RUB

RUHR-UNIVERSITÄT BOCHUM

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

26.02.2025 – IFIP WG 2.5 – Singapore

Prof. Dr. Yannic Noller
Software Quality group

Research Objective

- provide **reliable**, **trustworthy**, and **secure** software systems
- contribute to **(trusted) automated software engineering process**
  - automated software **testing** and automated program **repair**
  - explore the potential of **human** involvement in such process
    - automated support for student education
- in particular, in the context of more and more automated programming:
  - explore **unified** processes/workflows, i.e., bring testing and repair closer together

Research Objective (continued)

→ **This talk:** demonstrate **unified** process for software **side-channel vulnerabilities**

- timing side-channel vulnerabilities
- 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

Side-Channel Analysis

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

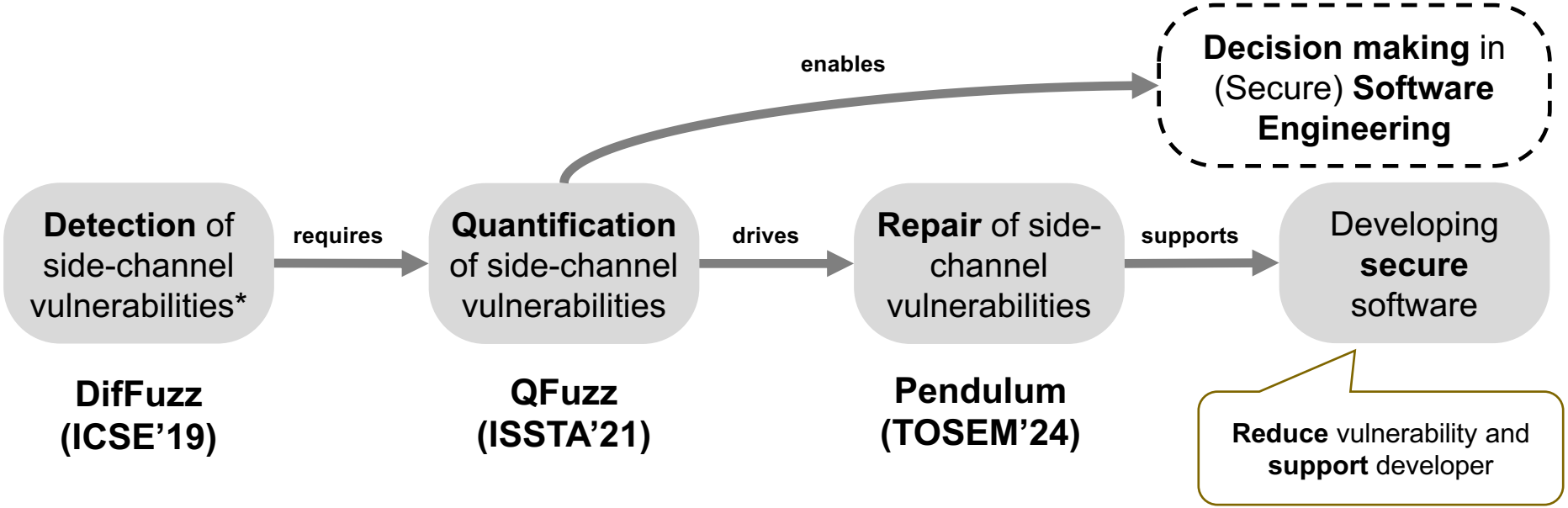
```
0  boolean pwcheck_unsafe (byte[] pub, byte[] sec) {
1      if (pub.length != sec.length) {
2          return false;
3      }
4      for (int i = 0; i < pub.length; i++) {
5          if (pub[i] != sec[i]) {
6              return false;
7          }
8      }
9      return true;
10 }
```

conditional early return causes leakage

Where do we find them?

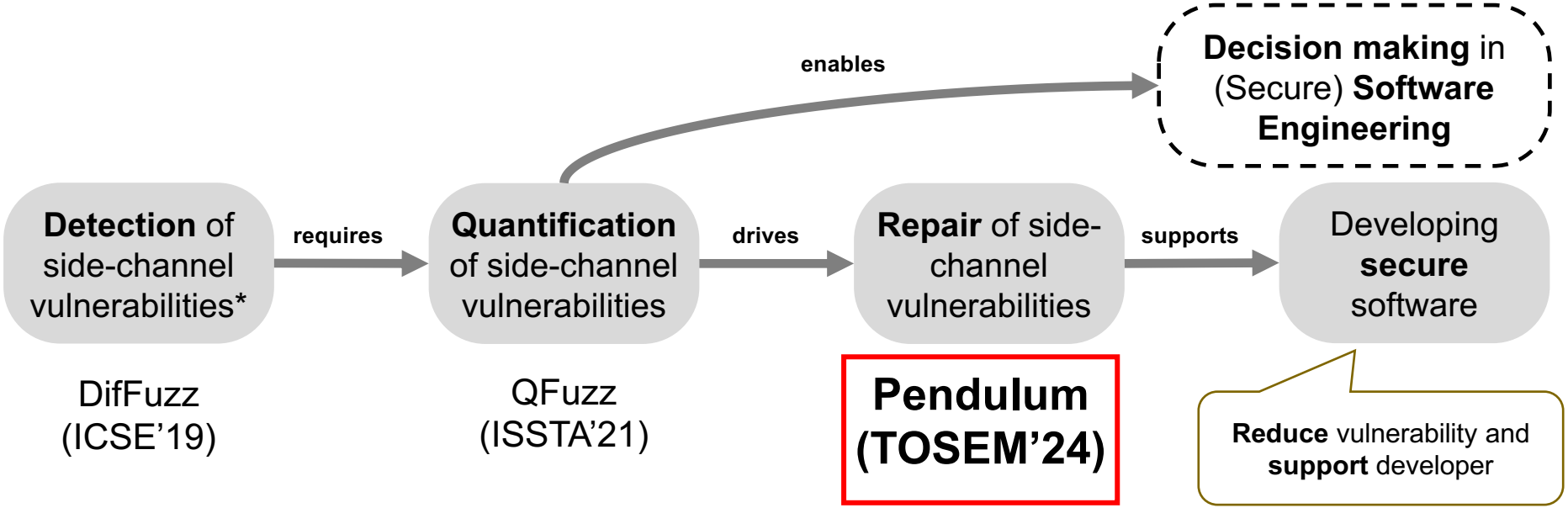
- ❑ application code, e.g., *Apache Tomcat, FtpServer, …*
- ❑ security libraries, e.g., *JDK, spring security, Bouncy Castle, …*

