Securing Software Systems via Fuzz Testing and Verification

CHEN Hongxu (G1502414K)
Supervisor: Prof LIU Yang
28/Nov/2019

Why Testing & Verification for Security?

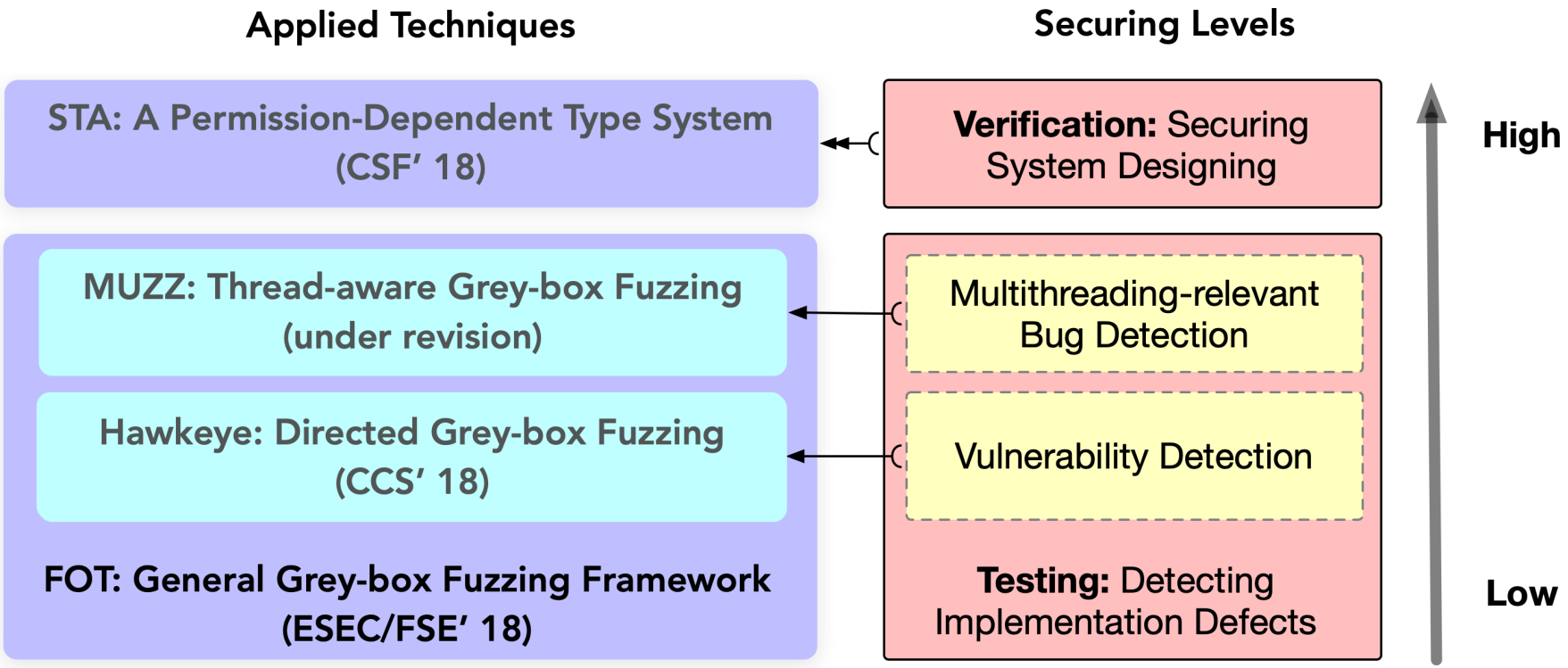
Testing

- Easy to conduct
- Relatively straightforward
- Can detect implementation vulnerabilities

