RUB

RUHR-UNIVERSITÄT BOCHUM

Automated Program Repair for Security

03.03.2025 – MBZUAI – Abu Dhabi

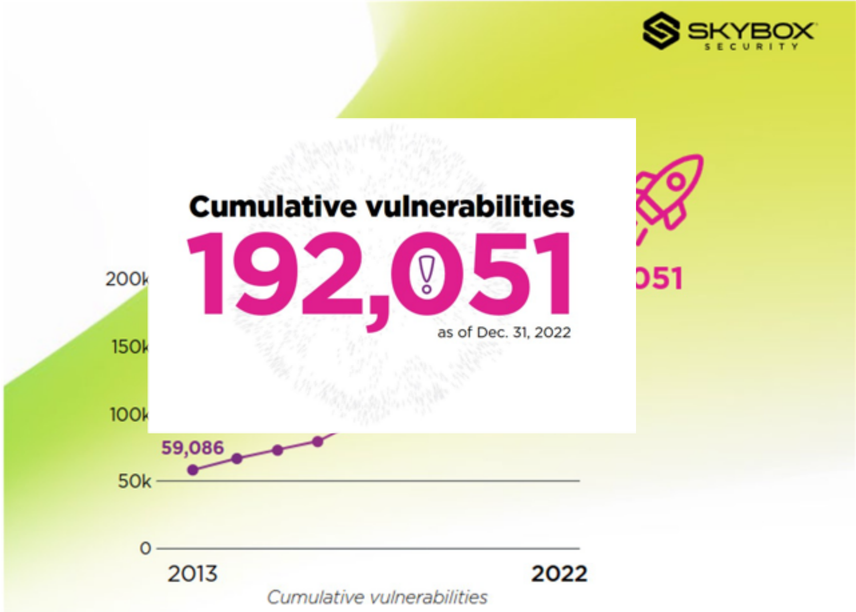
Prof. Dr. Yannic Noller
Software Quality group

Failures because of Software Bugs

https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history

Security Vulnerabilities



- 25k+ vulnerabilities in 2022
- Microsoft reported 1292 vulnerabilities in 2022 [1]
- Vulnerabilities remain unpatched for 55 months in NPM eco-system and 94 months for RubyGems
- 40.86% patched after disclosure in python packages
- Increase of backdoors in 2022 exploiting known vulnerabilities [2]

[1] Vulnerability and threat trends report, Skybox Security 2023
[2] Microsoft VulnerabilitiesReport, BeyondTrust, 2023
[3] X-Force Threat Intelligence Index 2023, IBM Security, February 24, 2023

This Talk: unified processes for software security repair

- **Part 1**: **vulnerabilities** that **can** be detected with sanitizers (e.g., during fuzzing)
  - **sanitizer-driven concolic execution** to compute repair constraint
  - taint/dependency analysis to identify potential **fix locations**
  - search-based inspired **code mutations**

- **Part 2**: **vulnerabilities** that **cannot** be detected with current sanitizers
  - timing **side-channel** vulnerabilities (hyper-property)
  - provide **feedback** to the **software developers** (not just a monitoring solution) to generate awareness for side channel risks arising from code patterns
  - allow **partial fixing** instead of complete elimination to allow a tradeoff between security and performance (pattern-based repair)

Part 0

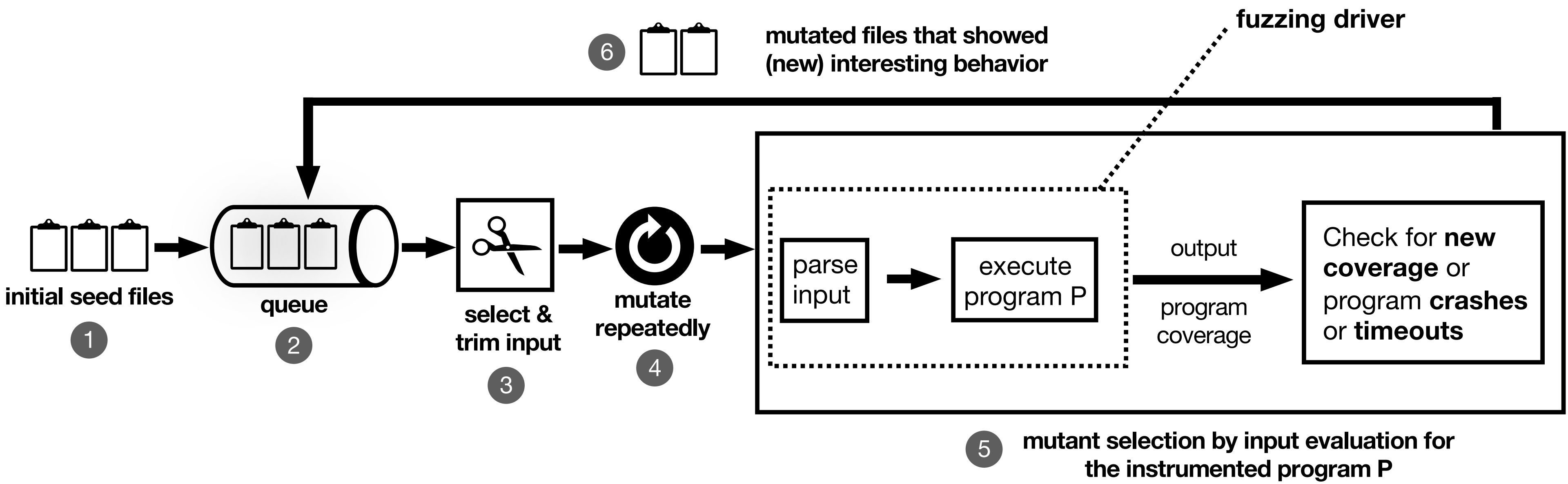
Background

Background – Fuzzing

- term **fuzzing** was coined by Miller et al. in 1990, when they used a random testing tool to investigate the reliability of UNIX tools
- classification based on degree of **program analysis**
  - blackbox / greybox / whitebox fuzzing
- classification based on **generation** technique
  - search-based fuzzing
  - generative fuzzing
- state-of-the-art in vulnerability detection: **coverage-based, mutational fuzzing**

Miller, B. P., Fredriksen, L., & So, B. “An Empirical Study of the Reliability of UNIX Utilities”, Commun. ACM 1990.

Greybox Fuzzing



RUHR
UNIVERSITÄT
BOCHUM

Background – Sanitizer

- **Key idea**: **instrument** the program **to make security issues *visible/observable***
- A sanitizer detects memory corruption, undefined behavior, and security vulnerabilities during runtime, which makes them useful for fuzzing.
- **Common Sanitizers:**
  - AddressSanitizer (ASan): Detects memory errors like buffer under/overflows, use-after-free
  - UndefinedBehaviorSanitizer (UBSan): Flags undefined behavior (e.g., signed integer overflows, use of uninitialized memory).
  - MemorySanitizer (MSan): Identifies use of uninitialized memory.
  - ThreadSanitizer (TSan): Detects data races and thread synchronization issues.
  - LeakSanitizer (LSan): Reports memory leaks.

Symbolic Execution

- introduced by King[1] and Clarke[2]
- analysis of programs with **unspecified inputs**, i.e. execute a program with **symbolic** inputs
- **symbolic states** represent sets of concrete states
- for each path, build a **path condition**
  - condition on inputs – for the execution to follow that path
  - check path condition satisfiability – explore only feasible paths
- symbolic state
  - symbolic values / expressions for variables
  - path condition
  - instruction pointer

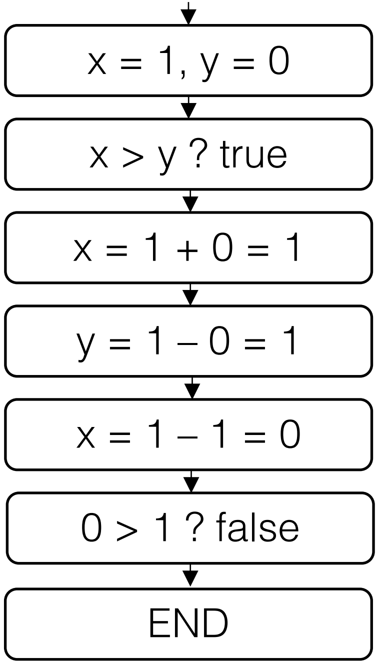
[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385-394.
[2] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," in IEEE Transactions on Software Engineering, vol. SE-2, no. 3, pp. 215-222, Sept. 1976.

Example: concrete execution

code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

concrete execution path

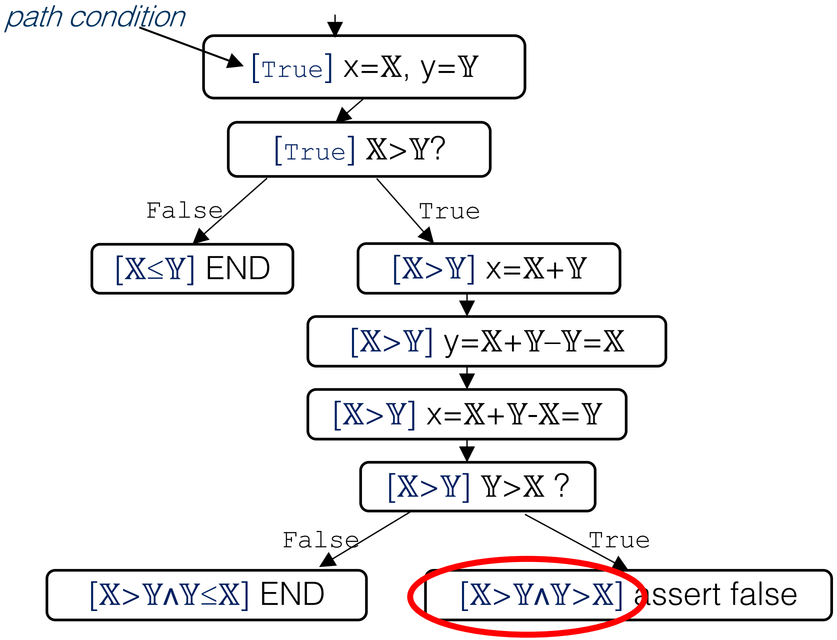


Example: symbolic execution

code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

symbolic execution tree



unsatisfiable !!!

Hint: solve PCs to obtain test inputs

Example: concolic execution

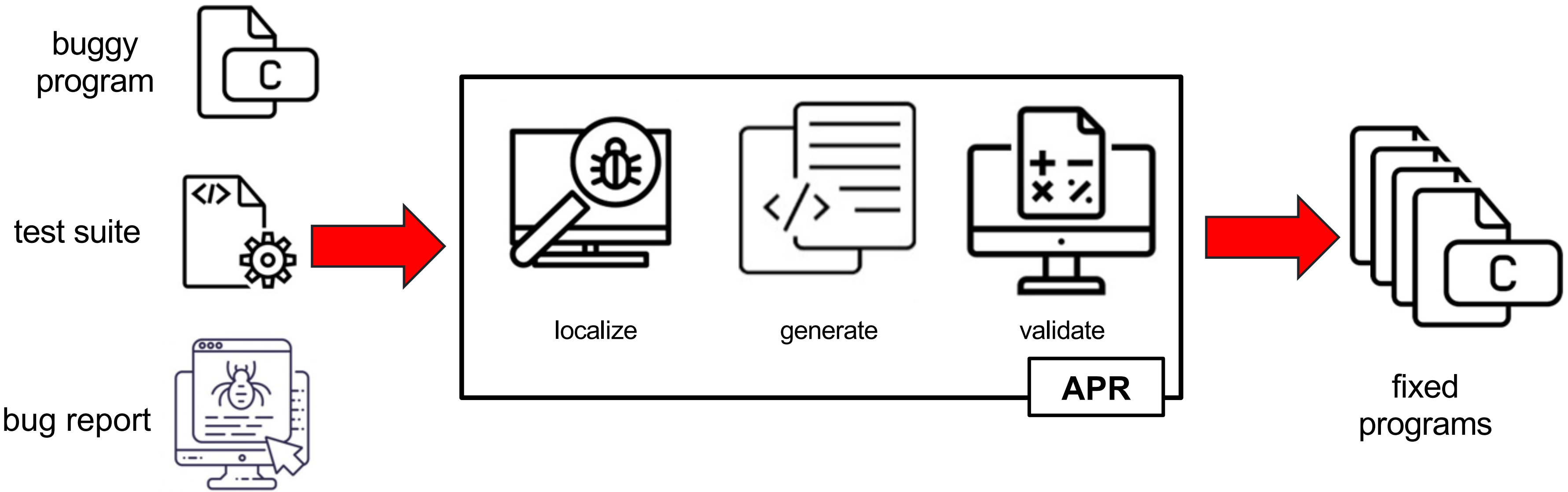
follow the concrete execution path while still collecting the information about the symbolic state

code that swaps 2 integers

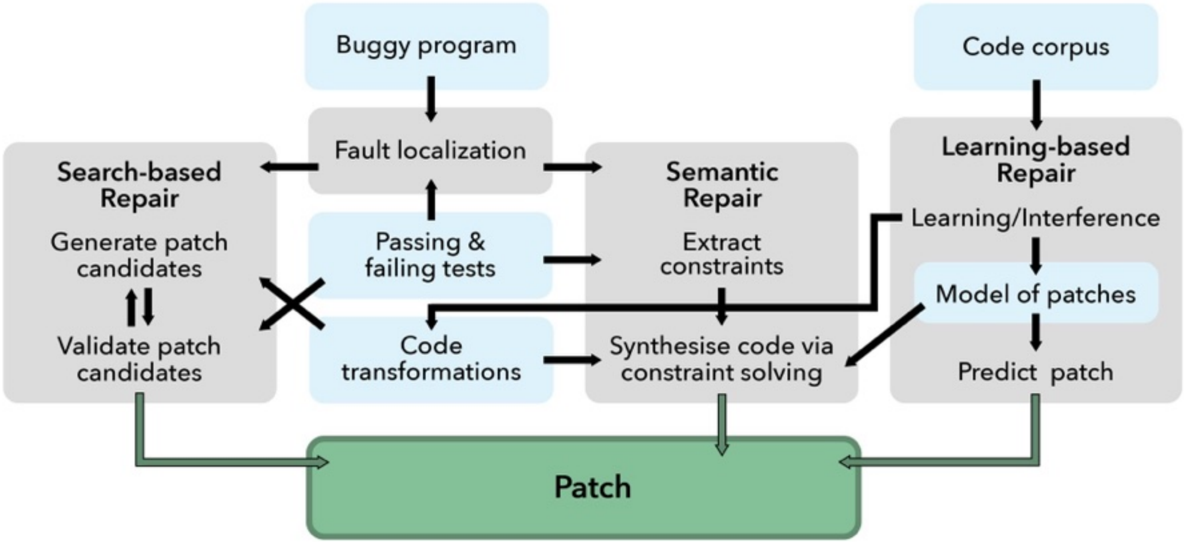
```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

concrete execution path	symbolic information
x = 1, y = 0	[True] $x = \mathbb{X}$, $y = \mathbb{Y}$
x > y ? true	$[\mathbb{X}>\mathbb{Y}]$ $x = \mathbb{X}$, $y = \mathbb{Y}$
x = 1 + 0 = 1	$[\mathbb{X}>\mathbb{Y}]$ $x = \mathbb{X}+\mathbb{Y}$, $y = \mathbb{Y}$
y = 1 – 0 = 1	$[\mathbb{X}>\mathbb{Y}]$ $x = \mathbb{X}+\mathbb{Y}$, $y=\mathbb{X}$
x = 1 – 1 = 0	$[\mathbb{X}>\mathbb{Y}]$ $x=\mathbb{Y}$, $y=\mathbb{X}$
0 > 1 ? false	$[\mathbb{X}>\mathbb{Y} \land \mathbb{Y}\leq\mathbb{X}]$ $x=\mathbb{Y}$, $y=\mathbb{X}$
END	$[\mathbb{X}>\mathbb{Y} \land \mathbb{Y}\leq\mathbb{X}]$ $x=\mathbb{Y}$, $y=\mathbb{X}$

Automated Program Repair (APR)



APR Approaches



State-of-the-art in Program Repair: Pictorial view derived from Communications of the ACM article 2019.

https://nus-apr.github.io/

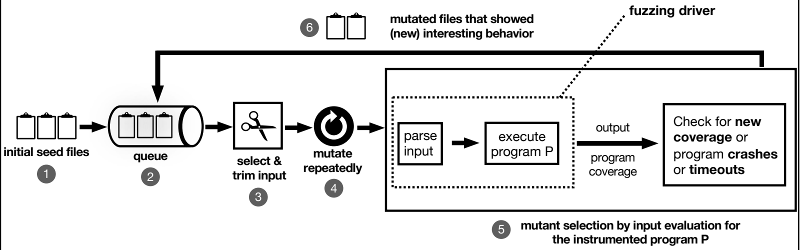
Security Vulnerability Repair

t_F → only one failing test-case available / test-oracle is crash-freedom → weak specification / many plausible patches

Examples of over-fitting patches:

```
if (((1 > 0) && (1 > 0))) exit(1);
if (((! (image->res_unit == 3)) && (! (image->res_unit == 3)))) return;
if ( (! (((! ((- 4) == 0))) && ((! (0 == 0)) || (! (64 == 0)))))) break;
if ( (! ((log_level && (! ((- 4) == 0))) && log_level))) exit(0);
```

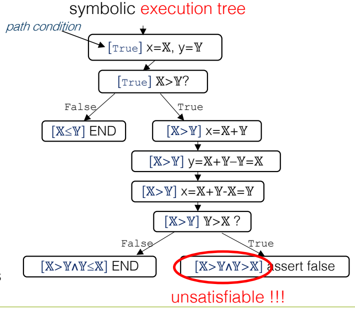
Greybox Fuzzing



Example: symbolic execution

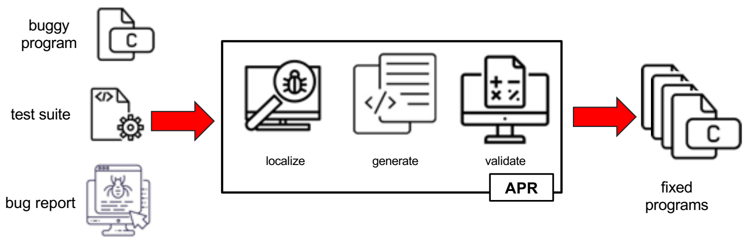
code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```



Hint: solve PCs to obtain test inputs

Automated Program Repair (APR)



Security Vulnerability Repair

Examples of over-fitting patches:

```
if (((1 > 0) && (1 > 0))) exit(1);
if (((! (image->res_unit == 3)) && (! (image->res_unit == 3)))) return;
if ( (! (((! ((- 4) == 0))) && ((! (0 == 0)) || (! (64 == 0)))))) break;
if ( (! ((log_level && (! ((- 4) == 0))) && log_level))) exit(0);
```

Part 1

Security Vulnerability Repair via Concolic Execution and Code Mutations

Vulnerability Repair via Concolic Execution and Code Mutations

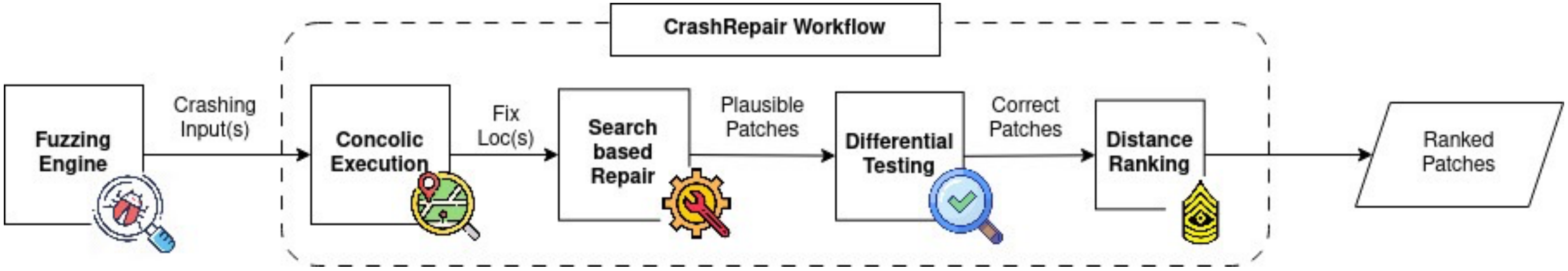
RIDWAN SHARIFFDEEN, National University of Singapore, Singapore
CHRISTOPHER S. TIMPERLEY, Carnegie Mellon University, USA
YANNIC NOLLER, Ruhr University Bochum, Germany
CLAIRE LE GOUES, Carnegie Mellon University, USA
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Security vulnerabilities detected via techniques like greybox fuzzing are often fixed with a significant time lag. This increases the exposure of the software to vulnerabilities. Automated fixing of vulnerabilities where a tool can generate fix suggestions is thus of value. In this work, we present such a tool, called CrashRepair, to automatically generate fix suggestions using concolic execution, specification inference, and search techniques. Our approach avoids generating fix suggestions merely at the crash location because such fixes often disable the manifestation of the error instead of fixing the error. Instead, based on sanitizer-guided concolic execution, we infer desired constraints at specific program locations and then opportunistically search for code mutations that help respect those constraints. Our technique only requires a single detected vulnerability or exploit as input; it does not require any user-provided properties. Evaluation results on a wide variety of CVEs in the VulnLoc benchmark, show CrashRepair achieves greater efficacy than state-of-the-art vulnerability repair tools like Senx. The repairs suggested come in the form of a ranked set of patches at different locations, and we show that on most occasions, the desired fix is among the top-3 fixes reported by CrashRepair.

- using sanitizer-guided **concolic execution**, **specification inference**, and **search techniques**
- avoids just disabling the error manifestation
- no user-provided property needed
- tool: CrashRepair

R. Shariffdeen, C. S. Timperley, Y. Noller, C. Le Goues, and A. Roychoudhury. 2024. Vulnerability Repair via Concolic Execution and Code Mutations. ACM Trans. Softw. Eng. Methodol. https://doi.org/10.1145/3707454

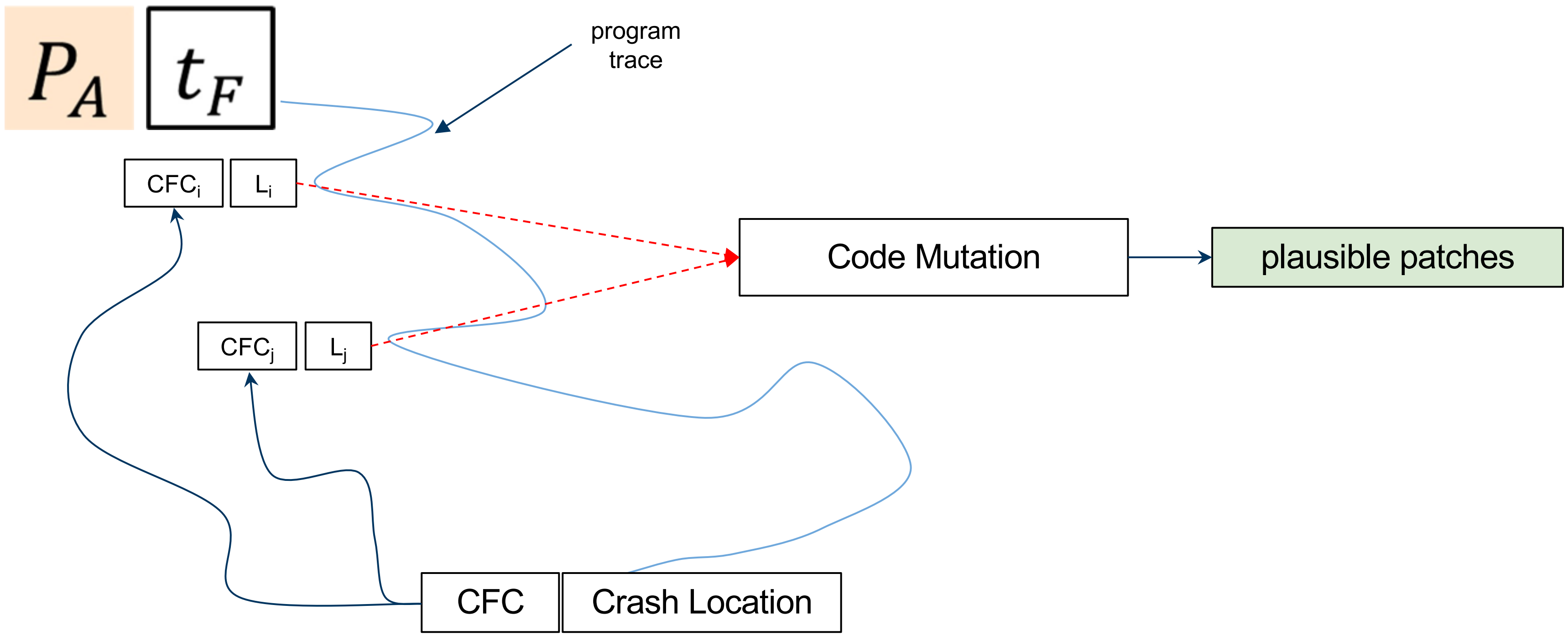
CrashRepair: Workflow



- **Fix Localization** and **Specification Inference** using **Semantic** Analysis
  - generate a crash-free constraint for the program
  - identify fix locations using program dependencies
- Constraint guided **Code Mutations**
  - finds correct error-handling procedures
  - constraint guided mutators to efficiently navigate the search space

RUHR UNIVERSITÄT BOCHUM RUB

CrashRepair: Key Idea



CVE-2016-10092
detected by using AFL

Illustrative Example

P_A — LibTiff v4.0.7

t_F — TIFF Image

```
FILE: libtiff/tif_unix.c:340

void
_TIFFmemcpy(void* d, const void* s, tmsize_t c)
{
        memcpy(d, s, (size_t) c);
}
```

- heap-based buffer overflow
- allows remote attackers to have unspecified impact via a crafted image

```
=================================================================
==173185==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6210000000ff at
pc 0x0000004d9dcc bp 0x7fff071360f0 sp 0x7fff071358a0
WRITE of size 1 at 0x6210000000ff thread T0
         #0 0x4d9dcb in __asan_memcpy /tmp/llvm/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cc:23
         #1 0x5e8984 in _TIFFmemcpy /data/vulnloc/libtiff/CVE-2016-10092/src/libtiff/tif_unix.c:340:2
         #2 0x5eacd0 in DumpModeDecode /data/vulnloc/libtiff/CVE-2016-10092/src/libtiff/tif_dumpmode.c:103:3
         #3 0x5ce351 in TIFFReadEncodedStrip /data/vulnloc/libtiff/CVE-2016-10092/src/libtiff/tif_read.c:2639:6
         #4 0x532b10 in readContigStripsIntoBuffer /data/vulnloc/libtiff/CVE-2016-10092/src/tools/tiffcrop.c:8408:30
         #5 0x5203e5 in loadImage /data/vulnloc/libtiff/CVE-2016-10092/src/tools/tiffcrop.c:10756:13
         #6 0x51a85f in main /data/vulnloc/libtiff/CVE-2016-10092/src/tools/tiffcrop.c:7064:11
         #7 0x7f70cbb90c86 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21c86)
         #8 0x41b1d9 in _start (/data/vulnloc/libtiff/CVE-2016-10092/src/tools/tiffcrop+0x41b1d9)
```

Illustrative Example

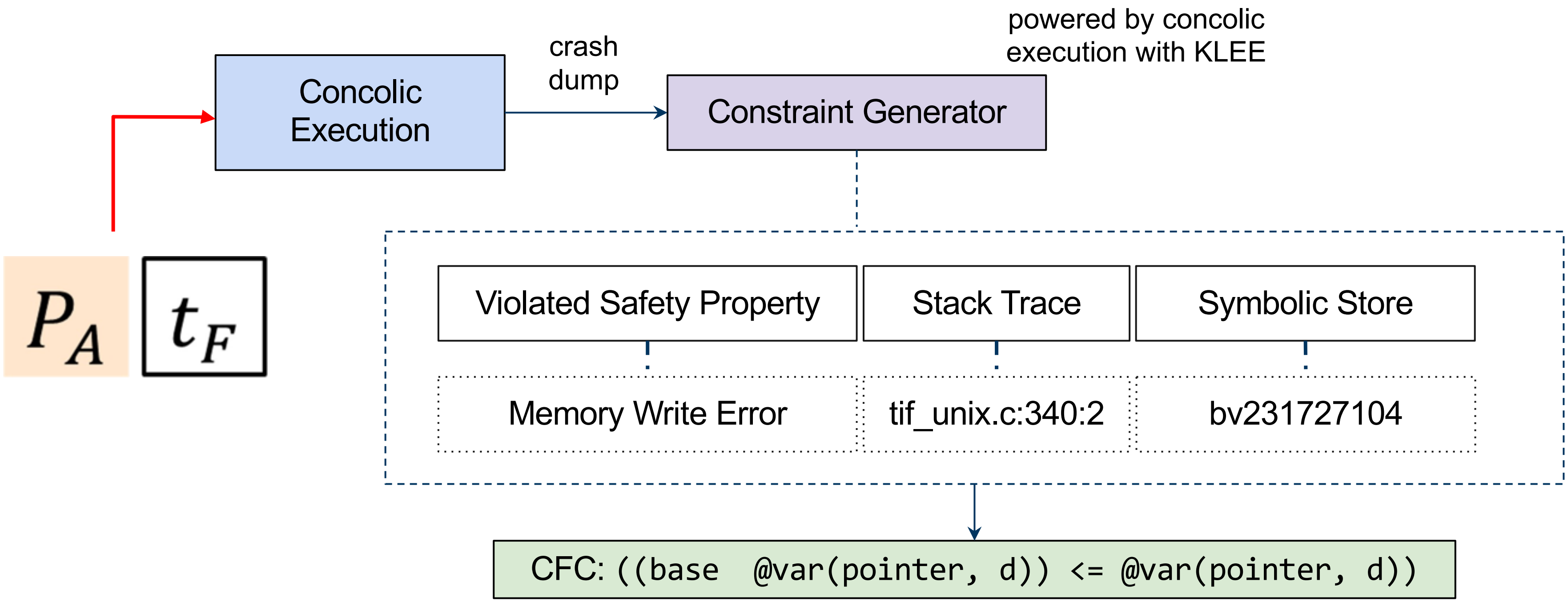
CVE-2016-10092