Path to Side-Channel Repair

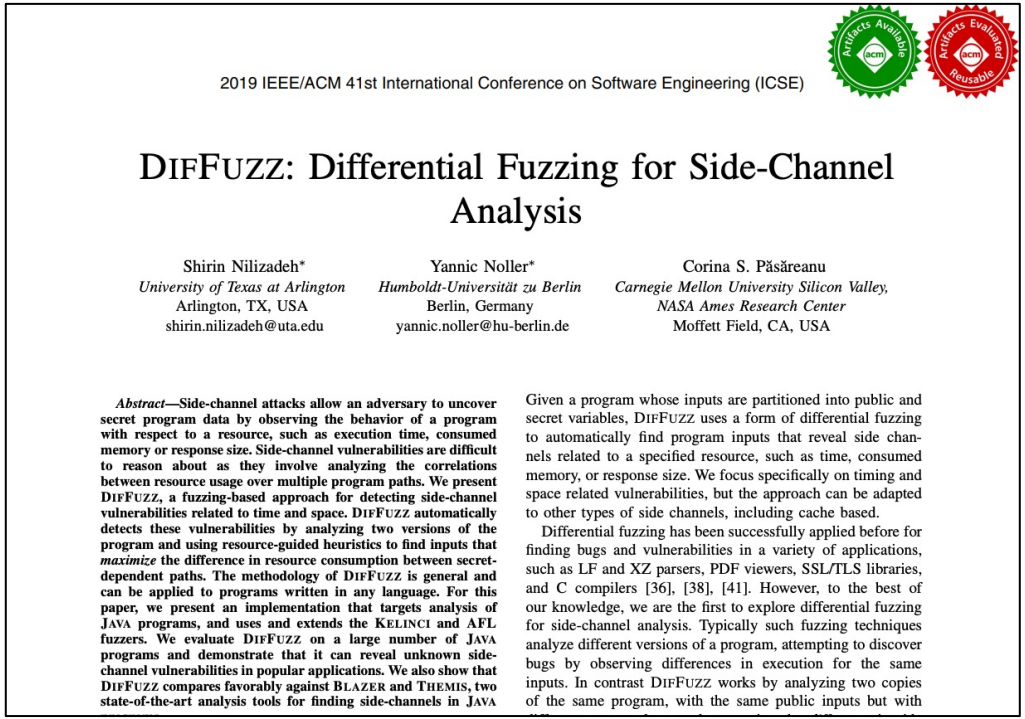


** initially motivated by the DARPA Space/Time Analysis for Cybersecurity (STAC) program*

Path to Side-Channel Repair



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



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.

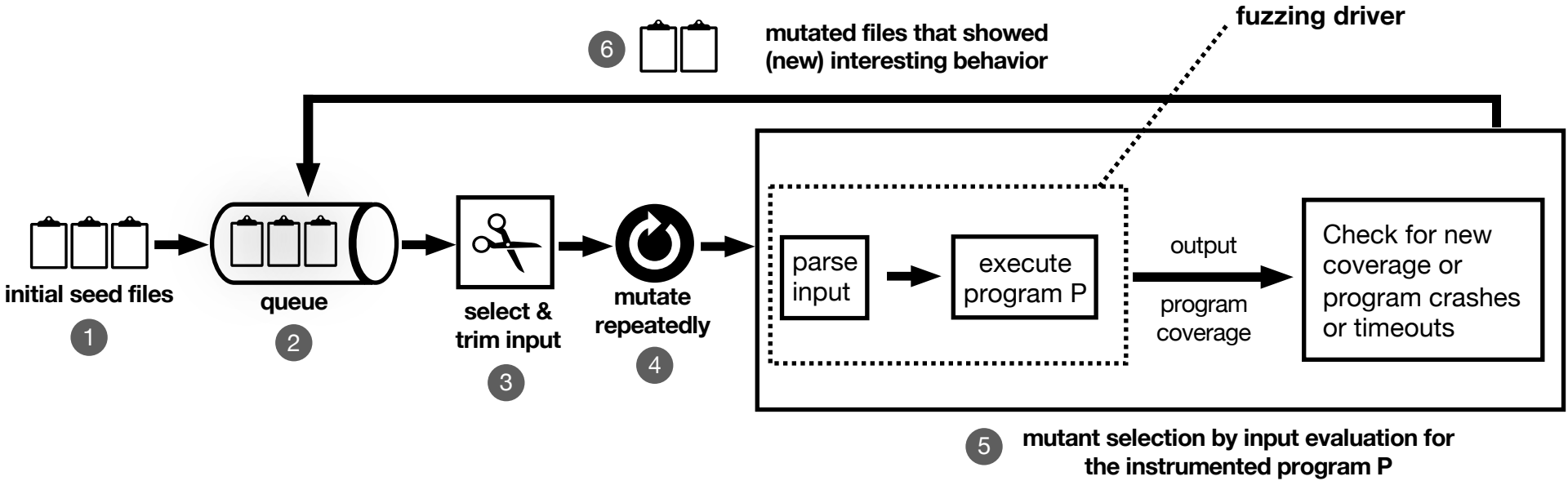
Differential fuzzing has been successfully applied before for finding bugs and vulnerabilities in a variety of applications, such as LF and XZ parsers, PDF viewers, SSL/TLS libraries, and C compilers [36], [38], [41]. However, to the best of our knowledge, we are the first to explore differential fuzzing 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

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

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

Greybox Fuzzing



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]

