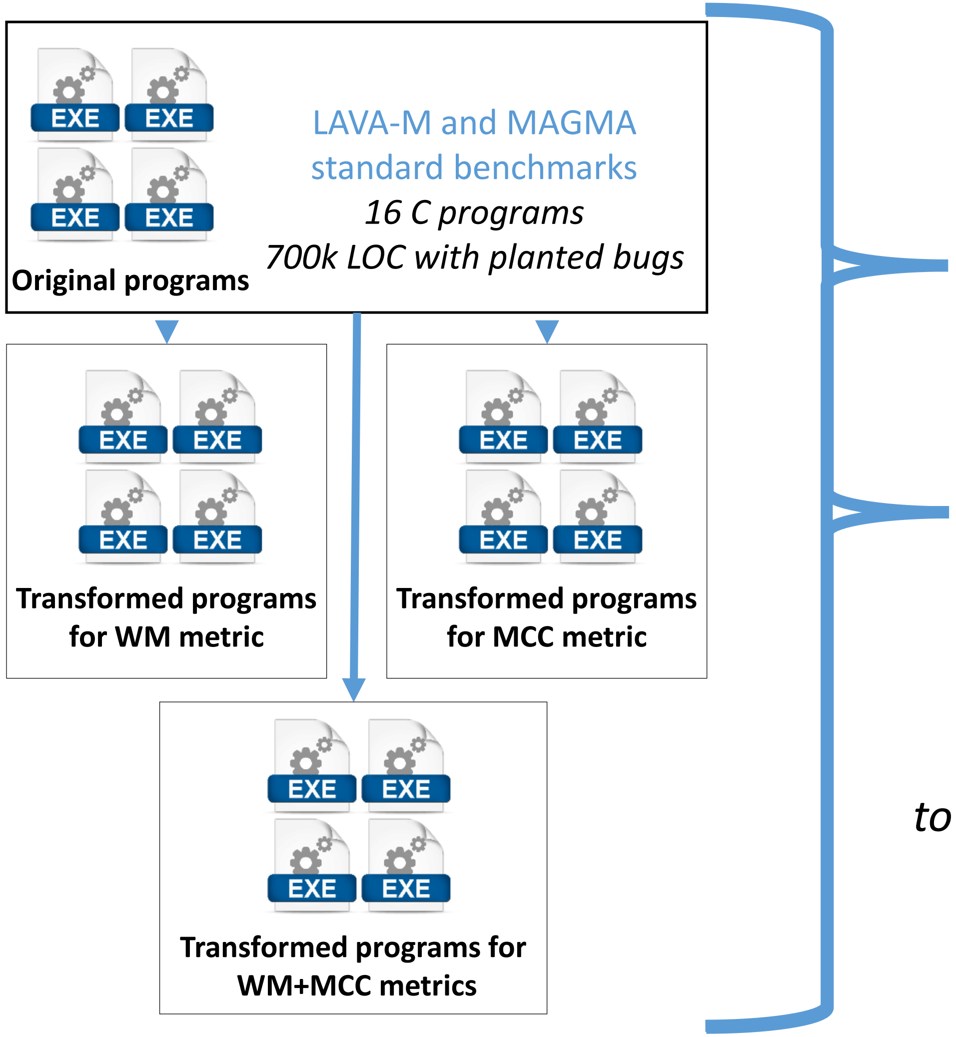
LAVA-M and MAGMA standard benchmarks
16 C programs
700k LOC with planted bugs

Main experimental setup



We use **Multiple Condition Coverage** (MCC) and **Weak Mutations** coverage (WM) two **common** fine-grained metrics, notoriously **denser** than branch coverage