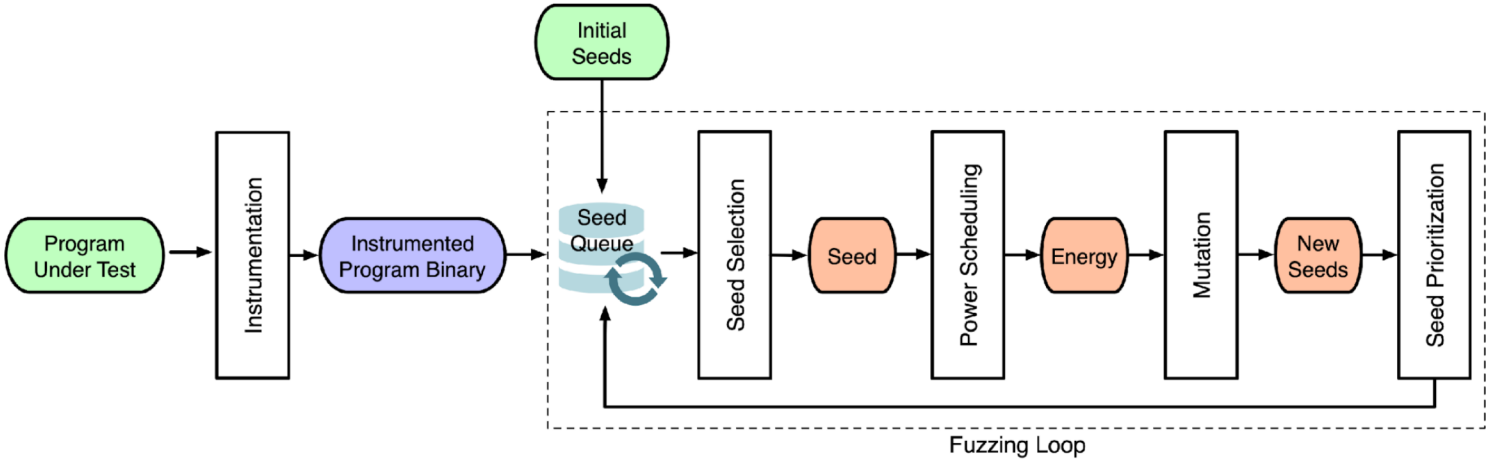
Verification

- Hard to conduct
- Usually require abstraction
- Can prove absence of vulnerabilities in various aspects

Thesis Contributions



Grey-box Fuzzing (1)

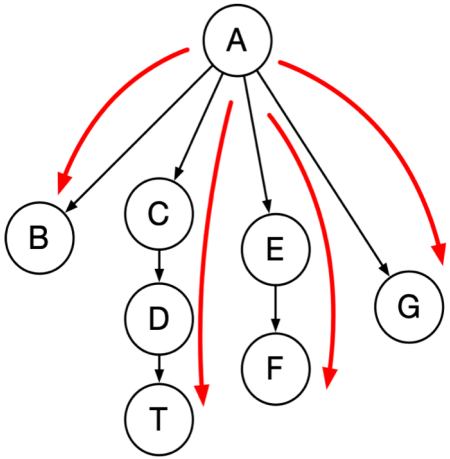


- Automatic Testing
- Program Feedback
- Mutation on Seeds

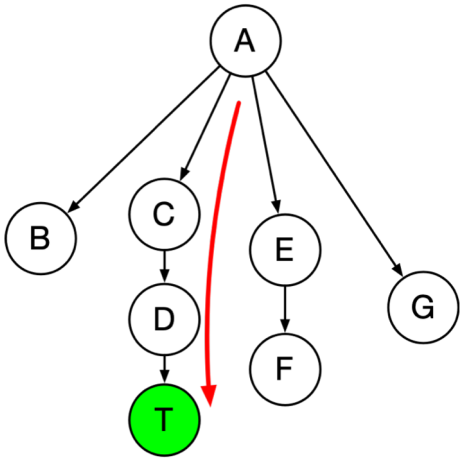
- ❑ 15,000+ vuls found by fuzzers
- ❑ 70+ papers in past 3 years

Grey-box Fuzzing (2)

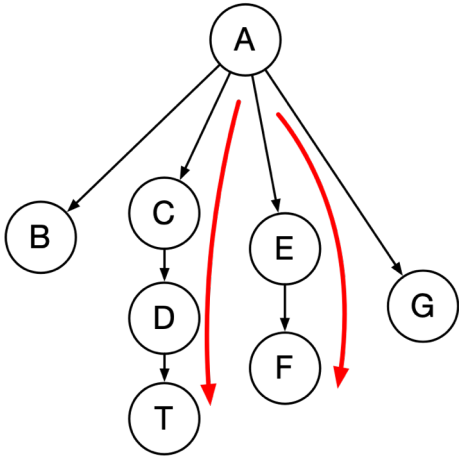
General-purpose Fuzzing



Directed Fuzzing



Specific-flow Fuzzing



Hawkeye: Directed Grey-box Fuzzing

Why Directed Grey-box Fuzzing (DGF)