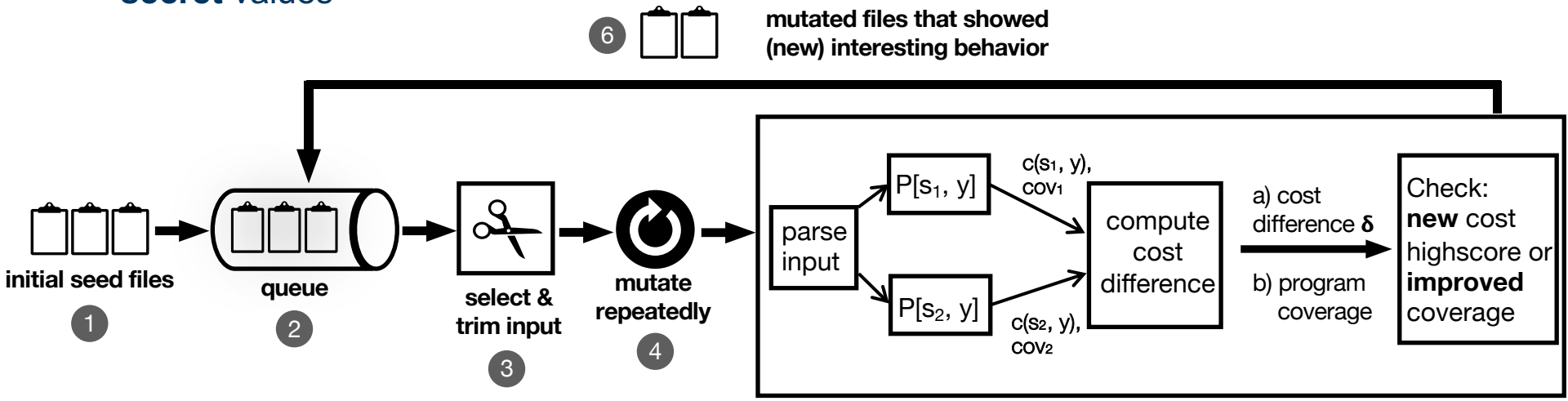
program execution	$P[pub, sec_1]$
cost observation	$c(P[pub, sec_1])$
two secret values	$c(P[pub, sec_1])$ $c(P[pub, sec_2])$
equivalence	$c(P[pub, sec_1]) = c(P[pub, sec_2])$

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

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

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

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



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

Example Results

```
0  boolean pwcheck_unsafe (byte[] pub, byte[] sec) {
1      if (pub.length != sec.length) {
2          return false;
3      }
4      for (int i = 0; i < pub.length; i++) {
5          if (pub[i] != sec[i]) {
6              return false;
7          }
8      }
9      return true;
10 }
```

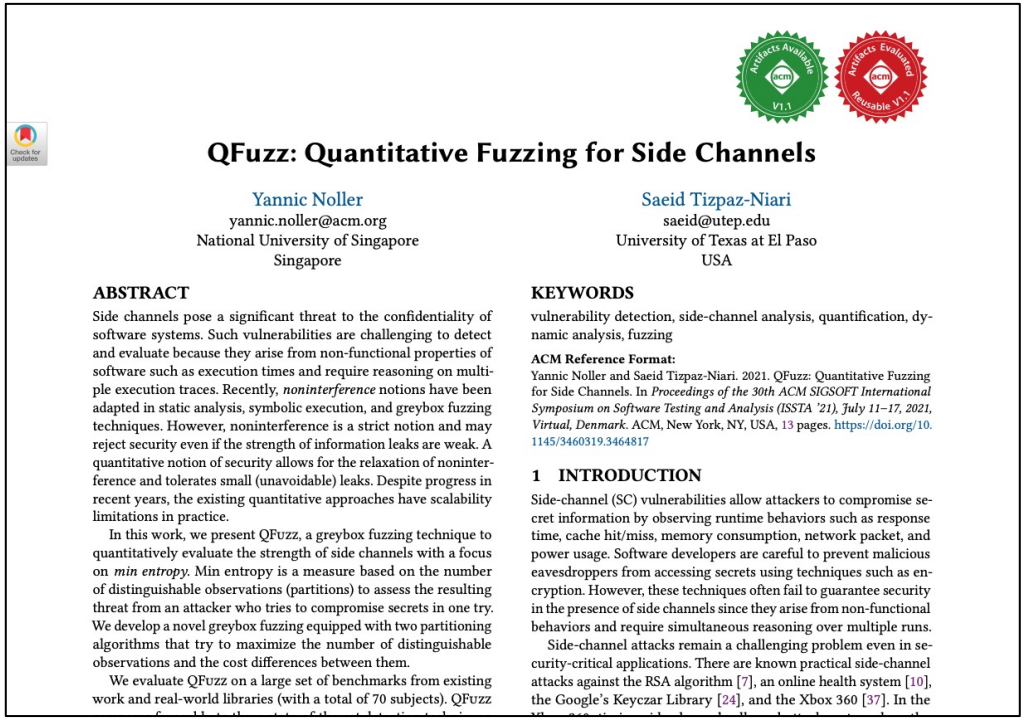
Initial Input: $cost_{Diff} = 0$

```
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]
```