Main experimental setup

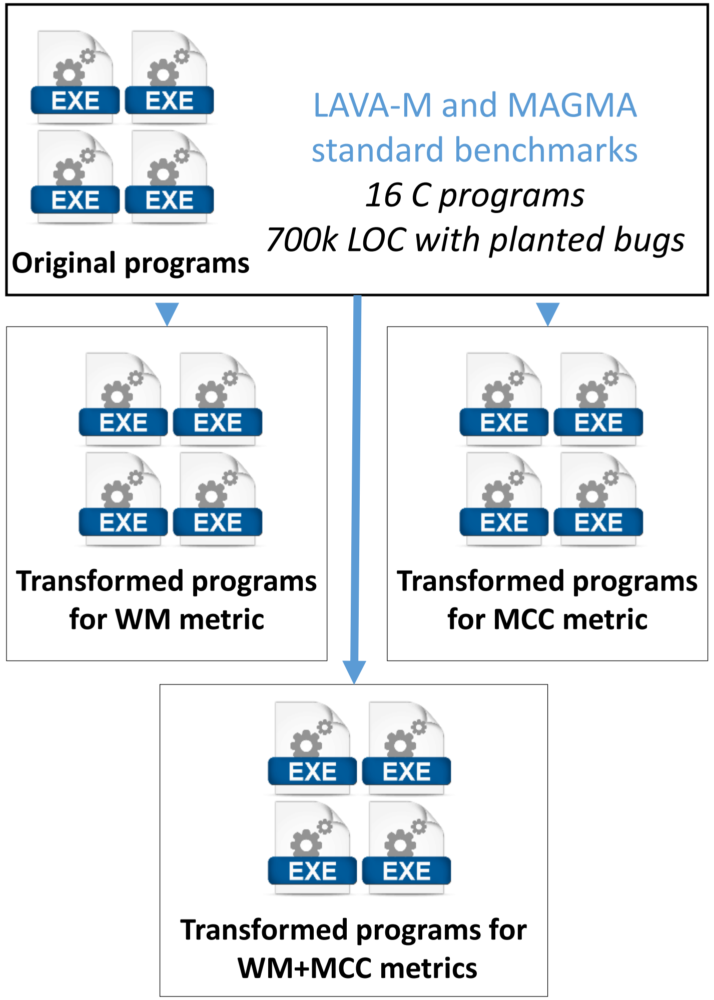


AFL++

qsym

5 x (24h fuzzing campaign)
per program
to improve statistical significance

Main experimental setup



AFL++

qsym

5 x (24h fuzzing campaign)
per program
to improve statistical significance

Averaged…

- fuzzer’s throughput
- # saved seeds
- # covered branches
- # planted bugs that were detected

2.5 years of CPU computation happen here

Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs
base64	+29%	+2%	+2	—	+40%	+1%	—	—	-5%	+2%	+2	—
uniq	+16%	-5%	+7	—	-1%	+12%	+10	—	-21%	+13%	+6	—
md5sum	+18%	-34%	-41	—	+3%	-31%	-41	—	+25%	-24%	-12	—
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	+54%	-1059	-1
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	-19%	-6062	—
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	-5%	-47	-1
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	-1%	-27	—
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	+1%	-13	-1

Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs
[redacted]												
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	+54%	-1059	-1
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	-19%	-6062	—
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	-5%	-47	-1
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	-1%	-27	—
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	+1%	-13	-1

Fuzzer quickly saturates on smaller and simpler programs…

Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs
base64	+29%	+2%	+2	—	+40%	+1%	—	—	-5%	+2%	+2	—
uniq	+16%	-5%	+7	—	-1%	+12%	+10	—	-21%	+13%	+6	—
md5sum	+18%	-34%	-41	—	+3%	-31%	-41	—	+25%	-24%	-12	—
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	+54%	-1059	-1
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	-19%	-6062	—
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	-5%	-47	-1
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	-1%	-27	—
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	+1%	-13	-1

Fine-grained metrics slow down the fuzzer

(instrumented program is slower and produces more coverage data)

Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs
base64	+29%	+2%	+2	—	+40%	+1%	—	—	-5%	+2%	+2	—
uniq	+16%	-5%	+7	—	-1%	+12%	+10	—	-21%	+13%	+6	—
md5sum	+18%	-34%	-41	—	+3%	-31%	-41	—	+25%	-24%	-12	—
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	+54%	-1059	-1
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	-19%	-6062	—
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	-5%	-47	-1
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	-1%	-27	—
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	+1%	-13	-1