- Test patches

```
diff --git a/bfd/dwarf2.c b/bfd/dwarf2.c
index 1566cd8..8abb3f0 100644 (file)
--- a/bfd/dwarf2.c
+++ b/bfd/dwarf2.c
@@ -1933,6 +1933,13 @@ read_formatted_entries (struct comp_unit *unit, bfd_byte **bufp,

   data_count = _bfd_safe_read_leb128 (abfd, buf, &bytes_read, FALSE, buf_end);
   buf += bytes_read;
+  if (format_count == 0 && data_count != 0)
+    {
+      _bfd_error_handler (_("Dwarf Error: Zero format count."));
+      bfd_set_error (bfd_error_bad_value);
+      return FALSE;
+    }
+
   for (datai = 0; datai < data_count; datai++)
     {
       bfd_byte *format = format_header_data;
```

- Reproduce crashes

CVE-2016-1835 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

Use-after-free vulnerability in the xmlSAX2AttributeNs function in libxml2 before 2.9.4, as used in Apple iOS before 9.3.2 and OS X before 10.11.5, allows remote attackers to cause a denial of service via a crafted XML document.

Source: MITRE
Description Last Modified: 07/27/2016
View Analysis Description

- Justify vulnerabilities

Project Name	CID	Checker	Category
wazuh/ossec-wazuh	117766	USE_AFTER_FREE	Memory - illegal accesses

File: /wazuh_modules/wmodules.c

```
< 4. Condition "cur_module", taking true branch
57        for (cur_module = wmodules; cur_module; wmodules = next_module) {
<<< CID 117766: Memory - illegal accesses USE_AFTER_FREE
<<< 5. Dereferencing freed pointer "cur_module".
58            next_module = cur_module->next;
59            cur_module->context->destroy(cur_module->data);
<< 2. "free" frees "cur_module".
60            free(cur_module);
< 3. Jumping back to the beginning of the loop
```

- ...

Motivating Example

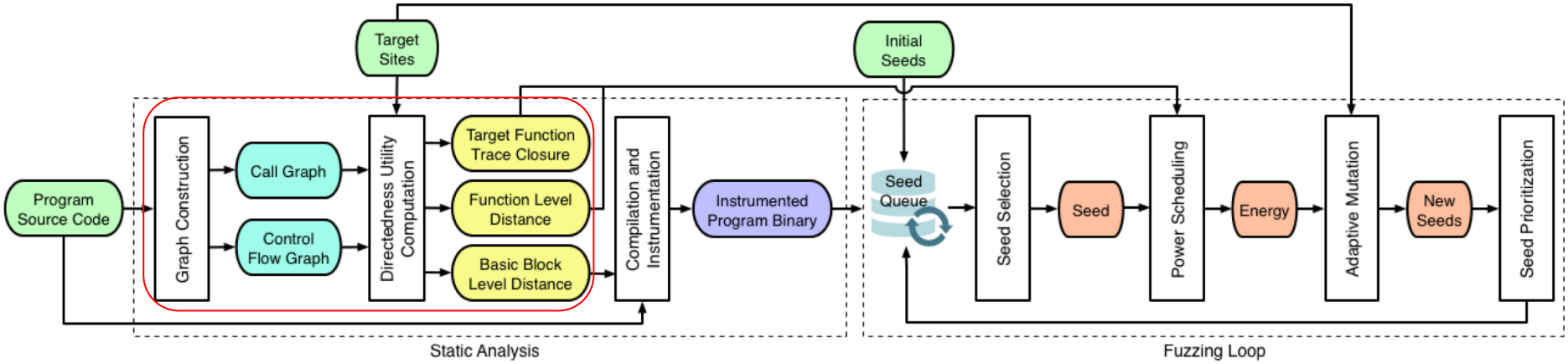
- Given a patch for GNU Binutils nm CVE-2017-15023
- Target T is **dwarf2.c:1601** in ***concat_filename***
- traces reachable to T
  - M → … a → b → c → d → T
  - M→ … a → e → T