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

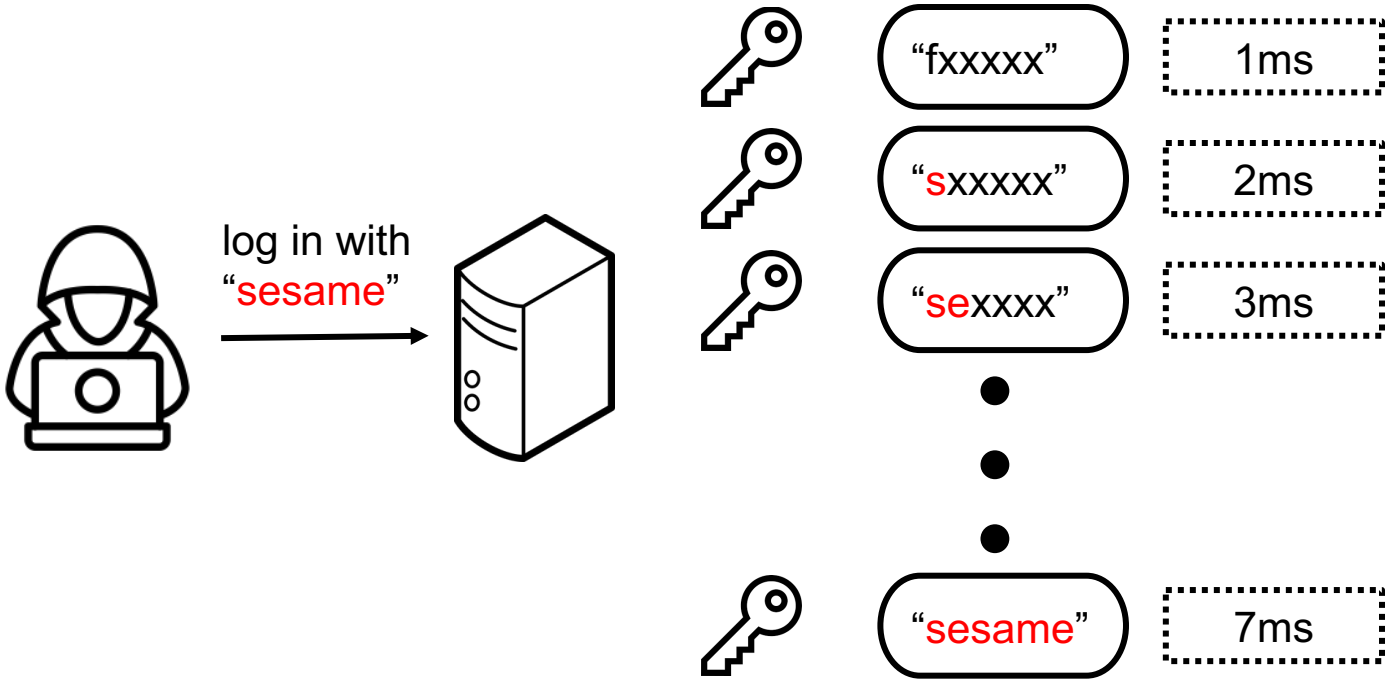
Side-channel (SC) vulnerabilities allow attackers to compromise secret information by observing runtime behaviors such as response time, cache hit/miss, memory consumption, network packet, and power usage. Software developers are careful to prevent malicious eavesdroppers from accessing secrets using 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

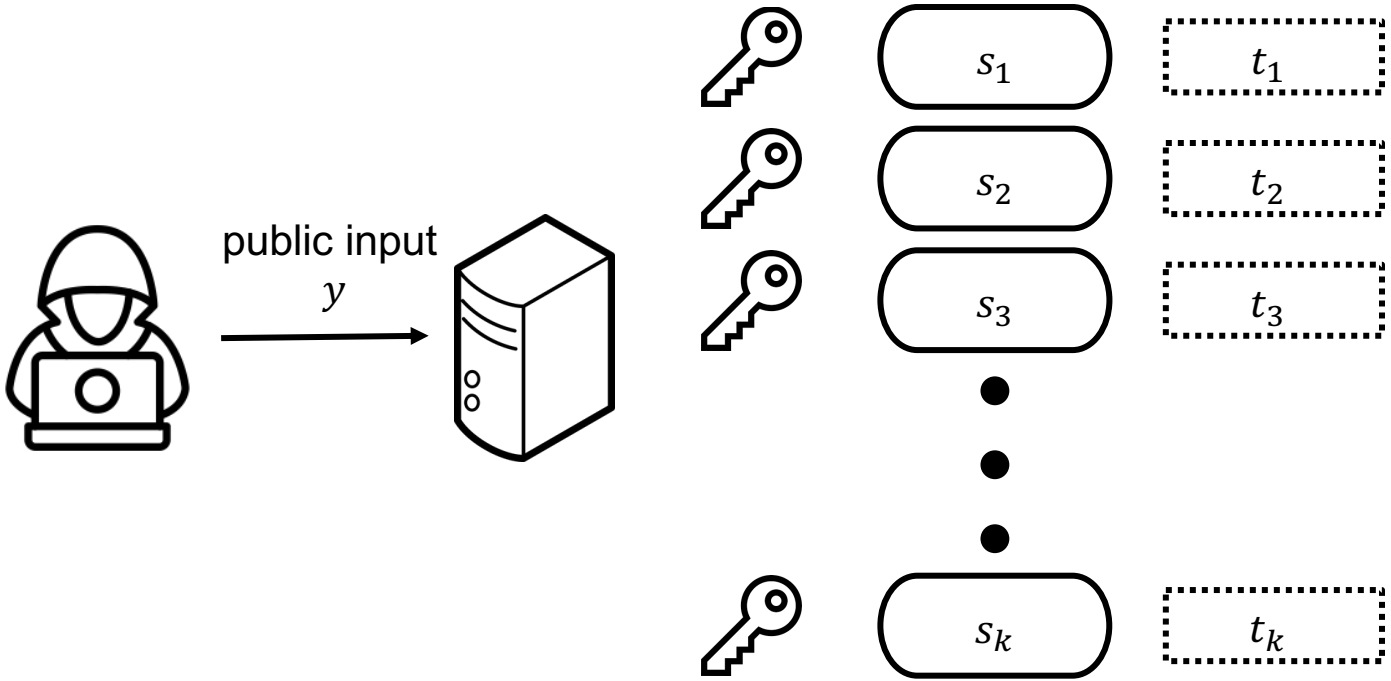
- uses **greybox fuzzing** to quantitatively evaluate the **strength** of side channels
- focuses on min entropy
- explores two **partitioning** algorithms that try to **maximize** the number of **distinguishable** observations
- cannot localize the vulnerability
- published at **ISSTA**'2021