```
1  static int readContigStripsIntoBuffer(TIFF* in, uint8* buf) {
2    uint8* bufp = buf;
3    int32  bytes_read = 0;
4    uint32 stripsize = TIFFStripSize(in);
5
6    for(strip = 0; strip < nstrips; strip++) {
7      bytes_read = TIFFReadEncodedStrip(in, strip, bufp, -1);
8      rows = bytes_read / scanline_size;
9      if ((strip < (nstrips - 1)) &&  (bytes_read != (int32)stripsize))
10       TIFFError(...);
11
12 - bufp += bytes_read;
13 + bufp += stripsize;
14
15   }
16   return 1;
17 } /* end readContigStripsIntoBuffer */
```

in the failing test case:

- the bytes_read gets assigned to a **negative** number, which later, in line 12,
- causes a **buffer overflow** triggered in a **different** program location
- when **accessing** the pointer **bufp**

Specification Inference



Specification Inference

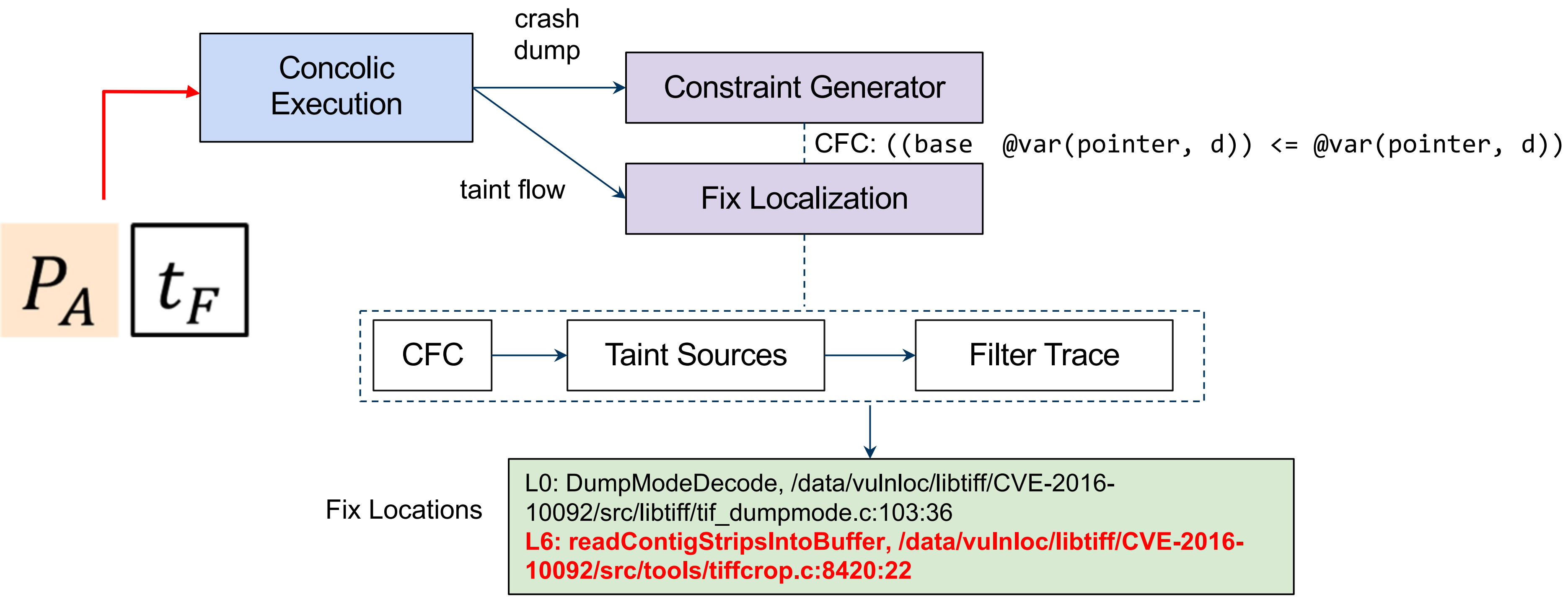
CFC = Crash Free Constraint

CFC: `((base  @var(pointer, d)) <= @var(pointer, d))`

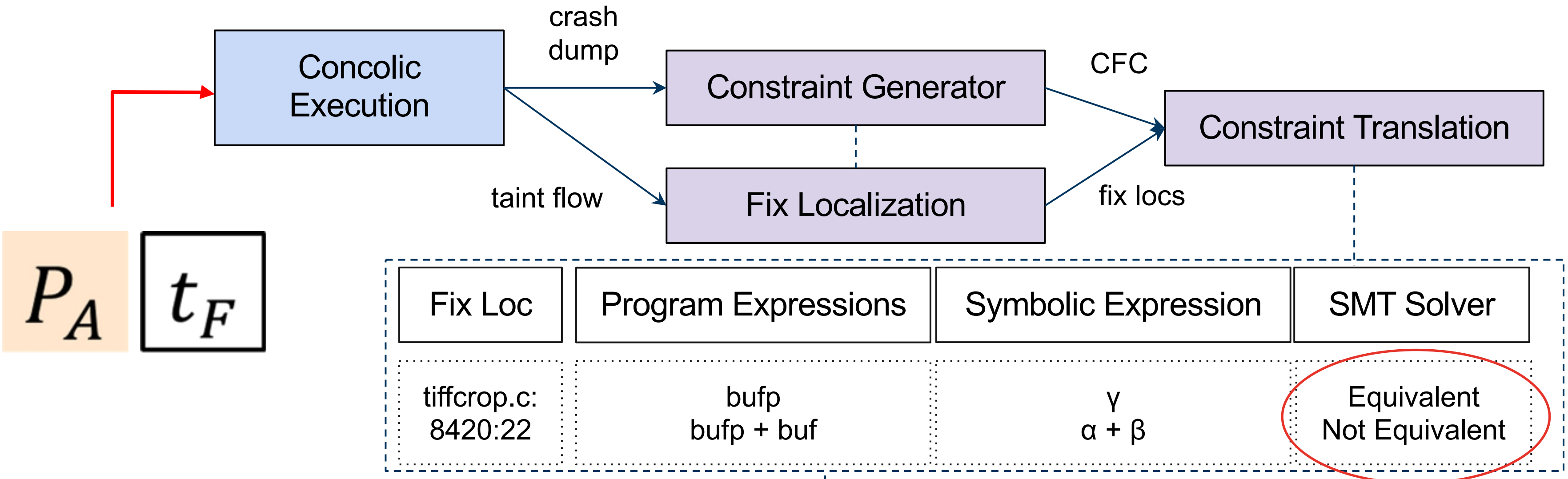
- a **security property** capturing a memory safety property for the pointer variable d
- the memory address accessed by the pointer should be **within the bounds** of the memory allocation
- in this case, the **violation is on the lower bound**, which is the base address of the memory region
- variable **d** is a pointer used by the crashing function _TIFFmemcpy located in the source file libtiff/tif_unix.c
- **@var(pointer, d)** = the current address captured by the pointer d
- **(base @var(pointer, d))** = base address for the pointer captured by the program
  - base address = starting address for allocated memory region accessed with d

RUHR UNIVERSITÄT BOCHUM

Fix Localization



Constraint Translation



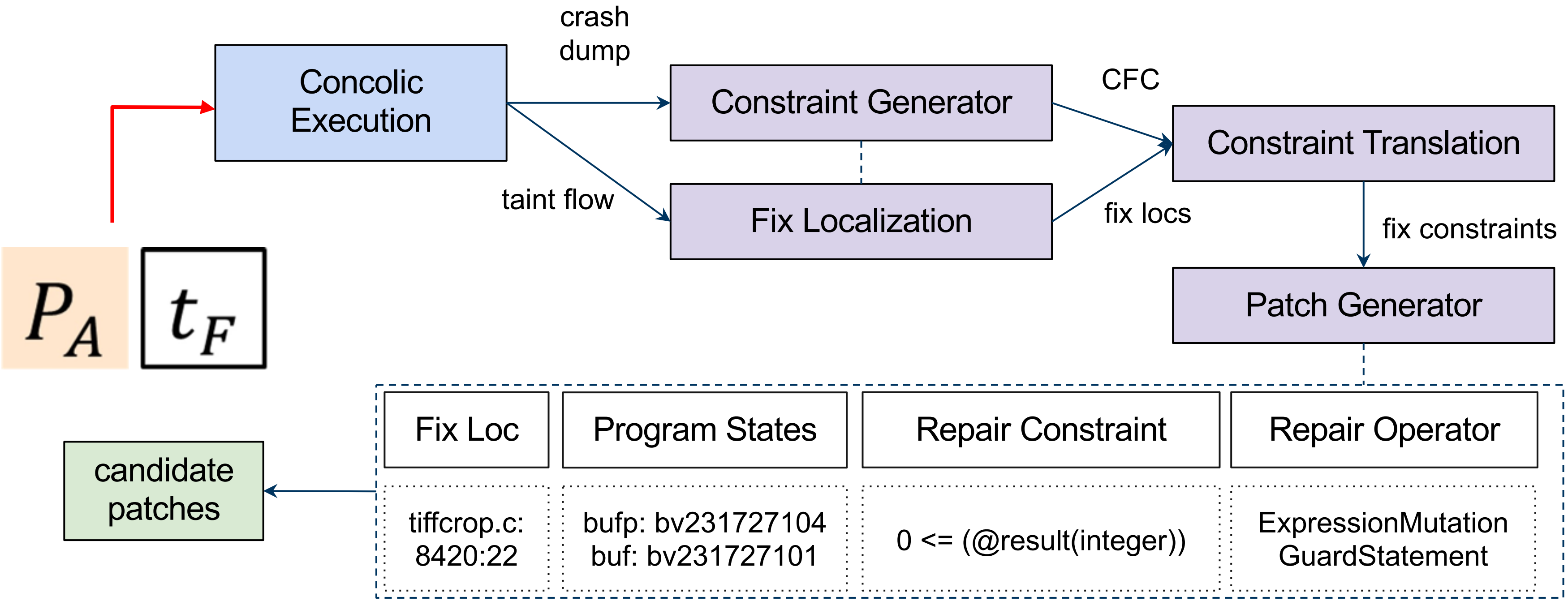
Translated Constraints

L0: (@var(pointer, crepair_base(buf)) <= @var(pointer, buf))
L6: (@var(pointer, bufp) <= (@var(pointer, bufp) + (@result(integer))))

→ L6: 0 <= (@result(integer))

Can we map the expressions in the CFC to local variables in the fix location?

Code Mutation



RUHR UNIVERSITÄT BOCHUM RUB

Illustrative Example

CVE-2016-10092

P_A — LibTiff v4.0.7

t_F — TIFF Image

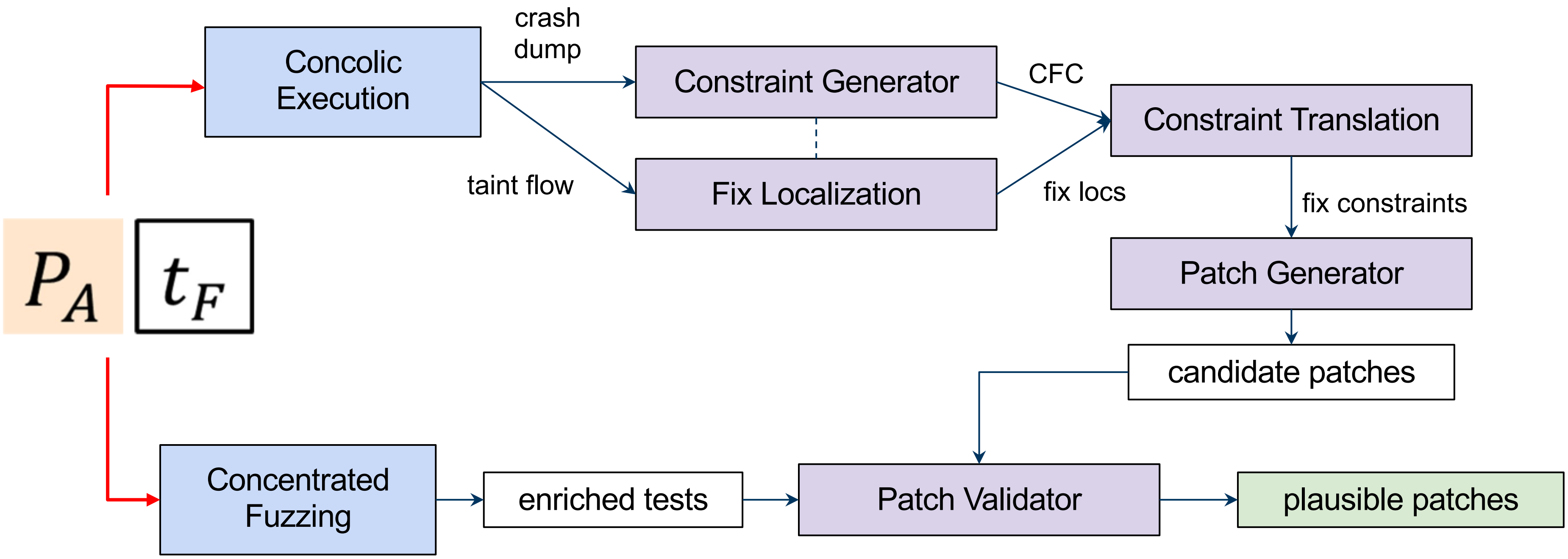
Plausible Patches