Functions in a Crashing Trace	File & Line	Symbol
main	nm.c :1794	M
…	…	…
_bfd_dwarf2_find_nearest_line	dwarf2.c :4798	a
comp_unit_find_line	dwarf2.c :3686	b
comp_unit_maybe_decode_line_info	dwarf2.c :3651	c
decode_line_info	dwarf2.c :2265	d
concat_filename	dwarf2.c :1601	T
…	…	Z
Functions in a Normal Trace	**File & Line**	**Symbol**
main	nm.c :1794	M
…	…	…
_bfd_dwarf2_find_nearest_line	dwarf2.c :4798	a
scan_unit_for_symbols	dwarf2.c :3211	e
concat_filename	dwarf2.c :1601	T
…	…	Z

Desired Properties for DGF

- P1: Provide a distance metric avoiding bias to traces reachable to targets
  - All traces reachable to the target should be considered
- P2: Balance cost-effectiveness between static analysis and dynamic analysis
  - static analysis precision's impacts on dynamic fuzzing
- P3: Prioritize proper seeds and schedule mutations
  - deprioritize seeds with less chances to reach target locations
  - schedule more powers on "good" seeds
- P4: Apply adaptive mutations to increase mutators' effectiveness
  - fine-grained mutations for traces close to target locations

Workflow of Hawkeye



Power Function

$$p(s, T_b) = c_s(s, T_f) \cdot (1 - \tilde{d}_s(s, T_b))$$

- C_s favors longer traces that share more executed functions with the “expected” traces
- d_s favors shorter traces that reach the expected targets
- Used directly for power scheduling

Adaptive Mutation

When a seed has reached target functions, prefer fine-grained mutations

- Fine-grained: bit/byte level flips, add/sub on bytes/words, replace with interesting values
- Coarse-grained: random chunk modifications, semantic mutations, cross-over

Seed Prioritization

A *three-tier* queue to differentiate seed priorities and favor those:

a. cover new edges
b. are close to targets
c. or reach target function(s)

Hawkeye’s Solutions

- P1: Combine two separate competing metrics for power function to avoid bias
- P2: Apply precise graph construction to generate cost-effective directedness utilities for dynamic fuzzing
- P3: Apply target-favored seed prioritization and mutation power scheduling
- P4: Apply granularity adaptive mutation based on reachability to targets

Experimental Results

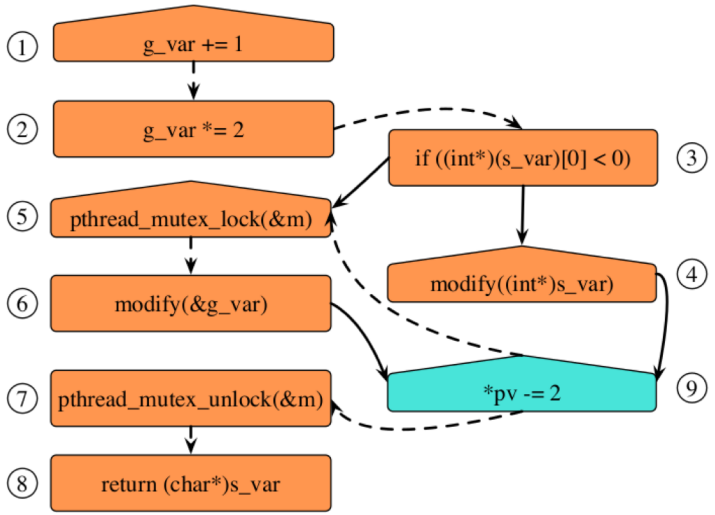
- Hawkeye reaches target sites and reproduce crashes much faster than AFL/AFLGo
  - e.g., vulnerabilities time-to-exposure reduced from 3.5h to 0.5h

- 41 zero-day vulnerabilities in Oniguruma, MJS, and Espruino
  - Among which 15 CVE IDs have been assigned