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

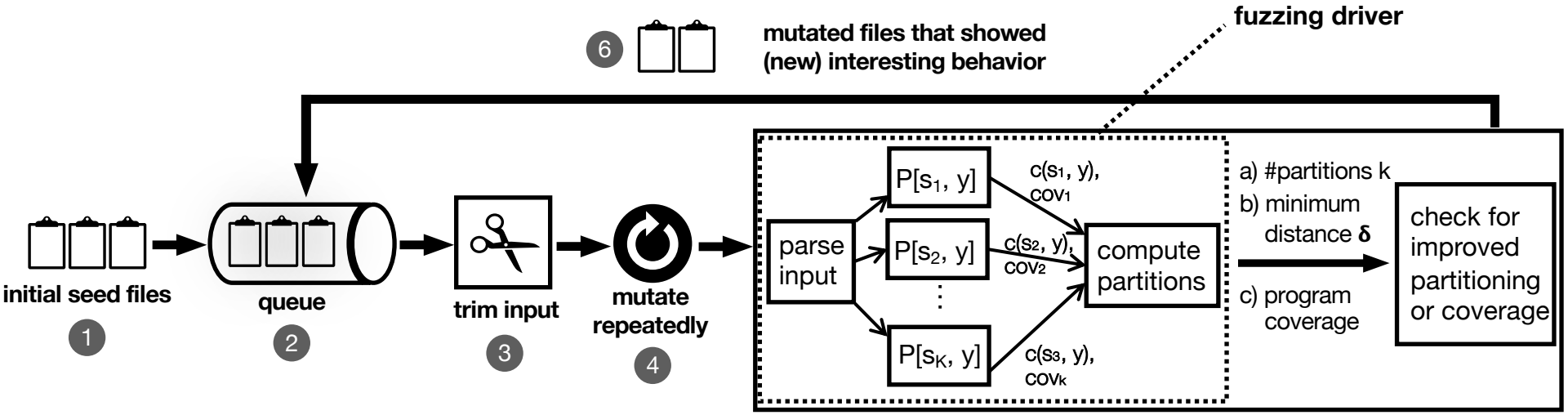
Timing SC Vulnerability: An Example



Timing SC Vulnerability: Quantification



QFuzz: Workflow



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

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;
}
```

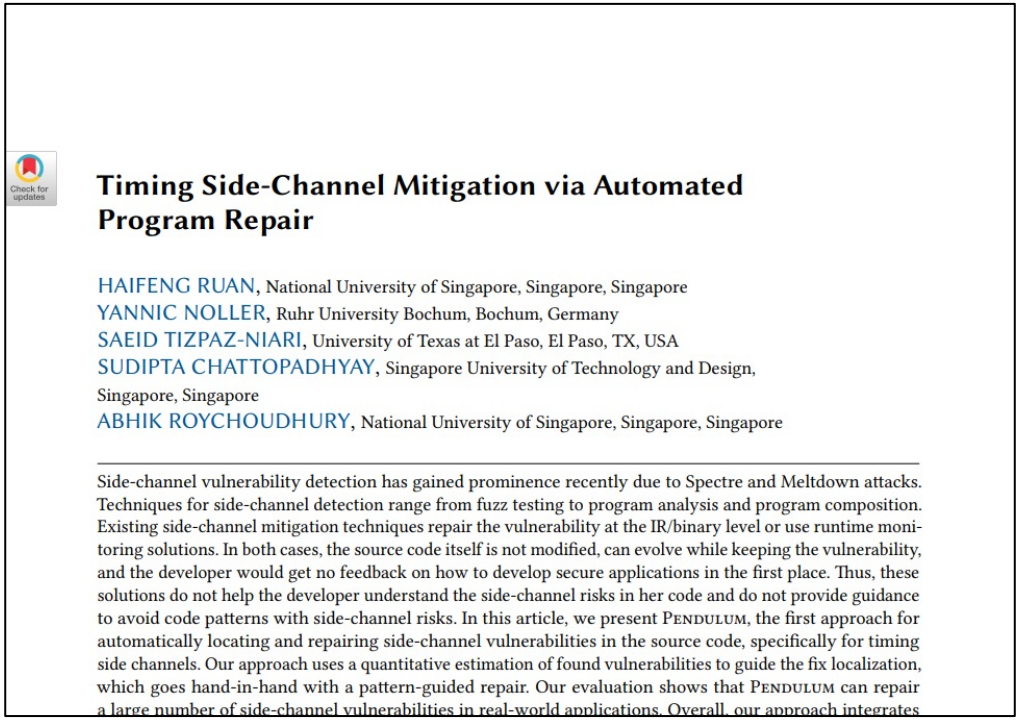
Equals (Unsafe Spring-Security)

```
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



Timing Side-Channel Mitigation via Automated Program Repair

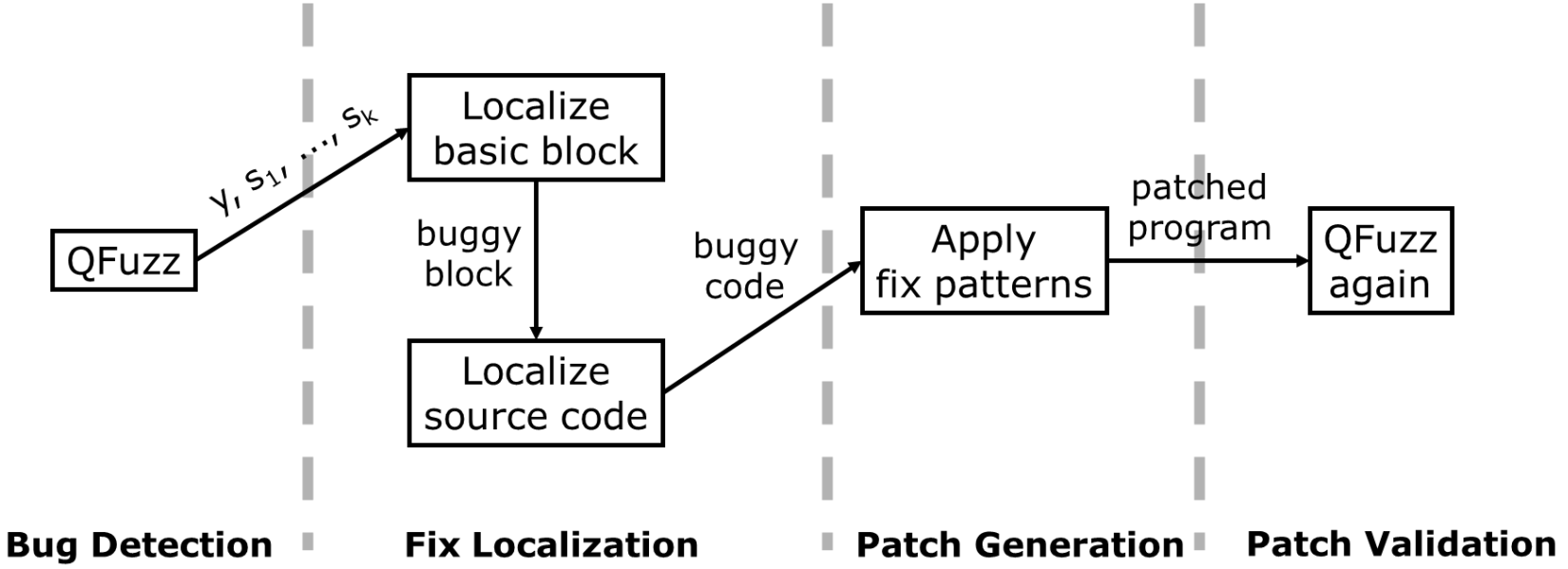
HAIFENG RUAN, National University of Singapore, Singapore, Singapore
YANNIC NOLLER, Ruhr University Bochum, Bochum, Germany
SAEID TIZPAZ-NIARI, University of Texas at El Paso, El Paso, TX, USA
SUDIPTA CHATTOPADHYAY, Singapore University of Technology and Design, Singapore, Singapore
ABHIK ROYCHOUDHURY,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