```
1: tiffcrop.c:8405: if (!((buf <= bufp))) { return -1; }
2: tiffcrop.c:8405: if (!((buf <= bufp))) { return -1.0; }
3: tiffcrop.c:8405: if (!((buf <= bufp))) { return 1; }
4: tiffcrop.c:8405: if (!((buf <= bufp))) { break; }
5: if ((buf <= bufp)) { bytes_read = TIFFReadEncodedStrip (in,
strip, bufp, -1); }
6: tiffcrop.c:8417: bufp += *bufp;
7: tiffcrop.c:8417: bufp += stripsize;
```

Identical developer patch is ranked in **top-10**

Same patch also fixes CVE-2016-10272 which is another buffer overflow

CrashRepair: Overview



RUHR UNIVERSITÄT BOCHUM RUB

Comparison with SOTA

Tool	# Plausible	# Correct
CrashRepair	29	19
SenX	12	3
ExtractFix	12	5
VulnFix	17	9
CPR	35	9

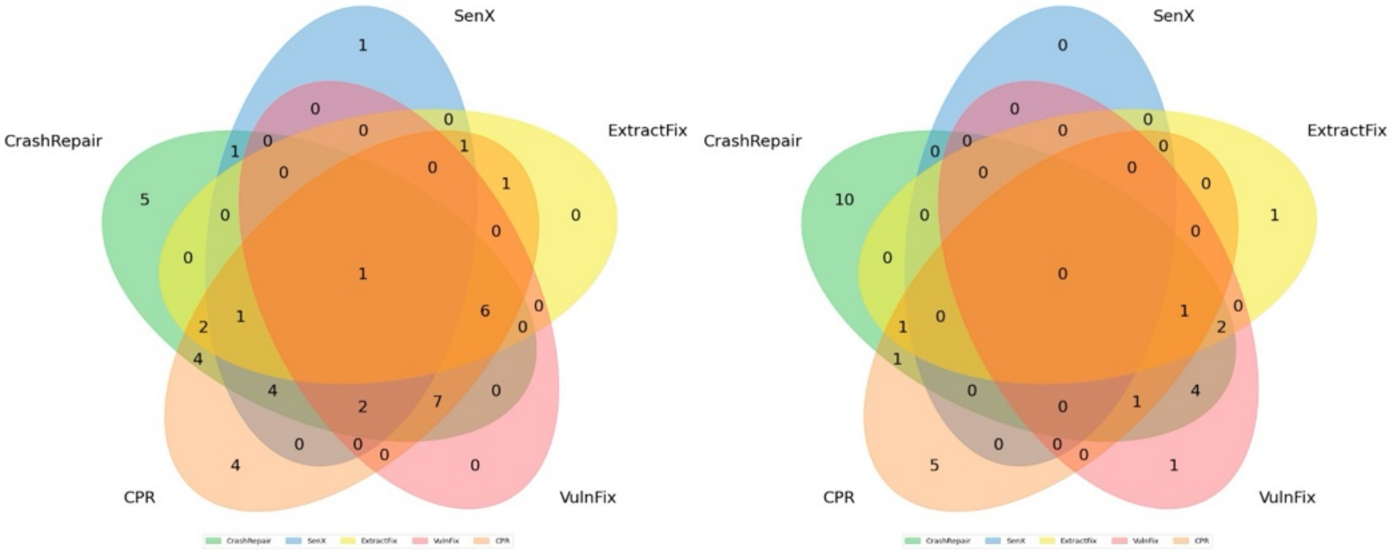
evaluated on 41 subjects in VulnLoc benchmark with 1hr timeout

CrashRepair generates a **plausible** patch for **29 instances** without additional information

CrashRepair generates more plausible patches than SenX, ExtractFix and VulnFix

CrashRepair is **more effective** than existing state-of-the-art for vulnerability repair

Comparison with SOTA



(a) Plausible Patches

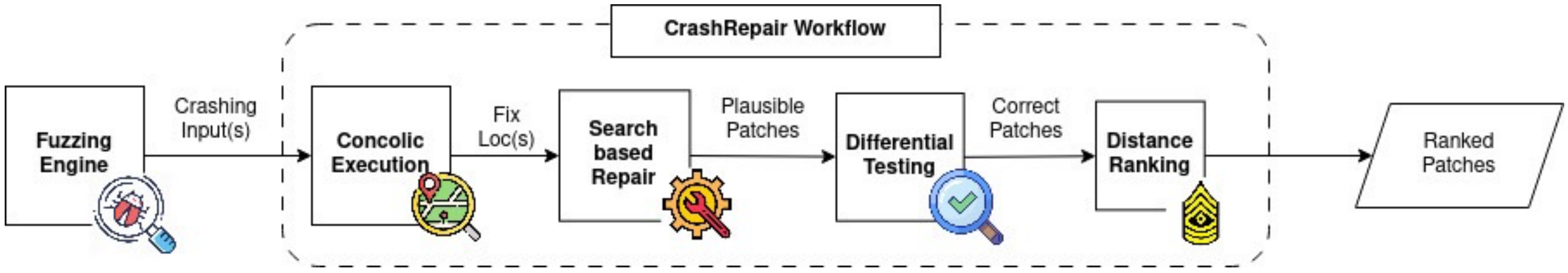
(b) Correct Patches

Limitations

- Limitations in KLEE
  - Does not support floating points, longjmps etc
  - Limitations in detecting memory overflows (i.e. environment modeling)
- Does not handle inputs which leads to large symbolic constraints which will timeout the concolic execution
- Fix-ingredients are derived from observed program expressions

Summary: CrashRepair

- **Combined** semantic analysis with code mutation to find high-quality patches for security vulnerabilities
- **Program dependency** based fix localization can effectively identify fix locations closer to the developer fix location
- **Constraint-guided** search finds high-quality patches compared to existing state-of-the-art techniques



Part 2

Detection, Quantification, Repair of Side-Channel Vulnerabilities

Potential Side-Channel Leakages

By David B. Gleason from Chicago, IL - The Pentagon, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=4891272

Side-Channel Analysis

- ❑ **leakage** of **secret data**
- ❑ **software** side-channels
- ❑ **observables**:
  - execution time
  - memory consumption
  - response size
  - network traffic
  - …

```
boolean pwcheck_unsafe (byte[] pub, byte[] sec) {
    if (pub.length != sec.length) {
        return false;
    }
    for (int i = 0; i < pub.length; i++) {
        if (pub[i] != sec[i]) {
            return false;
        }
    }
    return true;
}
```

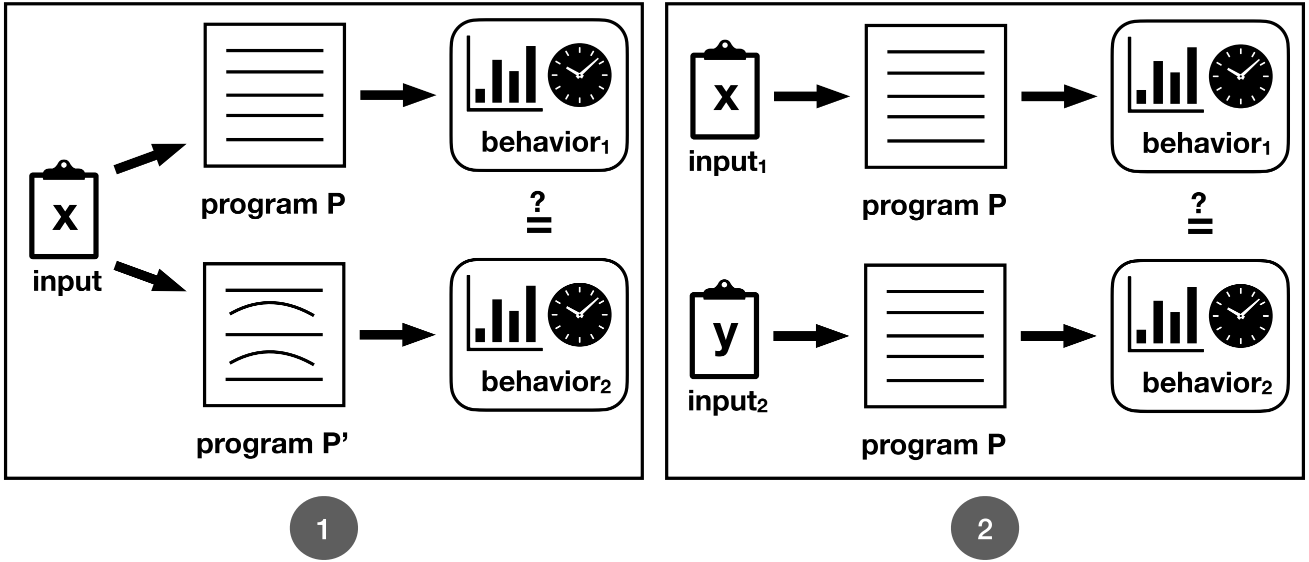
conditional early return causes leakage

Where do we find them?

- ❑ application code, e.g., *Apache Tomcat, FtpServer, …*
- ❑ security libraries, e.g., *JDK, spring security, Bouncy Castle, …*

Differential Software Testing

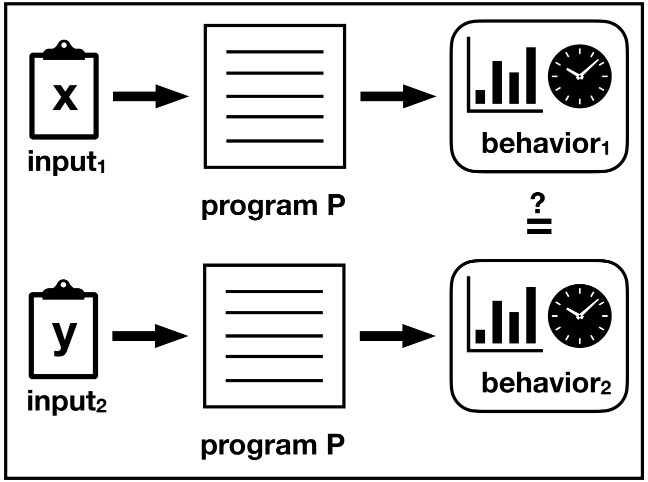
➥ identify behavioral differences



Differential Software Testing

➥ identify behavioral differences

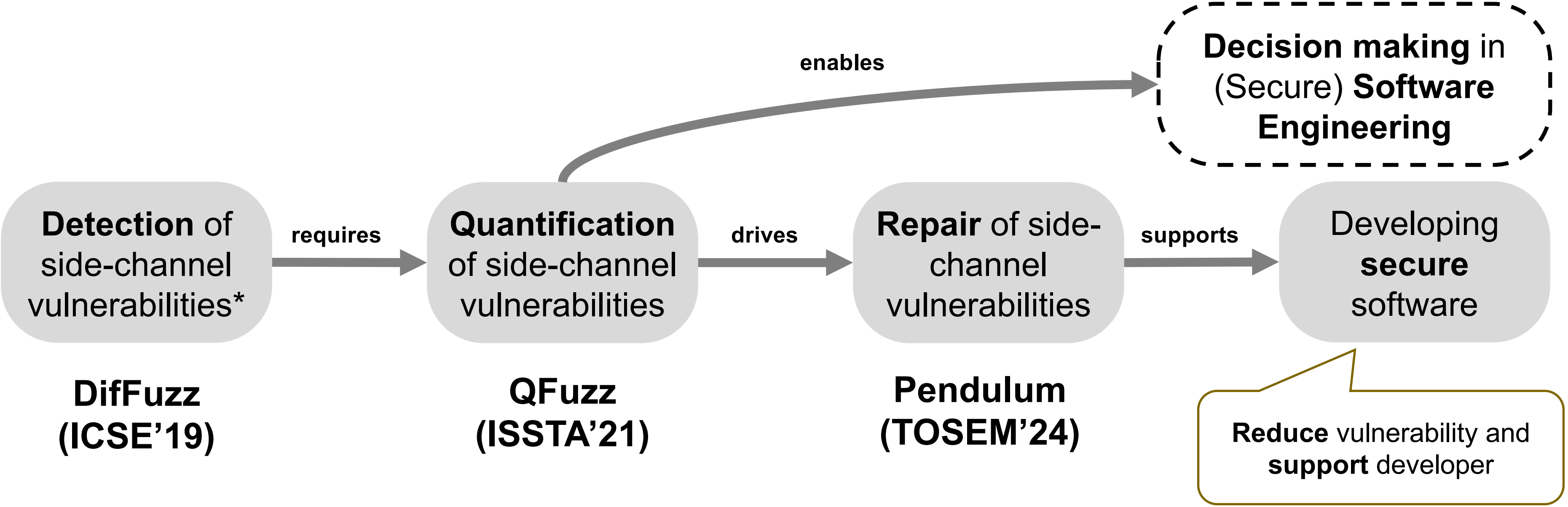
- for the **same** program with **two different** inputs
  ➥ security, reliability
- for example,
  - Worst-Case Complexity Analysis
  - Side-Channel Analysis
  - Robustness Analysis of Neural Network



2

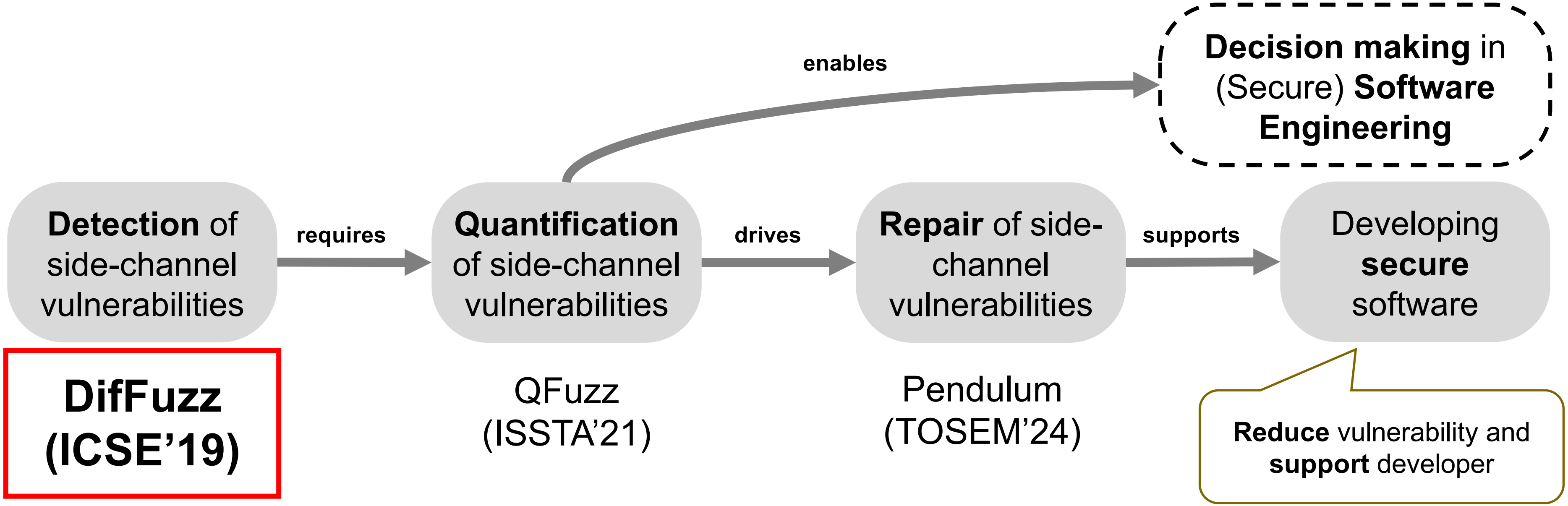
RUHR UNIVERSITÄT BOCHUM RUB

Path to Side-Channel Repair



** initially motivated by the DARPA Space/Time Analysis for Cybersecurity (STAC) program*

Path to Side-Channel Repair



2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

DIFFUZZ: Differential Fuzzing for Side-Channel Analysis

Shirin Nilizadeh*
University of Texas at Arlington
Arlington, TX, USA
shirin.nilizadeh@uta.edu

Yannic Noller*
Humboldt-Universität zu Berlin
Berlin, Germany
yannic.noller@hu-berlin.de

Corina S. Păsăreanu
Carnegie Mellon University Silicon Valley, NASA Ames Research Center
Moffett Field, CA, USA

***Abstract*—Side-channel attacks allow an adversary to uncover secret program data by observing the behavior of a program with respect to a resource, such as execution time, consumed memory or response size. Side-channel vulnerabilities are difficult to reason about as they involve analyzing the correlations between resource usage over multiple program paths. We present DIFFUZZ, a fuzzing-based approach for detecting side-channel vulnerabilities related to time and space. DIFFUZZ automatically detects these vulnerabilities by analyzing two versions of the program and using resource-guided heuristics to find inputs that *maximize* the difference in resource consumption between secret-dependent paths. The methodology of DIFFUZZ is general and can be applied to programs written in any language. For this paper, we present an implementation that targets analysis of JAVA programs, and uses and extends the KELINCI and AFL fuzzers. We evaluate DIFFUZZ on a large number of JAVA programs and demonstrate that it can reveal unknown side-channel vulnerabilities in popular applications. We also show that DIFFUZZ compares favorably against BLAZER and THEMIS, two state-of-the-art analysis tools for finding side-channels in JAVA**

Given a program whose inputs are partitioned into public and secret variables, DIFFUZZ uses a form of differential fuzzing to automatically find program inputs that reveal side channels related to a specified resource, such as time, consumed memory, or response size. We focus specifically on timing and space related vulnerabilities, but the approach can be adapted to other types of side channels, including cache based.

Differential fuzzing has been successfully applied before for finding bugs and vulnerabilities in a variety of applications, such as LF and XZ parsers, PDF viewers, SSL/TLS libraries, and C compilers [36], [38], [41]. However, to the best of our knowledge, we are the first to explore differential fuzzing for side-channel analysis. Typically such fuzzing techniques analyze different versions of a program, attempting to discover bugs by observing differences in execution for the same inputs. In contrast DIFFUZZ works by analyzing two copies of the same program, with the same public inputs but with

- uses **differential fuzzing** to automatically find side-channel vulnerabilities
- outperforms static analysis techniques
- applies on **system level**
- cannot tell how severe a vulnerability might be

S. Nilizadeh, Y. Noller and C. S. Pasareanu, "DifFuzz: Differential Fuzzing for Side-Channel Analysis", ICSE'2019, https://doi.org/10.1109/ICSE.2019.00034

Side-Channel Analysis (continued)

- *secure* if the secret data can not be inferred by an attacker through their observations of the system (aka *non-interference*)
- can be solved by self-composition [Barthe2004]

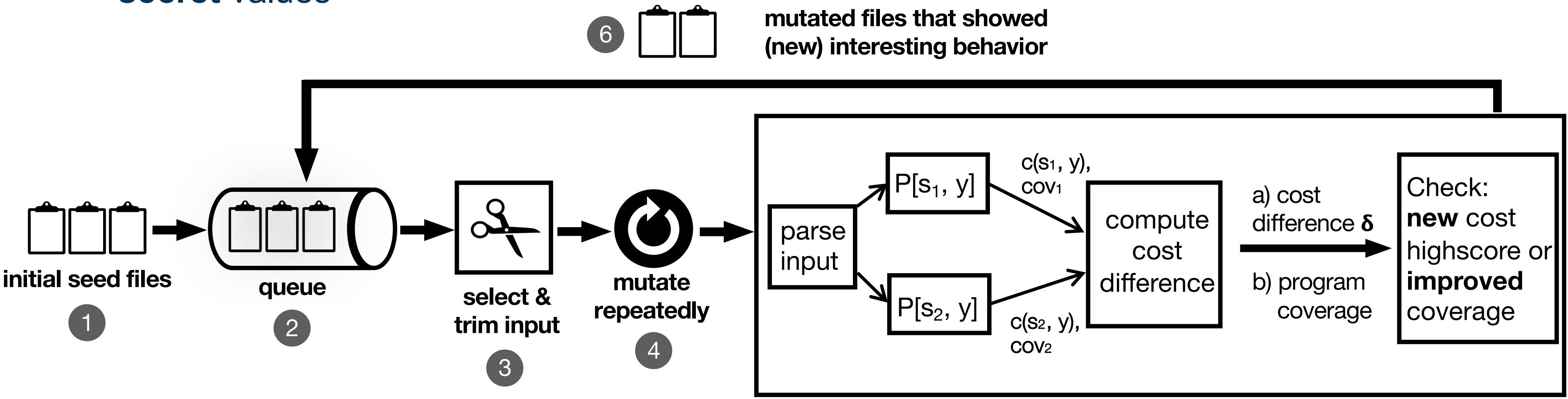
program execution	$P[pub, sec_1]$
cost observation	$c(P[pub, sec_1])$
two secret values	$c(P[pub, sec_1]) \quad c(P[pub, sec_2])$
equivalence	$c(P[pub, sec_1]) = c(P[pub, sec_2])$

$$\forall\, pub, sec_1, sec_2\colon\ c(P[pub, sec_1]) = c(P[pub, sec_2])$$

Barthe, G., D'Argenio, P. R., & Rezk, T. "Secure information flow by self-composition", IEEE Computer Security Foundations Workshop, 2004.

Fuzzing for Side-Channels (DiffFuzz, ICSE'19)

- **key aspect**: search for path, for which side-channel observation differs because of **secret** values



maximize$_{pub, sec_1, sec_2}$ $\delta = |c(P[pub, sec_1]) - c(P[pub, sec_2])|$

5 **mutant selection by input evaluation for the instrumented program P**