MUZZ: Thread-aware Grey-box Fuzzing

Motivating Example

```
1  int g_var = -1;
2  void modify(int *pv) { *pv -= 2;}                    // ⑨
3
4  void check(char * buf) {
5    if (is_invalid(buf)) { exit(1); }
6    else { modify((int*)buf); }
7  }
8
9  char* compute(void *s_var) {
10   g_var += 1;                                        // ①
11   g_var *= 2;                                        // ②
12   if ((int*)s_var[0]<0)                              // ③
13     modify((int*)s_var);                             // ④
14   pthread_mutex_lock(&m);                            // ⑤
15   modify(&g_var);                                    // ⑥
16   pthread_mutex_unlock(&m);                          // ⑦
17   return (char*)s_var;                               // ⑧
18 }
19
20 int main(int argc, char **argv) {
21   char * buf = read_file_content(argv[1]);
22   check(buf);
23   pthread_t T1, T2;
24   pthread_create(T1,NULL,compute,buf);
25   pthread_create(T2,NULL,compute,buf+128);
26   ......
27 }
```



- Not always easy to pass “check”
- Not enough coverage feedback caused by thread-interleaving

Existing Fuzzing Issues

Shared variable: g_var

g_var = -1

𝟙: “g_var+=1”

𝟚 : “g_var*=2”

Two threads: T1, T2

Interleaving Sequences	g_var
T1:𝟙 → T2:𝟙 → T2:𝟚 → T1:𝟚	4
T1:𝟙 → T2:𝟙 → T1:𝟚 → T2:𝟚	4
T1:𝟙 → T1:𝟚 → T2:𝟙 → T2:𝟚	2

Existing fuzzers:

- are only aware of instruction: 𝟙: “g_var+=1”
- only know a transition: 𝟙 → 𝟙

MUZZ's Improvements

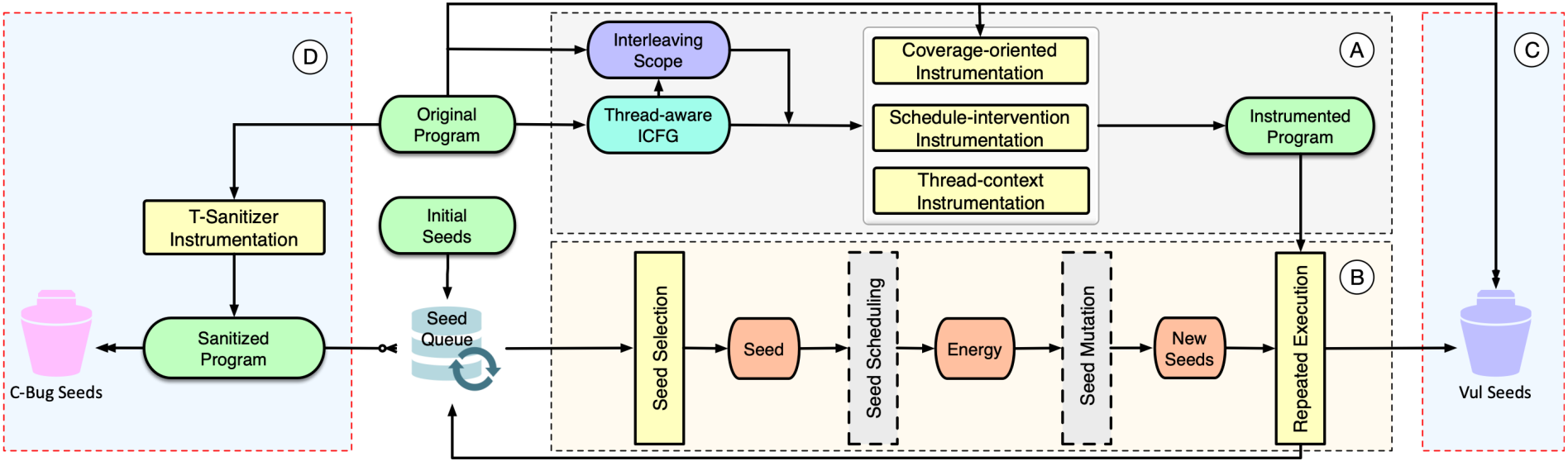
Feedback to track thread-interleavings and thread-context

- Coverage-oriented Instrumentation
  - assigns more deputies to multithreading-relevant segments
  - to add thread-interleaving feedback
- Thread-context instrumentation
  - distinguishes thread identities
  - to further differentiate threads