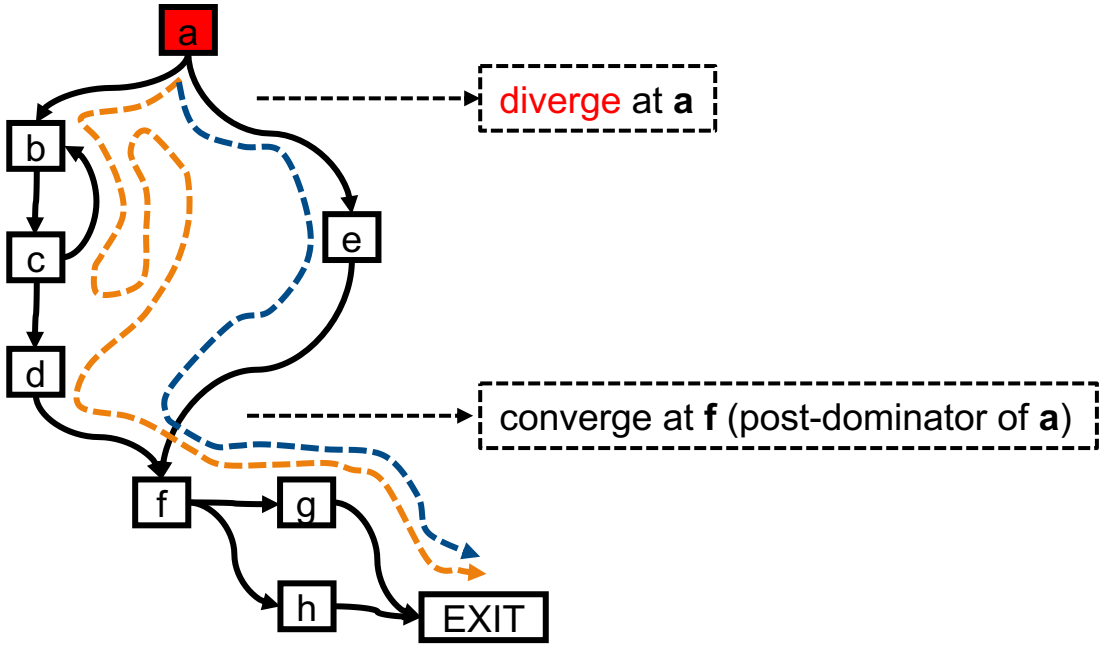
Pendulum – Repair Workflow (TOSEM'24)



Fix Localization (Basic Block)

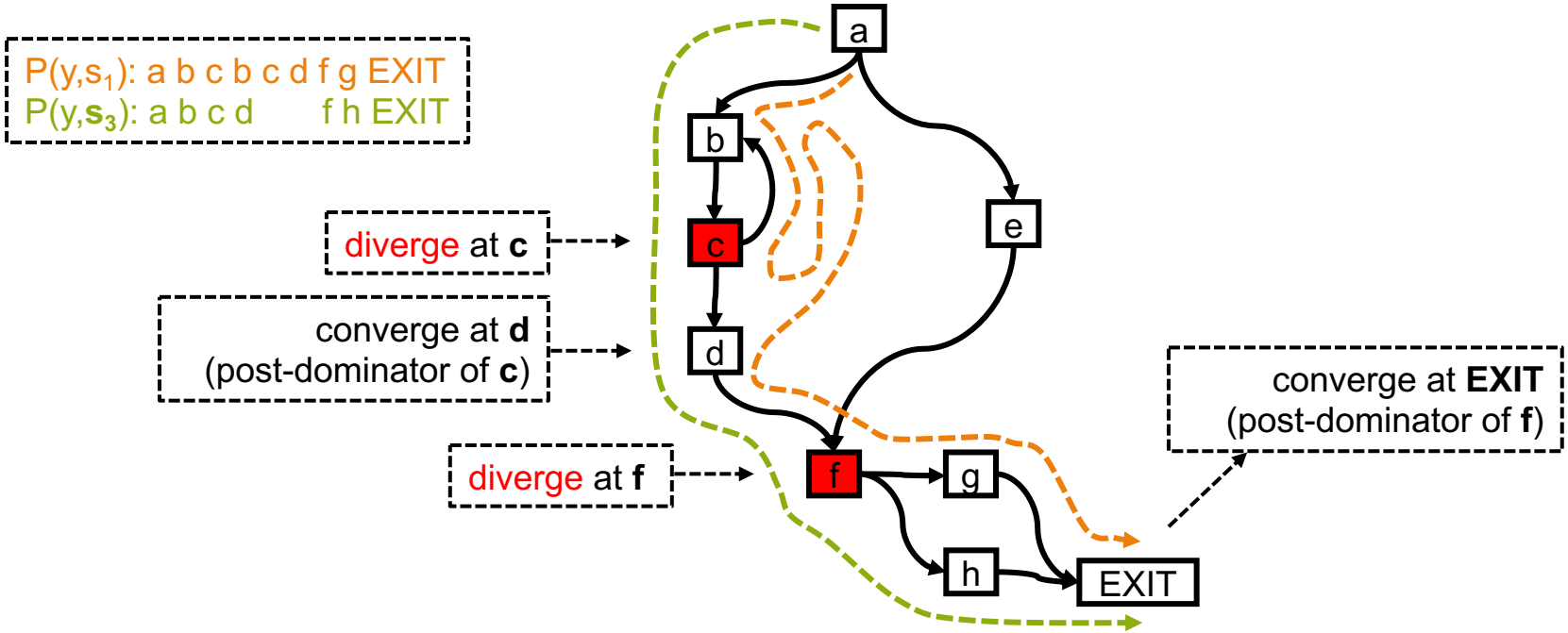
Compare traces to find where they diverge