```
boolean pwcheck_unsafe (byte[] pub, byte[] sec) {
    if (pub.length != sec.length) {
        return false;
    }
    for (int i = 0; i < pub.length; i++) {
        if (pub[i] != sec[i]) {
            return false;
        }
    }
    return true;
}
```

Example Results

Initial Input: $cost_{Diff}$ = 0

```
secret1 = [72, 101, 108, 108, 111,  32,  67]
secret2 = [97, 114, 110, 101, 103, 105, 101]
public  = [32,  77, 101, 108, 108, 111, 110]
```

$cost_{Diff}$ > 0 after ~ 5 sec

Input with highscore $cost_{Diff}$ = 47 after ~ 69 sec
(maximum length = 16 bytes):

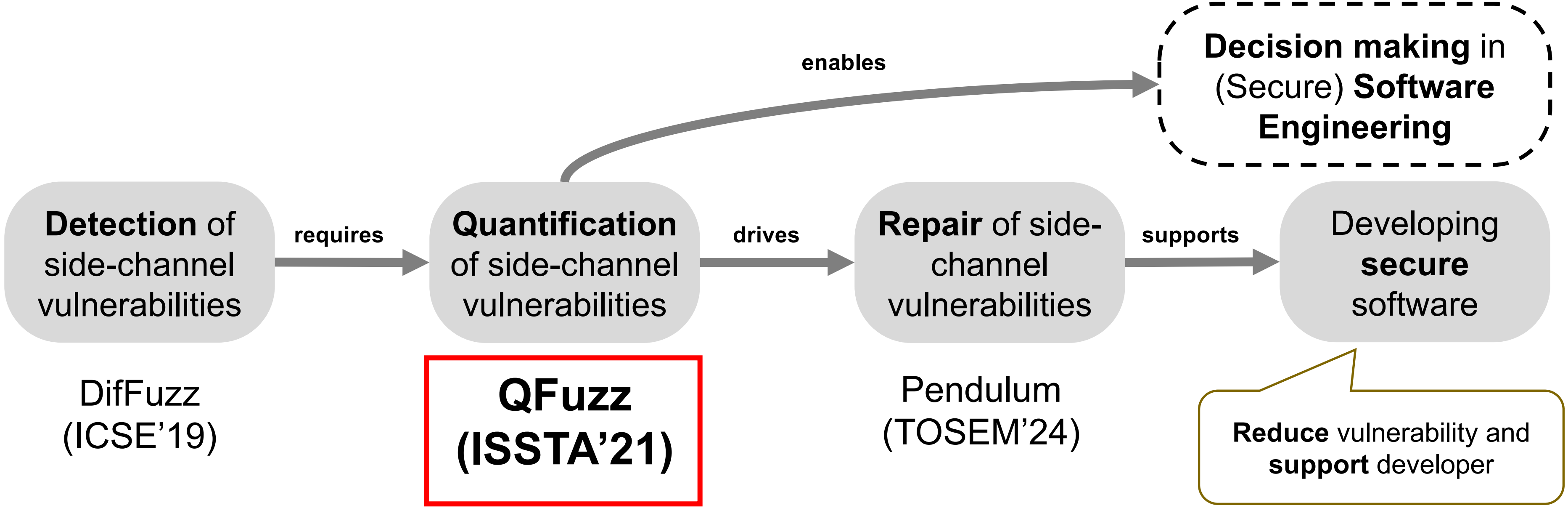
```
secret1 = [72, 77, -16, -66, -48, -48, -48, -48, -28, 0, 100, 0, 0, 0, 0, -48]
secret2 = [-48, -4, -48, 7, 17, 0, -24, -48, -48, 16, -48, -3, 108, 72, 32, 0]
public  = [-48, -4, -48, 7, 17, 0, -24, -48, -48, 16, -48, -3, 108, 72, 32, 0]
```

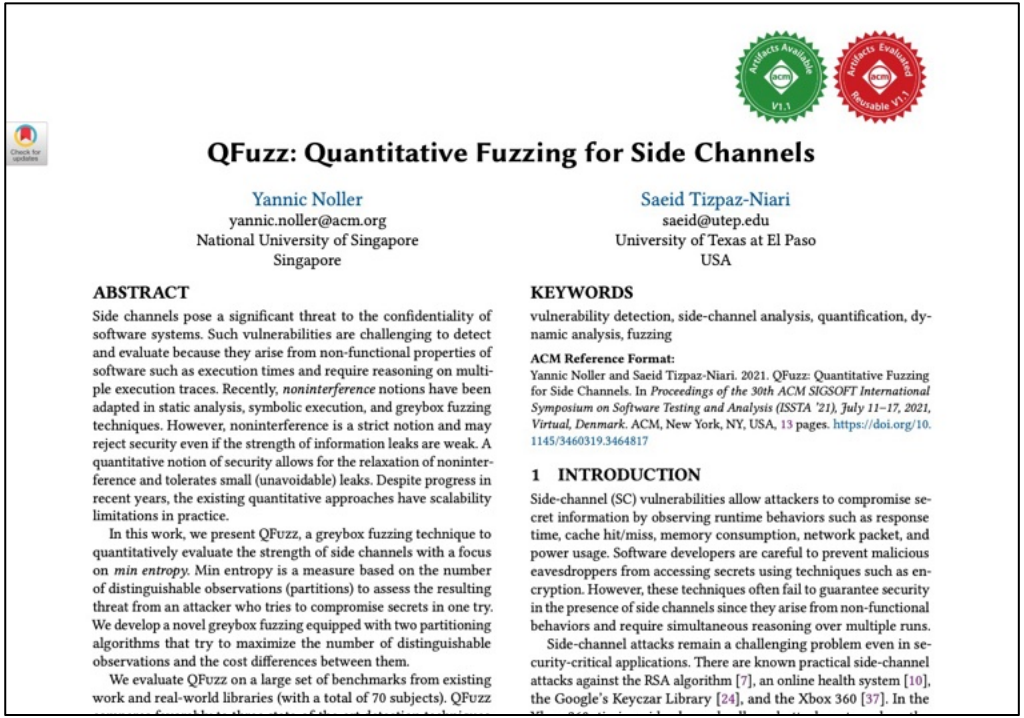
Is there a vulnerability?

⇔

How much information can be leaked?

Path to Side-Channel Repair





QFuzz: Quantitative Fuzzing for Side Channels

Yannic Noller
yannic.noller@acm.org
National University of Singapore
Singapore

Saeid Tizpaz-Niari
saeid@utep.edu
University of Texas at El Paso
USA

ABSTRACT

Side channels pose a significant threat to the confidentiality of software systems. Such vulnerabilities are challenging to detect and evaluate because they arise from non-functional properties of software such as execution times and require reasoning on multiple execution traces. Recently, *noninterference* notions have been adapted in static analysis, symbolic execution, and greybox fuzzing techniques. However, noninterference is a strict notion and may reject security even if the strength of information leaks are weak. A quantitative notion of security allows for the relaxation of noninterference and tolerates small (unavoidable) leaks. Despite progress in recent years, the existing quantitative approaches have scalability limitations in practice.

In this work, we present QFuzz, a greybox fuzzing technique to quantitatively evaluate the strength of side channels with a focus on *min entropy*. Min entropy is a measure based on the number of distinguishable observations (partitions) to assess the resulting threat from an attacker who tries to compromise secrets in one try. We develop a novel greybox fuzzing equipped with two partitioning algorithms that try to maximize the number of distinguishable observations and the cost differences between them.

We evaluate QFuzz on a large set of benchmarks from existing work and real-world libraries (with a total of 70 subjects). QFuzz

KEYWORDS

vulnerability detection, side-channel analysis, quantification, dynamic analysis, fuzzing

ACM Reference Format:
Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464817

1 INTRODUCTION

Side-channel (SC) vulnerabilities allow attackers to compromise secret information by observing runtime behaviors such as response time, cache hit/miss, memory consumption, network packet, and power usage. Software developers are careful to prevent malicious eavesdroppers from accessing secrets using techniques such as encryption. However, these techniques often fail to guarantee security in the presence of side channels since they arise from non-functional behaviors and require simultaneous reasoning over multiple runs.

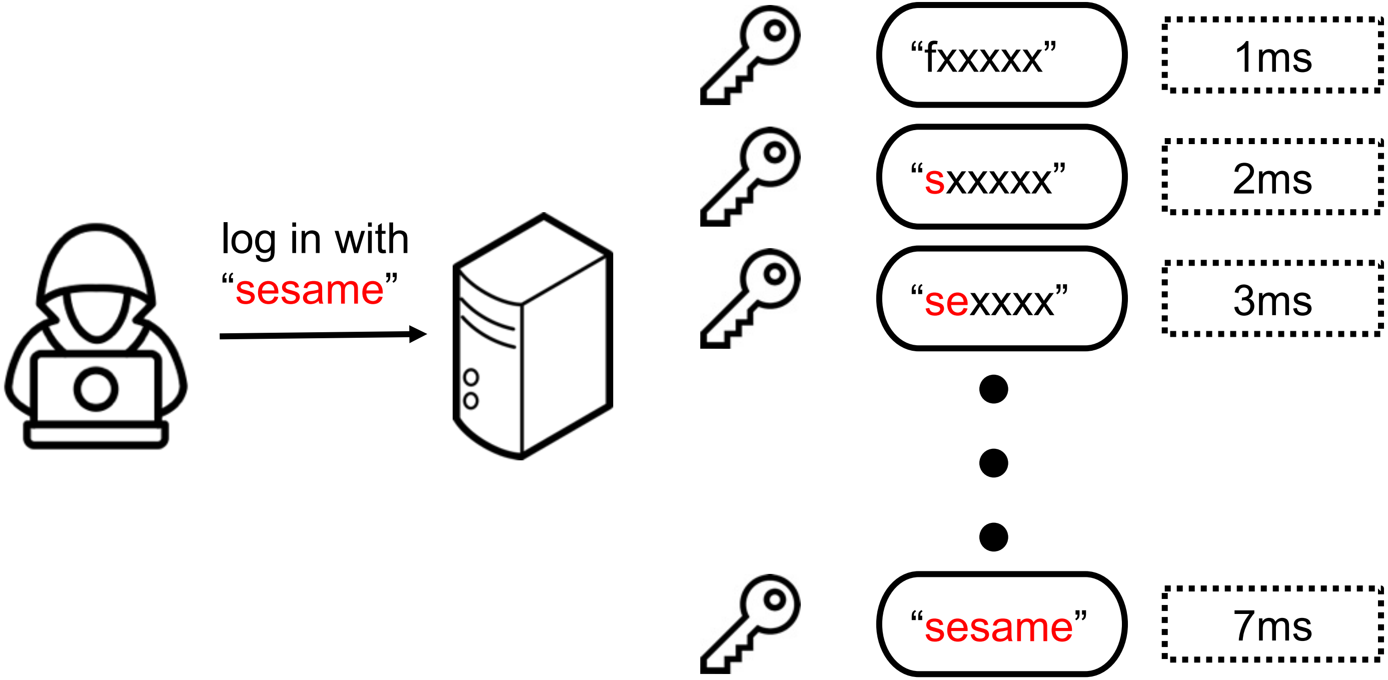
Side-channel attacks remain a challenging problem even in security-critical applications. There are known practical side-channel attacks against the RSA algorithm [7], an online health system [10], the Google's Keyczar Library [24], and the Xbox 360 [37]. In the

- uses **greybox fuzzing** to quantitatively evaluate the **strength** of side channels
- focuses on min entropy