Schedule-intervention across executions

- Schedule-intervention instrumentation
  - adjusts each thread's priority
  - to increase the execution diversity to cover more thread-interleavings

Workflow of MUZZ



Part A: Static Analysis

A. Construct thread-aware ICFG
   - TFork, TJoin, TLock, TUnlock, TShareVar
B. Extract suspicious interleaving scope L_m based on ICFG
   - instructions belonging to L_m are likely to interleave with others
C. Apply coverage-oriented instrumentation based on L_m & McCabe complexity
   - McCabe complexity helps determine instrumentation ratios
D. Apply thread-context instrumentation based on threading functions
   - TFork/TJoin, TLock/TUnLock
E. Apply schedule-intervention instrumentation based on thread-forking routines
   - TFork

Part B: Dynamic Fuzzing

A. Seed Selection
   - prioritize to select seeds with
     - new coverage (including those caused by thread-interleavings)
     - new thread-context
   - to focus on feedback provided by multithreading-relevant traces

B. Repeated Execution
   - Repeat more times for executions with non-deterministic behaviors
   - to focus on behaviors probably caused by thread-interleavings

Part C & D: Vulnerability/Bug Detection

Detect vulnerabilities during fuzzing

- “crashes” during fuzzing
- manually triage based on the root causes
  - multithreading-relevant V_m
  - other vulnerabilities

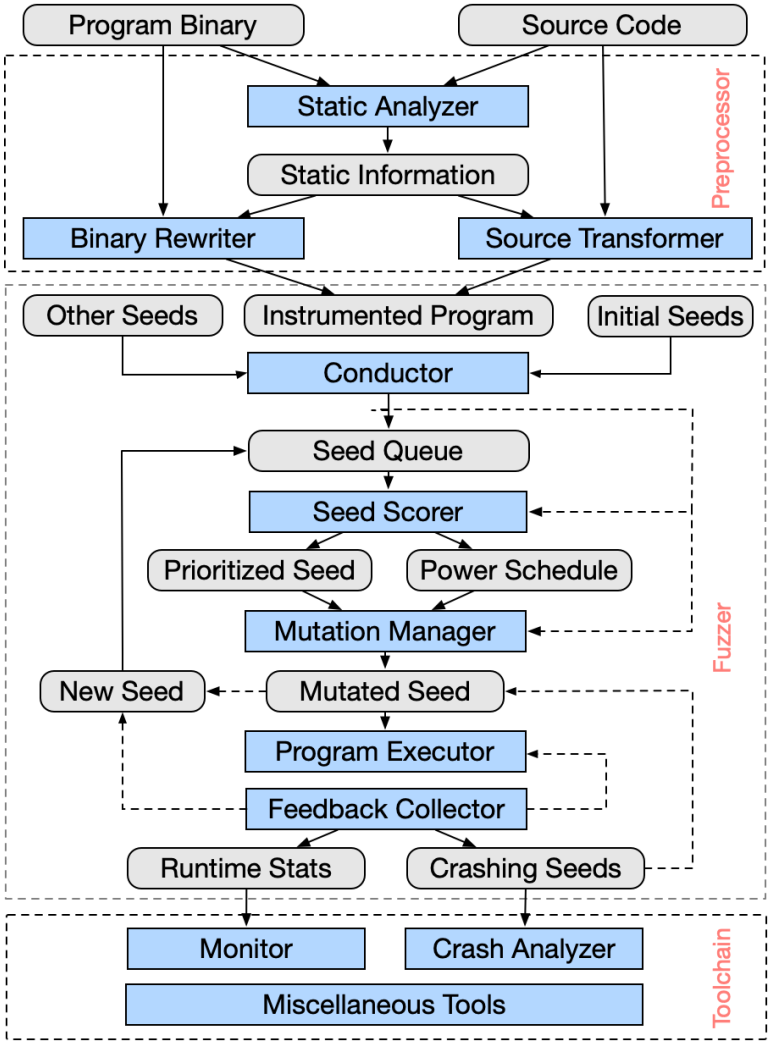
Reveal concurrency-bugs during replaying

- with help of ThreadSanitizer
- replay with a time-budget
- manually triage based on root causes
  - data-race
  - lock-order-inversion
  - …