Fine-grained metrics improve performance when fuzzer slowdown is low enough and bug density is high enough (favour local exploration vs. new branch discovery)

Consolidated results for AFL++

(detailed results for AFL++ and QSYM are available in the paper, observations are similar)

Executable	AFL++ with MCC				AFL++ with WM				AFL++ with MCC + WM			
	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs	Throughput	Seeds	Branches	Bugs
base64	+29%	+2%	+2	—	+40%	+1%	—	—	-5%	+2%	+2	—
uniq	+16%	-5%	+7	—	-1%	+12%	+10	—	-21%	+13%	+6	—
md5sum	+18%	-34%	-41	—	+3%	-31%	-41	—	+25%	-24%	-12	—
who	-6%	+19%	+133	+165	-9%	+28%	+6	+98	-19%	+22%	-4	-56
lua	-8%	+6%	-65	—	-33%	+7%	-159	—	-36%	+6%	-99	—
exif	-21%	-19%	-41	+1	-12%	-5%	-13	+1	-27%	-25%	-98	+1
sndfile	-48%	+2%	-239	—	-72%	+39%	-578	—	-64%	+48%	-373	—
libpng_read	-7%	+64%	-33	-1	-3%	+45%	-12	—	-13%	+95%	-16	—
tiff_read_rgba	-49%	-2%	-268	-2	-48%	+11%	-354	-1	-45%	+15%	-158	-1
tiffcp	-49%	-9%	-653	-2	-52%	+18%	-512	-2	-44%	+18%	-543	-2
read_memory	-84%	+35%	-1447	—	-63%	+8%	-556	—	-86%	+53%	-1333	—
xmllint	-72%	+46%	-850	-1	-49%	+12%	+401	-1	-77%	[illegible]	[illegible]	[illegible]
sqlite3	-19%	-7%	-2489	—	-25%	-10%	-5297	—	-45%	[illegible]	[illegible]	[illegible]
server	-17%	-3%	+3	-1	-18%	-5%	-26	-1	-33%	[illegible]	[illegible]	[illegible]
client	-17%	+2%	+16	—	-27%	—	-20	—	-42%	[illegible]	[illegible]	[illegible]
x509	-18%	+1%	-9	-1	-21%	+1%	-11	—	-24%	[illegible]	[illegible]	[illegible]

Hard to know if these conditions are met before fuzzing (most of the time, no)… :-(

Fine-grained metrics improve performance when fuzzer slowdown is low enough and bug density is high enough (favour local exploration vs. new branch discovery)

Outline

1. **Context**: coverage-based fuzzing
2. **Problem**: branch coverage is shallow
3. **Goal:** enable and evaluate fuzzer guidance with fine-grained metrics
4. **Proposal**: finer-grained objectives as new branches in fuzzed code
5. **Experimental evaluation of impact**
6. **Conclusions**

Conclusions [1/2]

Adding **no-op branches** to **fuzzed code**…

- Can provide runtime guidance to legacy (branch) fuzzers out of the box
- Can encode guidance from most fine-grained coverage metrics
- Requires careful transformation for not breaking semantics (beware of corner cases)

Future work involves…

- Study tighter integration with fuzzer harness and configuration
- Use to encode human directives or bug preconditions from static analysers

Conclusions [2/2]

Fine-grained metrics should **not replace branch coverage** to guide fuzzers…

- Impact is hard to predict before fuzzing and usually neutral or negative
- Other studies (with tight fuzzer/metric integration) tend to confirm this trend
- Yet, they might be useful in small doses, to improve local exploration where needed

Future work involves…

- Investigate favourable circumstances that could make fine-grained metrics profitable
- Notably, use them only in fragile or sensitive parts of the code…

Fine-Grained Coverage-Based Fuzzing

Key takeaways

> Carefully adding branches to fuzzed code provides guidance to fuzzers

> Fine-grained metrics slow down fuzzers but favour local exploration

Dr. Michaël Marcozzi

Permanent Researcher

@michaelmarcozzi

www.marcozzi.net

www.marcozzi.net

Postdocs, Ph.D. students and interns

Software security and program analysis