- explores two **partitioning** algorithms that try to **maximize** the number of **distinguishable** observations
- cannot localize the vulnerability
- published at **ISSTA**'2021

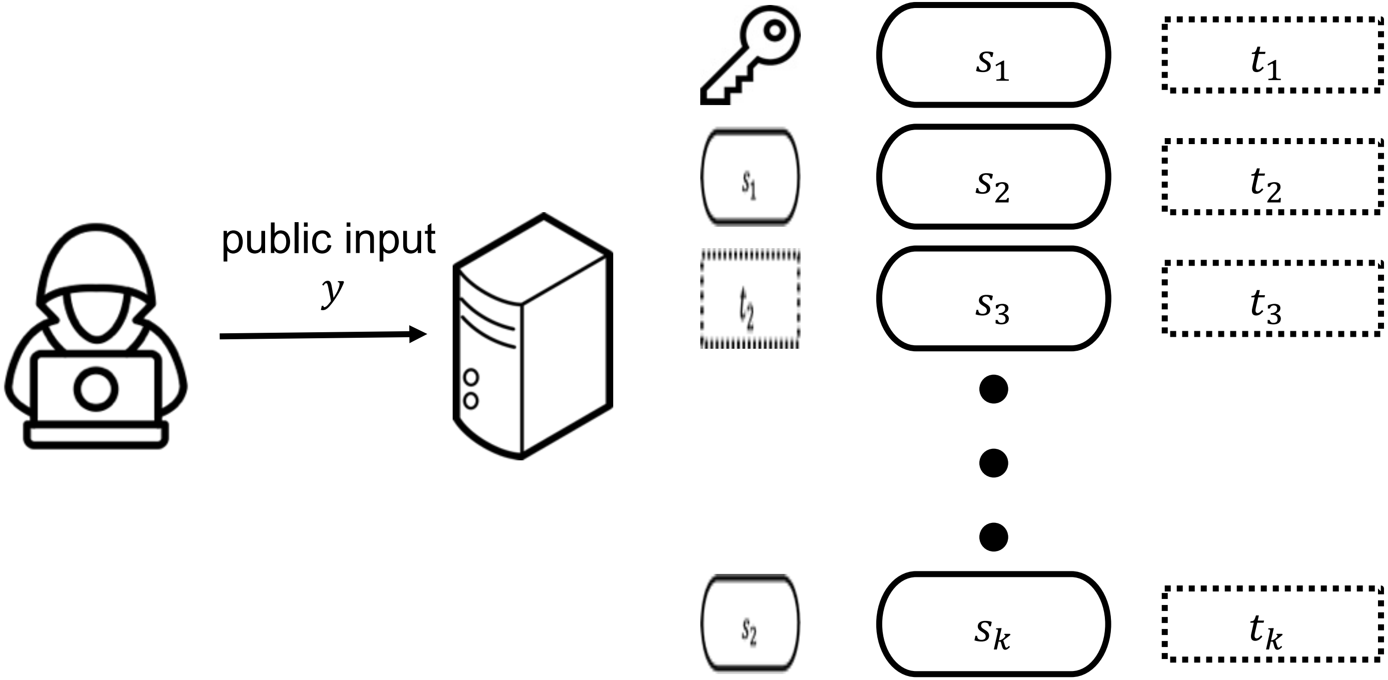
Yannic Noller and Saeid Tizpaz-Niari, "QFuzz: quantitative fuzzing for side channels", ISSTA 2021
https://doi.org/10.1145/3460319.3464817

RUHR UNIVERSITÄT BOCHUM RUB

Timing SC Vulnerability: An Example



Timing SC Vulnerability: Quantification



Threat Model

- We adapt our threat model from a **chosen-message attack** [CCS'07]
- i.e., an adversary picks an **ideal** public input to compromise secret inputs in **one trial**
- **Offline:** The attacker, who has access to the source code, can sample secret and public inputs on their local machine arbitrarily many times and construct an ideal public input that partitions the secret into many classes of timing observations.
- **Online**: The attacker queries the target application with the best guess, observes side channels, and maps the observation to a partition of secret inputs.

Boris Köpf and David Basin. 2007. An Information-Theoretic Model for Adaptive Side-Channel Attacks. CCS '07. https://doi.org/10.1145/1315245.1315282

Quantification (QFuzz, ISSTA'21)

Threat Model

Attacker can pick an **ideal public input** to compromise the **secret value** or some properties of it in **one try**.

Information Leakage: min-entropy [Smith2009]

Assuming that the program P is deterministic and the distribution over secret input Σ is uniform, then the information leakage can be characterized $\boldsymbol{log_2 k^*}$ (ε=0).

maximum number of classes in the cost observations

$\boldsymbol{log_2 |\Sigma_{Y=y^*}|}$

$\varepsilon \geq 0$

① **How to identify such inputs?**

Problem Statement

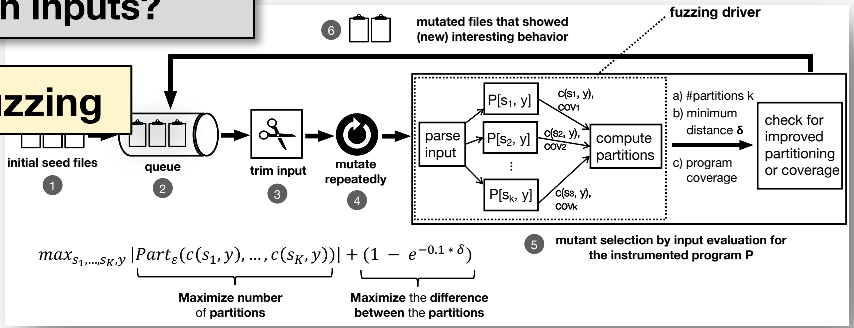
Find set of secret values Σ and public value y* that characterize the maximum number of observation classes with the highest distance δ.

② **How to characterize observation classes?**

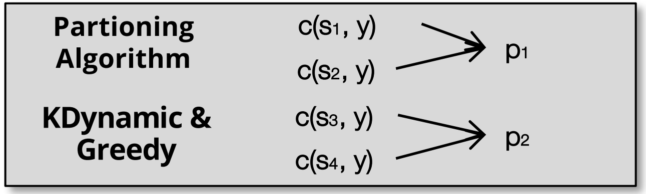
Quantification (QFuzz, ISSTA'21)

1 **How to identify such inputs?**

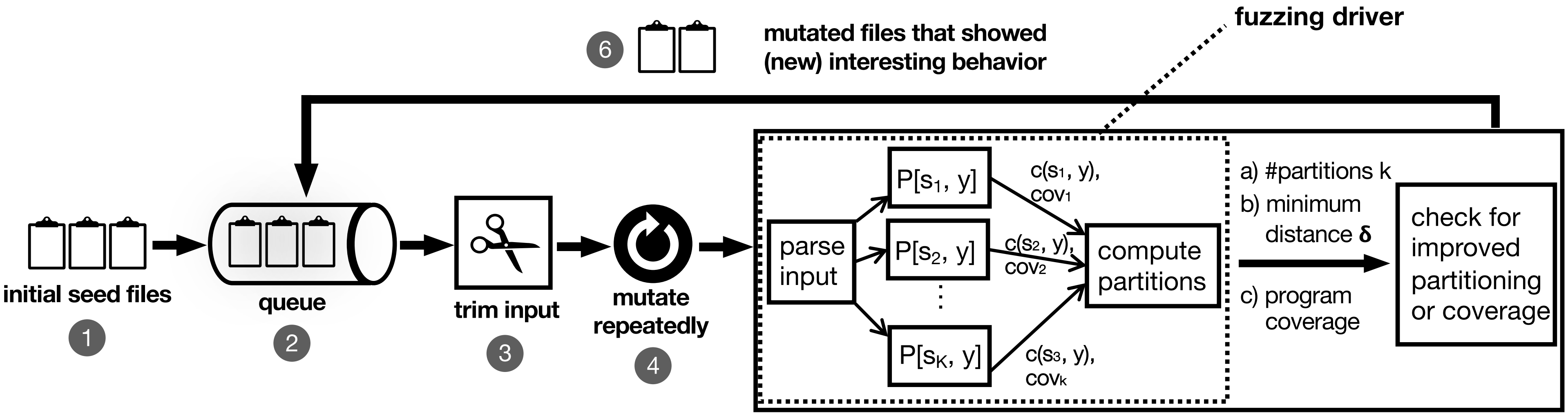
Fuzzing



2 **How to characterize observation classes?**



QFuzz: Workflow



5 **mutant selection by input evaluation for the instrumented program P**

$$max_{s_1,\ldots,s_K,y}\ \underbrace{|Part_\varepsilon(c(s_1,y),\ldots,c(s_K,y))|}_{\text{Maximize number of partitions}} + \underbrace{(1 - e^{-0.1 * \delta})}_{\text{Maximize the difference between the partitions}}$$

Maximize number of **partitions**

Maximize the **difference between** the **partitions**

Example (K=100, ε=1, length=16, count=bytecode-instruction)

K=17
δ=3

stringEquals (Original Jetty, v1)

```
boolean stringEquals(String s1, String s2) {
  if (s1 == s2)
    return true;
  if (s1 == null || s2 == null ||
      s1.length() != s2.length())
    return false;
  for (int i = 0; i < s1.length(); ++i)
    if (s1.charAt(i) != s2.charAt(i))
      return false;
  return true;
}
```

stringEquals (Current Jetty, v4)

```
boolean stringEquals(String s1, String s2) {
  if (s1 == s2) return true;
  if (s1 == null || s2 == null)
    return false;
  boolean result = true;
  int l1 = s1.length();
  int l2 = s2.length();
  for (int i = 0; i < l2; ++i)
    result &= s1.charAt(i%l1) == s2.charAt(i));
  return result && l1 == l2;
}
```

K=9
δ =1

K=1

stringEquals (Safe Jetty, v5)

```
boolean stringEquals(String s1, String s2) {
  if (s1 == s2) return true;
  if (s1 == null || s2 == null)
    return false;
  int l1 = s1.length();
  int l2 = s2.length();
  if(l2 == 0){return l1 == 0}
  int result |= l1 - l2;
  for (int i = 0; i < l2; ++i){
    int r = ((i - l1) >>> 31) * i;
    result |= s1.charAt(r) ^ s2.charAt(i);
  }
  return result == 0;
}
```

Equals (Unsafe Spring-Security)