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



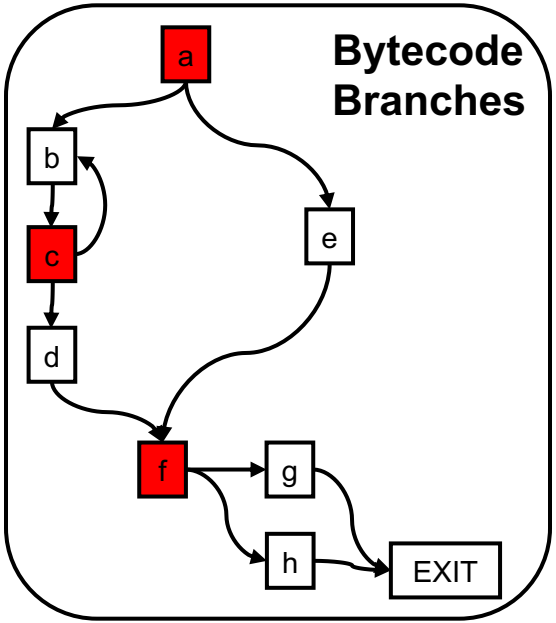
Fix Localization (Basic Block)

Compare traces to find where they diverge



Fix Localization (Source Code)

Map conditional branches to source code



Map

Debug Info

Source Code

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

Fix Patterns (Unsafe Operators)