```
boolean Equals(String s1, String s2) {
  if (s1 == null || s2 == null)
    return false;
  byte[] s1B = s1.getBytes("UTF-8");
  byte[] s2B = s2.getBytes("UTF-8");
  int len1 = s1B.length;
  int len2 = s2B.length;
  if (len1 != len2)
    return false;
  int result = 0;
  for (int i = 0; i < len2; i++)
    result |= s1B[i] ^ s2B[i];
  return result == 0;
}
```

K=2
δ =149

DifFuzz

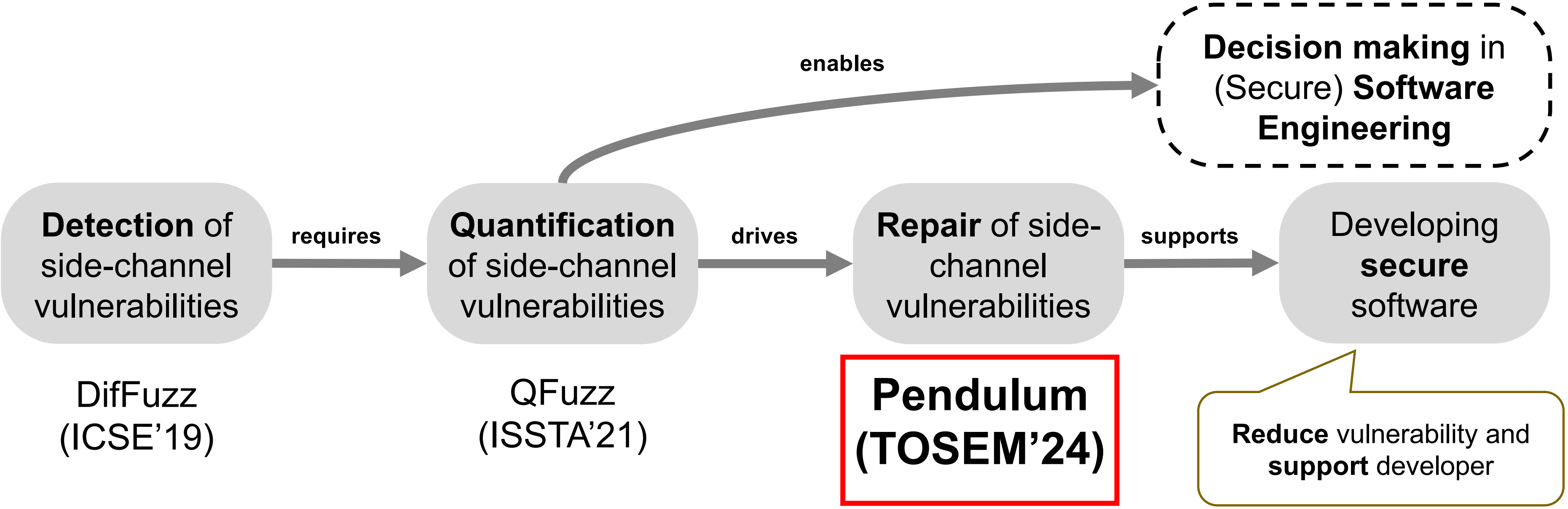
only leaks existence of special character

How much information can be leaked?

⇔

How can we fix the issue?

Path to Side-Channel Repair



RUHR UNIVERSITÄT BOCHUM RUB

Timing Side-Channel Mitigation via Automated Program Repair

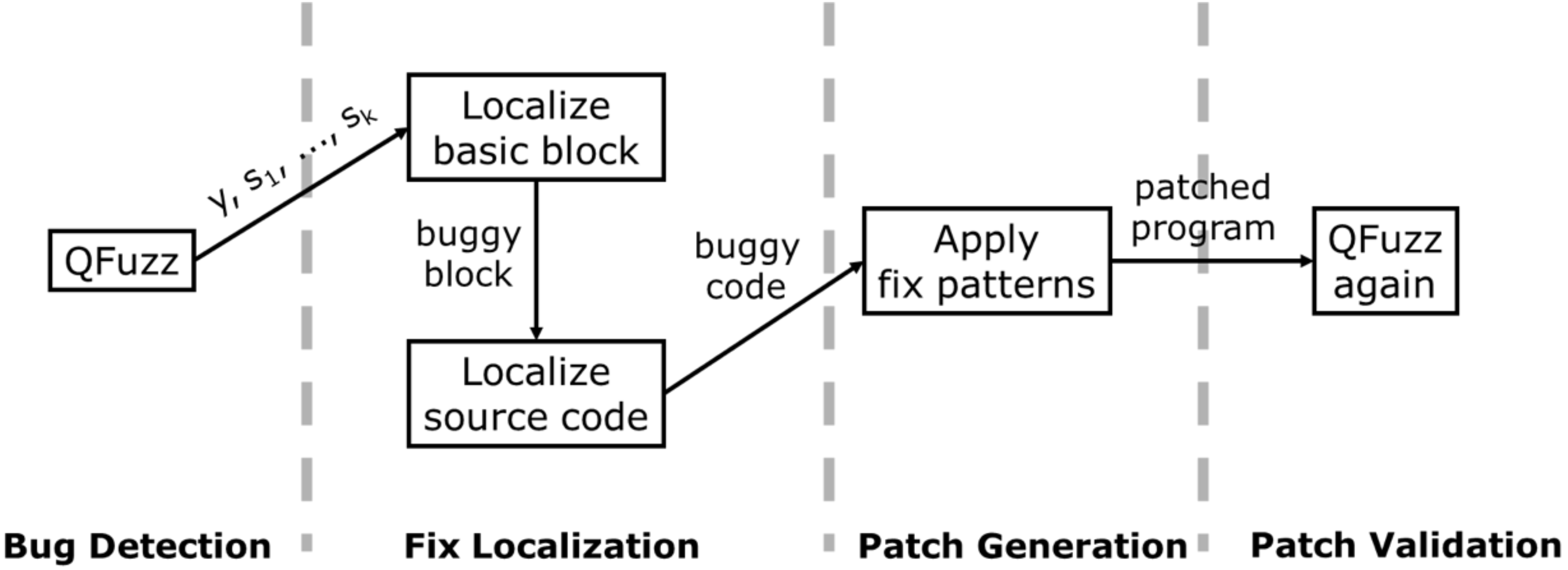
HAIFENG RUAN, National University of Singapore, Singapore, Singapore
YANNIC NOLLER, Ruhr University Bochum, Bochum, Germany
SAEID TIZPAZ-NIARI, University of Texas at El Paso, El Paso, TX, USA
SUDIPTA CHATTOPADHYAY, Singapore University of Technology and Design, Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore, Singapore

Side-channel vulnerability detection has gained prominence recently due to Spectre and Meltdown attacks. Techniques for side-channel detection range from fuzz testing to program analysis and program composition. Existing side-channel mitigation techniques repair the vulnerability at the IR/binary level or use runtime monitoring solutions. In both cases, the source code itself is not modified, can evolve while keeping the vulnerability, and the developer would get no feedback on how to develop secure applications in the first place. Thus, these solutions do not help the developer understand the side-channel risks in her code and do not provide guidance to avoid code patterns with side-channel risks. In this article, we present PENDULUM, the first approach for automatically locating and repairing side-channel vulnerabilities in the source code, specifically for timing side channels. Our approach uses a quantitative estimation of found vulnerabilities to guide the fix localization, which goes hand-in-hand with a pattern-guided repair. Our evaluation shows that PENDULUM can repair a large number of side-channel vulnerabilities in real-world applications. Overall, our approach integrates

- uses collected **observations** from QFuzz to **localize** the vulnerability
- applies **(safe) operators** to transform the source code
- can introduce side-effects
- published in TOSEM 2024

Haifeng Ruan, Yannic Noller, Saeid Tizpaz-Niari, Sudipta Chattopadhyay, and Abhik Roychoudhury. “Timing Side-Channel Mitigation via Automated Program Repair”, TOSEM 2024. https://doi.org/10.1145/3678169

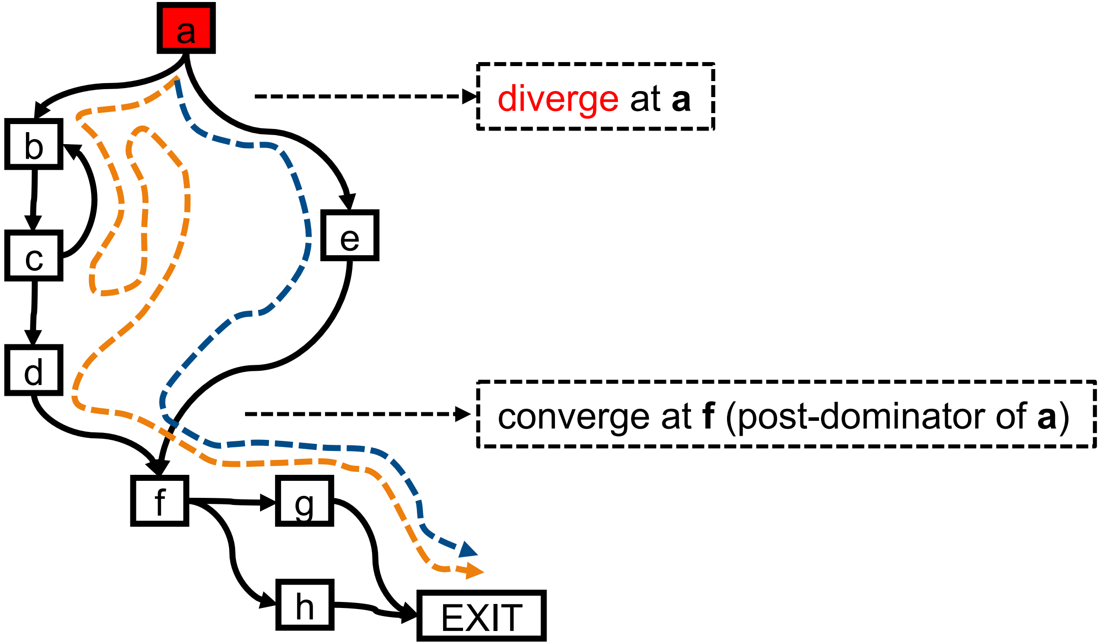
Pendulum – Repair Workflow (TOSEM'24)



Fix Localization (Basic Block)

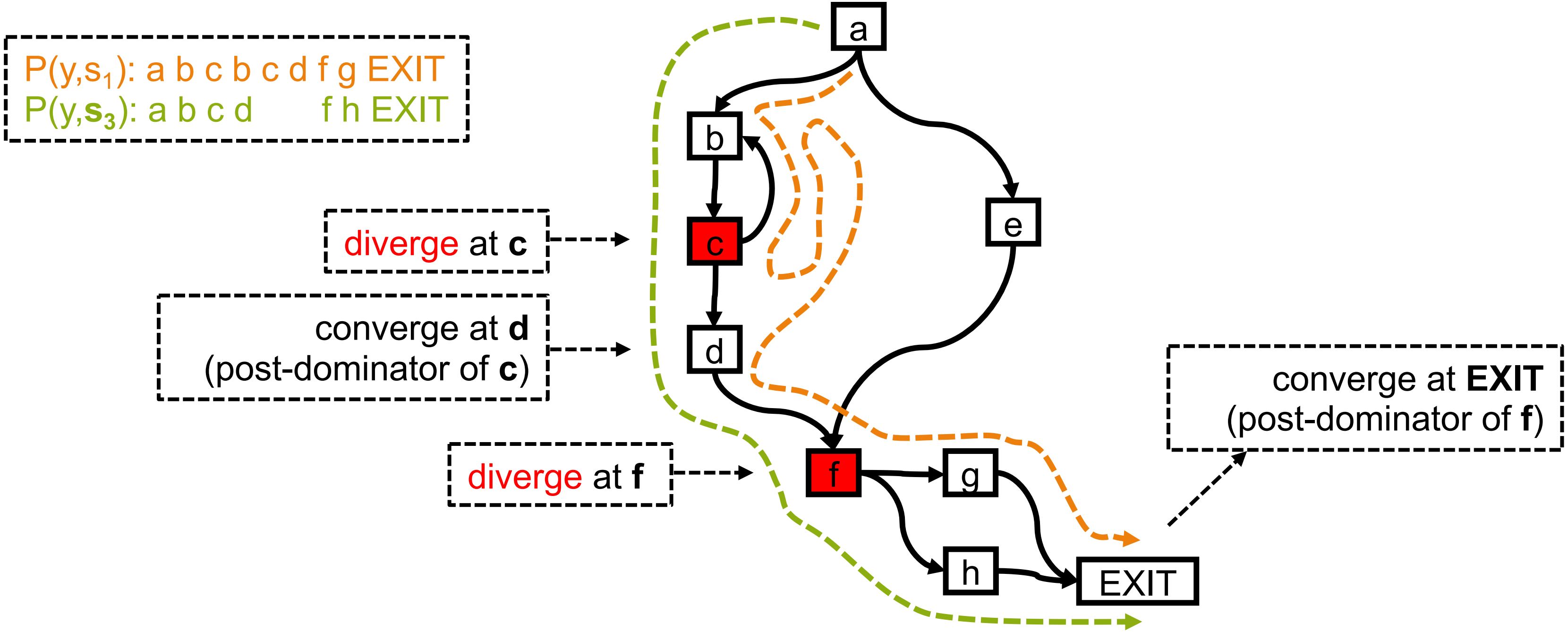
Compare traces to find where they diverge

$P(y,s_1)$: a b c b c d f g EXIT
$P(y,\mathbf{s_2})$: a e f g EXIT



Fix Localization (Basic Block)

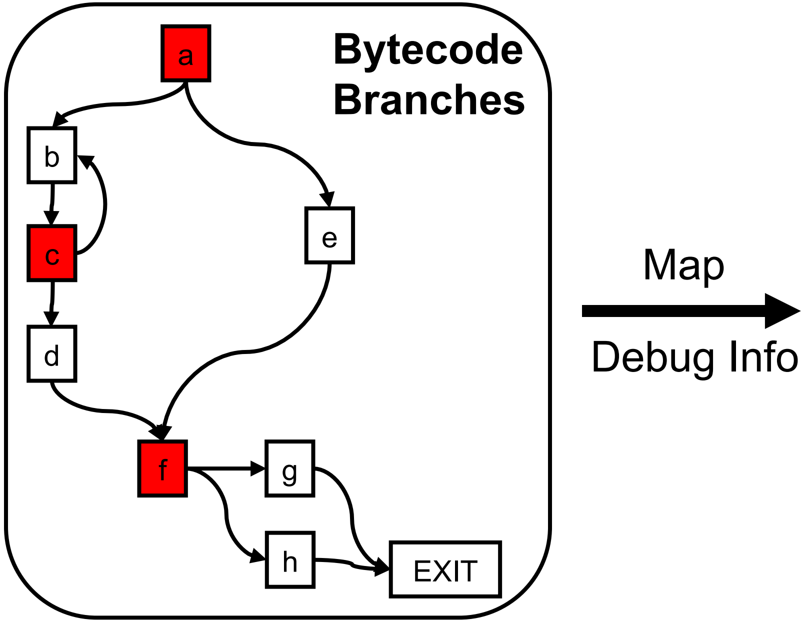
Compare traces to find where they diverge



RUHR UNIVERSITÄT BOCHUM RUB

Fix Localization (Source Code)

Map conditional branches to source code

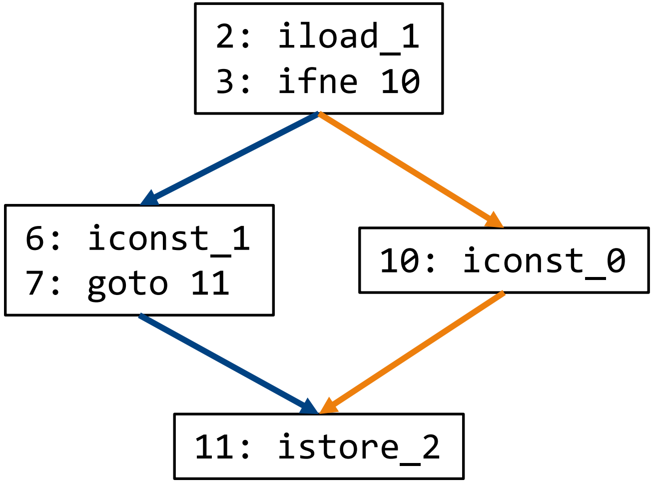


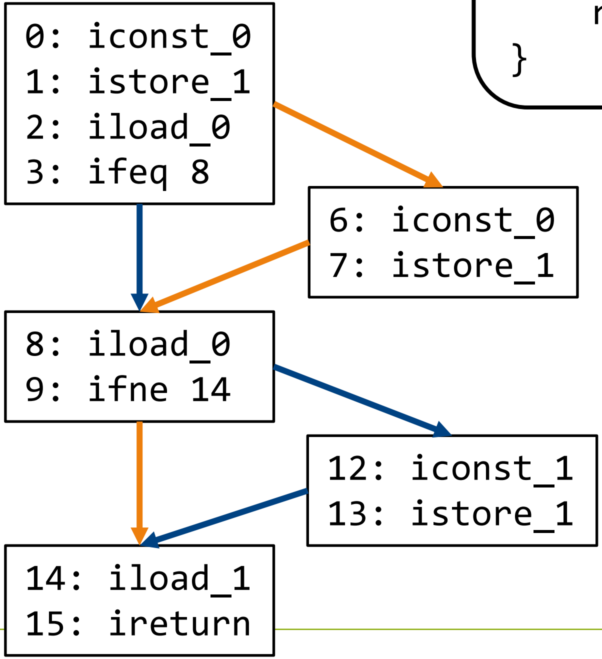
Source Code

1. **If Statement**
2. **Loop Statements**
   for, while, do...while
3. **Unsafe Operators**
   !, >, <, >=, <=, ==, !=, &&, ||, ?:

Fix Patterns (Unsafe Operators)

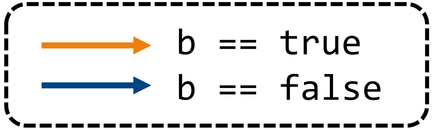
```
boolean b = !a;
```
➡
```
boolean b = not(a);
```






```
boolean not (boolean b) { // !
    boolean result = false;
    if (b) result = false;
    if (!b) result = true;
    return result;
}
```

constant-time utility methods



Fix Patterns (If Statement)

Turn branches into conditional assignments

```
+ boolean cond = condExp;
- if (condExp) {
        ...
-       var1 = exp1;
+       var1 = ite(cond, exp1, var1);
        ...
- } else {
        ...
-       var2 = exp2;
+       var2 = ite(cond, var2, exp2);
        ...
- }
```

```
<T> T ite (boolean cond, T t1, T t2)
{ // ?:
    T t = null;
    if (cond) t = t1;
    if (!cond) t = t2;
    return t;
}
```

constant-time utility methods

Fix Patterns (If Statement)

What if there is an early return / break / continue?