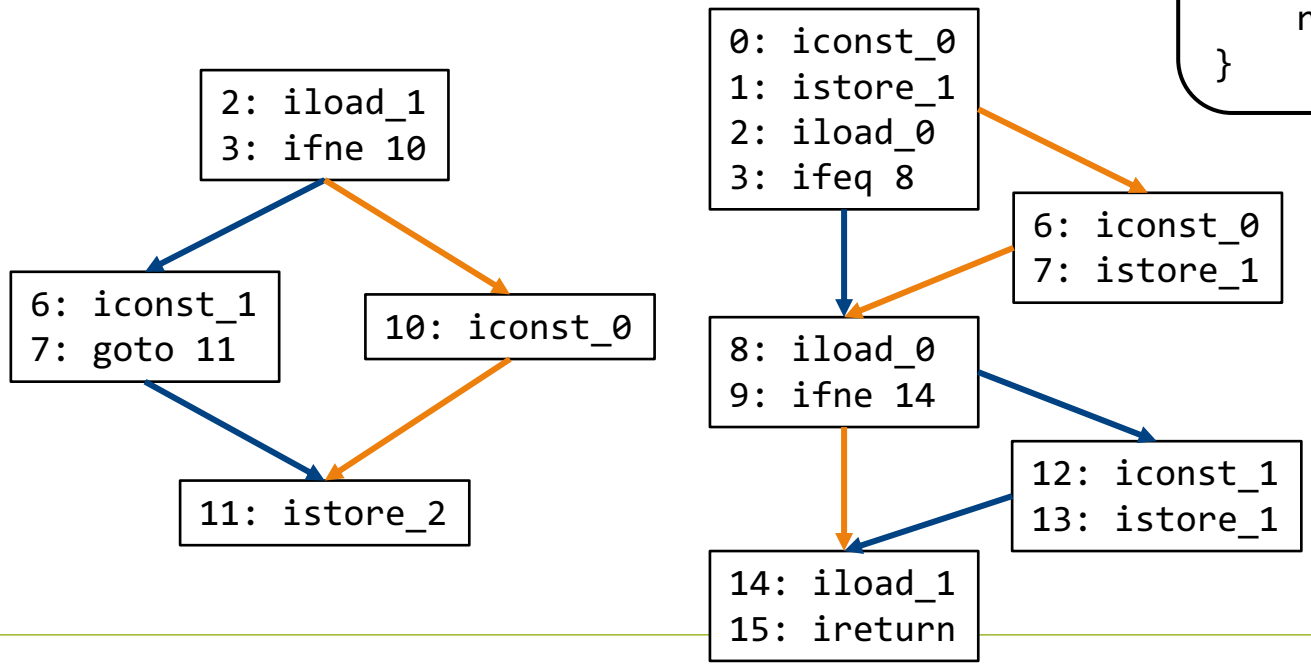
```
boolean b = !a;
```

➡

```
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)

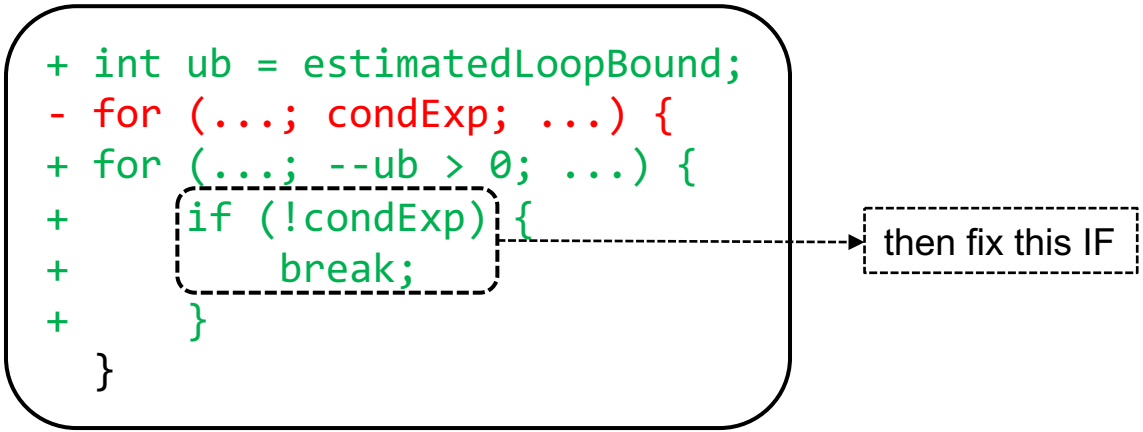
What if there is an early return / break / continue?

```
+ 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



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?

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

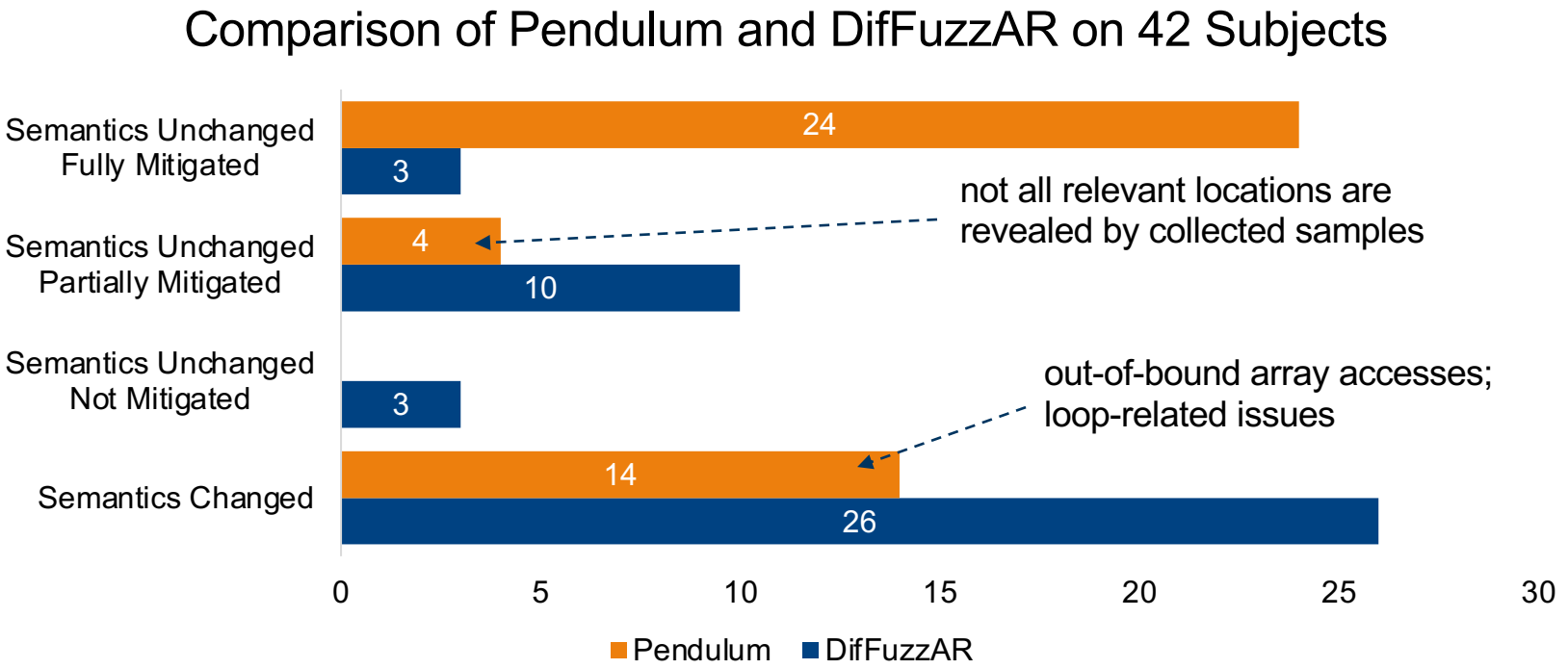
- we compare the identified fix locations with that of the developer fix for Pendulum and DifFuzzAR
- **Pendulum identifies the fix locations successfully for all 42 subjects**
- while **DifFuzzAR** fails for **13** subjects: limited fix localization supported

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

RQ3: Side Effects



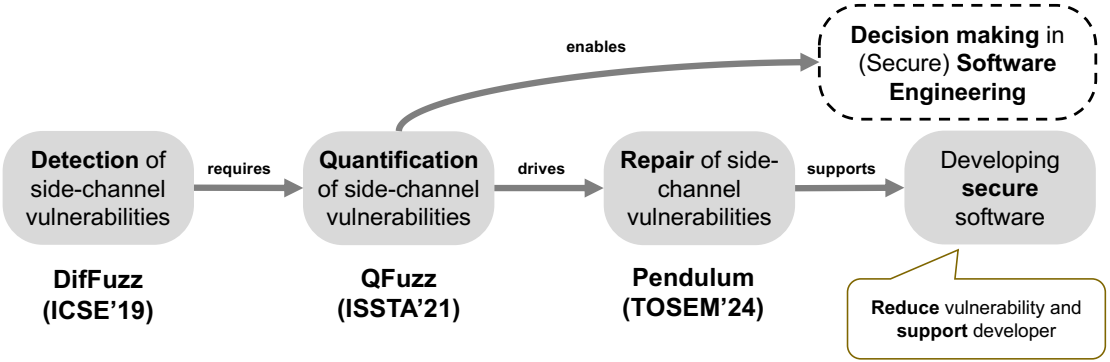
RQ4: Time and Space Impact

- 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**