```
+ boolean earlyReturn = false;
+ RT returnValue = DEFAULT_VALUE;
  ...
  if (condExp) {
      ...
-     return x;
+     returnValue = x;
+     earlyReturn = true;
  }
  ...
- return y;
+ return ite(earlyReturn, returnValue, y);
```

then use the pattern from the previous slide

Fix Patterns (Loop Statement)

Iterate for a constant number of times

```
+ int ub = estimatedLoopBound;
- for (...; condExp; ...) {
+ for (...; --ub > 0; ...) {
+     if (!condExp) {
+         break;
+     }
  }
```

then fix this IF

Research Questions

- **RQ1 (Fix localization)** Can Pendulum find the correct fix locations for the side-channel vulnerabilities?
- **RQ2 (Vulnerability mitigation)** To what extent does Pendulum mitigate the side-channel vulnerabilities?
- **RQ3 (Side effect)** Does Pendulum preserve the functionality of the program-to-fix?
- **RQ4 (Time and space impact)** How do the generated patches influence the execution time of the programs? How large are the patches?

RUHR UNIVERSITÄT BOCHUM RUB

Evaluation

- focus on timing side-channel vulnerabilities
  - secret-dependent unsafe operators, if statements, and loop statements
- **42 subjects** taken from QFuzz benchmark and other well-known Java security projects
  - e.g., Apache FTPServer, Eclipse Jetty, JDK, OrientDB, Picketbox, Spring-Security, ...
- comparison to **DifFuzzAR**: DifFuzz-based repair approach
  - driver as localizer
  - removes early exits (elimination of all return statements but one)
  - adapts control-flow (modifies stopping condition, replication of block statements)

RQ1: Fix Localization

- we compare the identified fix locations with that of the developer fix for Pendulum and DifFuzzAR
- **Pendulum identifies the fix locations successfully for all 42 subjects**
- while **DifFuzzAR** fails for **13** subjects: limited fix localization supported

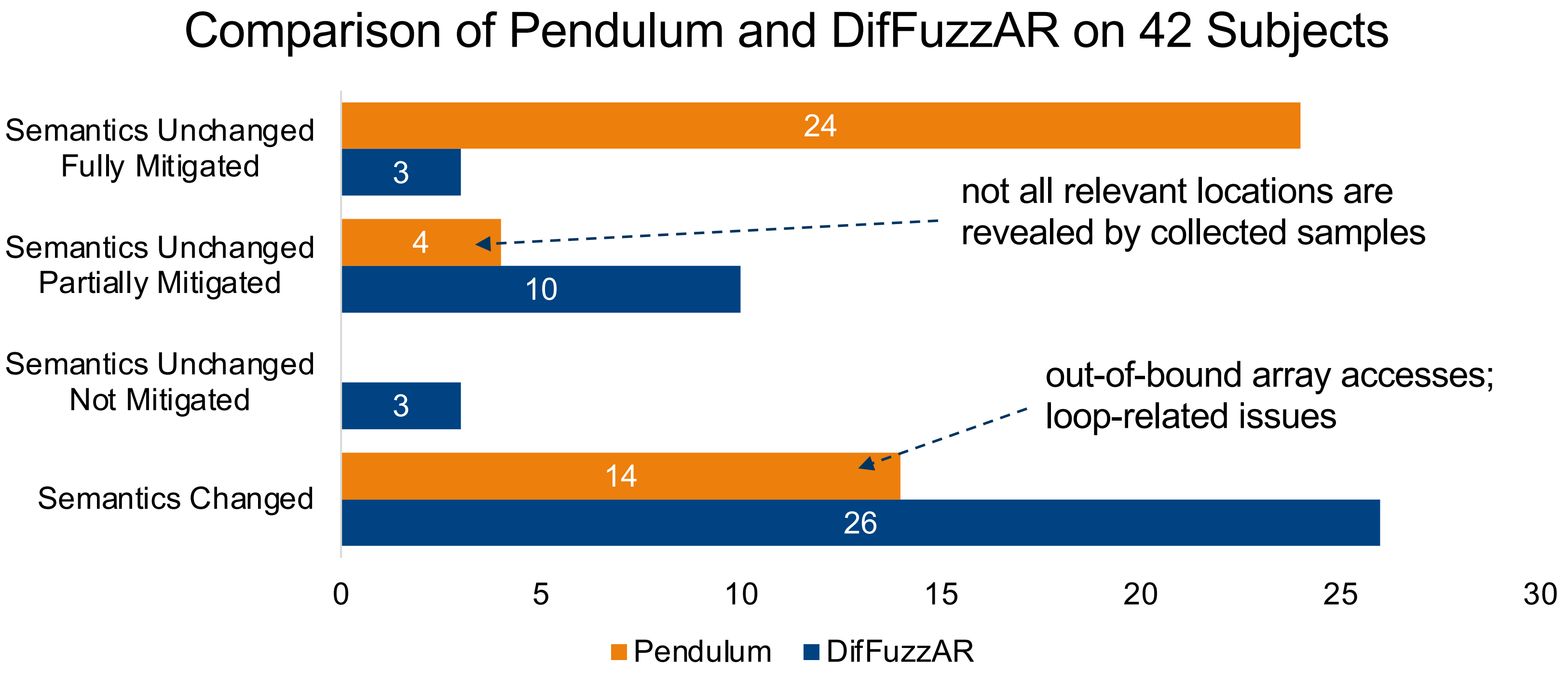
RQ2: Vulnerability Mitigation

- compare the number of side-channel partitions between the original program, the Pendulum-fixed program, and the developer fix
- **Pendulum** is able to mitigate the vulnerability effectively for **33** of 42 (79%) subjects.
- for **26** of these 33 subjects, Pendulum can **entirely eliminate** the side-channel vulnerability
- in contrast, **DifFuzzAR** can mitigate the vulnerability for only **15** (36%) subjects

RQ3: Side Effects

EVOSUITE
Automatic Test Suite Generation for Java

RQ3: Side Effects



RUHR UNIVERSITÄT BOCHUM

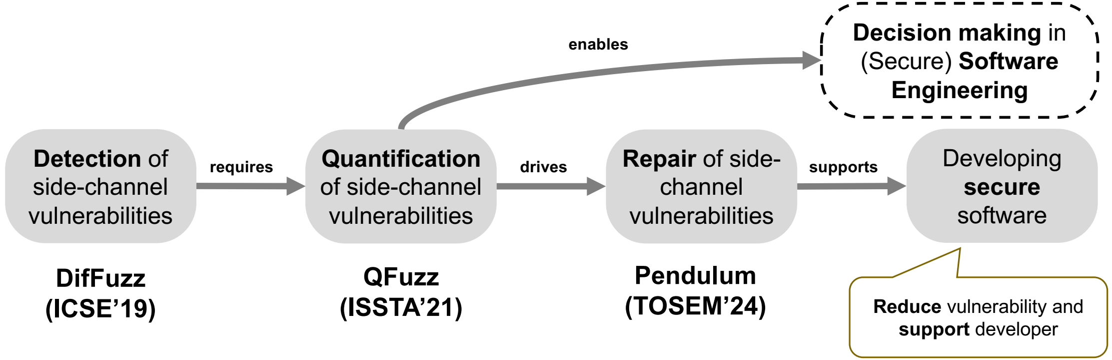
RQ4: Time and Space Impact

- The Pendulum-generated repairs have an **average slowdown** of **43%** and a **median slowdown** of **3%**.
- This performance is close to that of the developer fixes.
- Our median repairs are five lines larger than the original code and six lines larger than the developer fixes.

Subject	Side-Channel Partitions				Average Execution Time (msec)			
	Orig	PDL-1	PDL-2	PDL-3	Orig	PDL-1	PDL-2	PDL-3
Eclipse_jetty_4	9	2	1	-	17 ± 8	16 ± 4	16 ± 6	-
rsa_modpow_1717	49	39	21	1	14 ± 6	14 ± 3	14 ± 4	20 ± 5
rsa_modpow_1964903306	71	39	12	2	14 ± 7	14 ± 4	14 ± 3	18 ± 7
rsa_modpow_834443	69	62	15	2	16 ± 6	17 ± 3	17 ± 4	22 ± 5

Summary: Automated Detection, Quantification, and Repair of Side-Channel Vulnerabilities

- **localizing** timing side-channel vulnerabilities
- **mitigating** them at source code automatically
- integrates with quantitative **fuzzing**

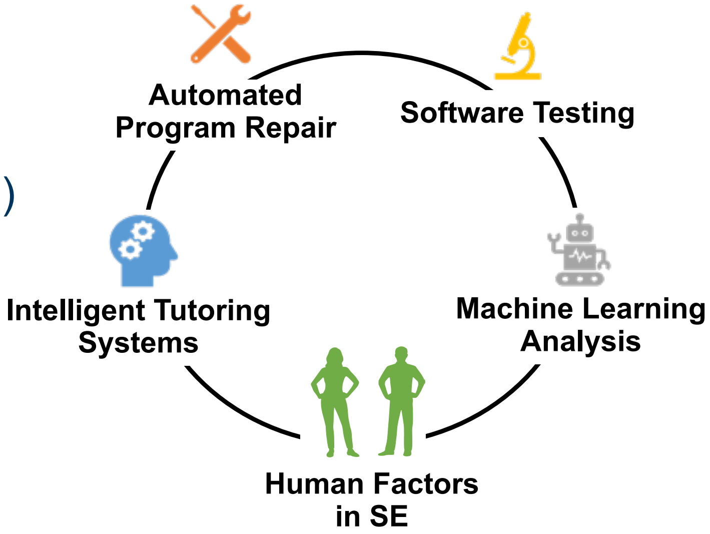


- **Trusted** Automatic Programming → **Trusted** Automated Software Engineering
- in the context of more and more automated programming:
  - explore **unified** processes/workflows, i.e., bring testing and repair closer together
- Fuzzing Shifting **Left**

RUHR UNIVERSITÄT BOCHUM RUB

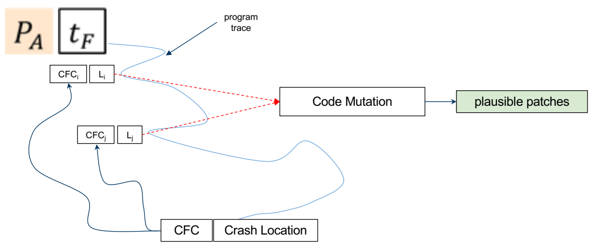
Other things we work on

- Trusted Automatic Programming
  - APR in the era of Large Language Models (LLM)
  - Agentic Workflows for APR
  - Repair of Machine Learning models
- Human Studies in SE
  - Developer surveys: Fuzzing + APR
- Intelligent Tutoring Systems
  - Simulated Interactive Debugging

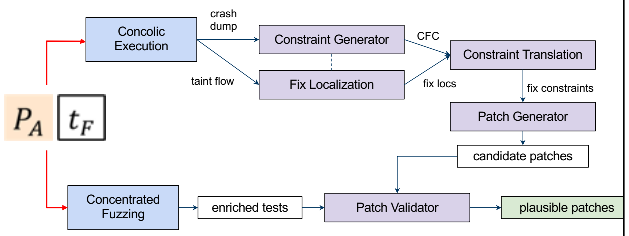


RUHR UNIVERSITÄT BOCHUM RUB

CrashRepair: Key Idea



CrashRepair: Overview



Comparison with SOTA

Tool	# Plausible	# Correct
CrashRepair	29	19
SenX	12	3
ExtractFix	12	5
VulnFix	17	9
CPR	35	9

evaluated on 41 subjects in VulnLoc benchmark with 1hr timeout

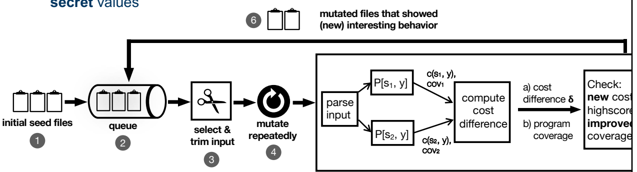
CrashRepair generates a **plausible** patch for **29 instances** without additional information

CrashRepair generates more plausible patches than SenX, ExtractFix and VulnFix

CrashRepair is **more effective** than existing state-of-the-art for vulnerability repair

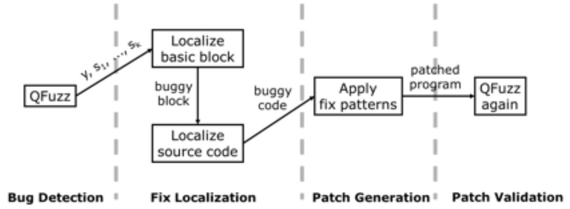
Fuzzing for Side-Channels (DifFuzz, ICSE'19)

- **key aspect**: search for path, for which side-channel observation differs because of **secret** values

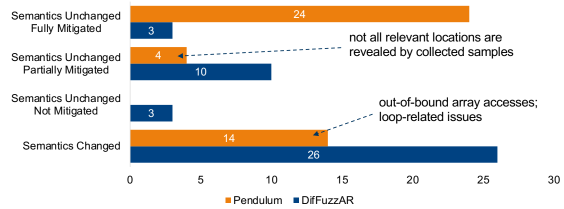


$$\text{maximize}_{pub, sec_1, sec_2} \quad \delta = |c(P[pub, sec_1]) - c(P[pub, sec_2])|$$

Pendulum – Repair Workflow (TOSEM'24)



RQ3: Side Effects



Prof. Dr. **Yannic Noller**
yannic.noller@rub.de
https://yannicnoller.github.io/

Automated Program Repair for Security

RUHR UNIVERSITÄT BOCHUM RUB