Experimental Results

- MUZZ outperforms thread-unaware fuzzers in both multithreading-relevant seed generation and concurrency-vulnerability detection
- MUZZ reveals more concurrency-bugs by replaying the target programs against the generated seeds
- 8 new multithreading-relevant zero-day vulnerabilities
- 19 new concurrency-bugs

FOT: General Grey-box Fuzzing Framework



FOT: Fuzzing Orchestration Toolkit

- Versatility
- Configurability
- Extensibility

Components:

- Preprocessor
- Fuzzer
- Toolchain

Comparison with Other Frameworks

Framework / Features	AFL	libFuzzer	honggfuzz	FOT
Binary-Fuzzing Support	●	○	●	●
Multi-threading Mode	○	●	●	●
In-memory Fuzzing	●	●	●	●
Advanced Configuration	○	◐	○	●
Modularized Functionality	○	◐	○	●
Structure-aware Mutation	○	○	○	◐
Interoperability	○	○	○	◐
Toolchain Support	●	○	○	●
Precise Crash Analysis	○	○	●	●
Runtime Visualization	◐	○	○	●

Trophies

300+ vulnerabilities in 120+ projects

- GNU libc
- FFmpeg
- ImageMagick
- ...

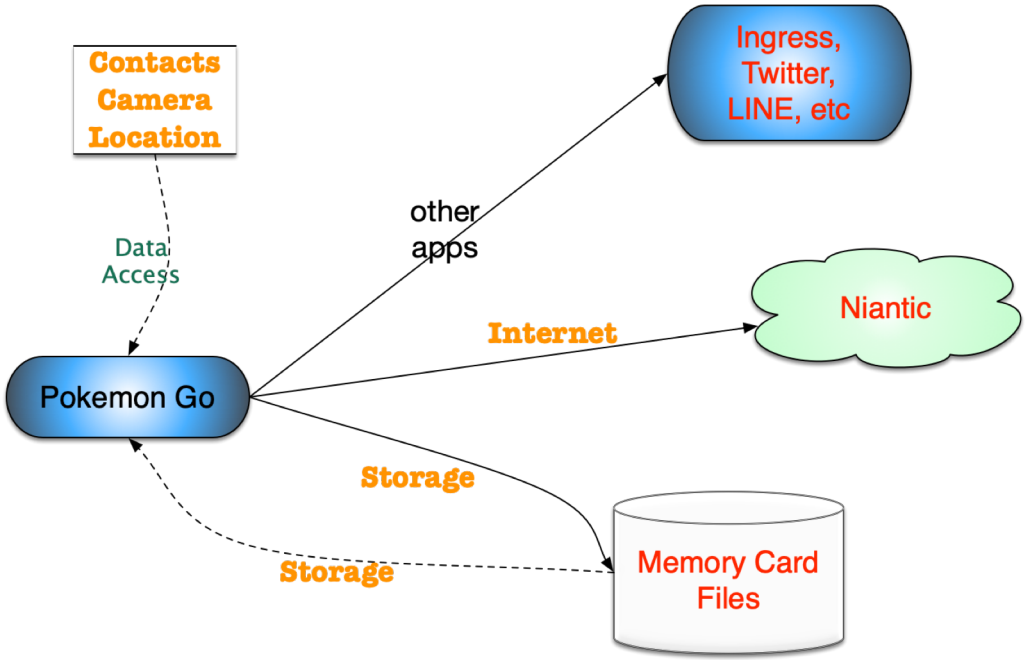
https: //github.com/ntu-sec/pocs

61 CVEs, 10 with critical/high CVSS3.x severity

1) CVE-2019-9169 (critical, 9.8)
2) CVE-2018-17293 (high, 8.8)
3) CVE-2019-15140 (high, 8.8)
4) CVE-2018-20819 (high, 7.8)
5) CVE-2018-11595 (high, 7.8)
6) CVE-2018-20796 (high, 7.5)
7) CVE-2018-15822 (high, 7.5)
8) CVE-2009-5155 (high, 7.5)
9) CVE-2018-11598 (high, 7.1)
10) CVE-2018-11593 (high, 7.1)

STA: Permission-Dependent Security Type System

Information flow in Android



How to prove no information leakage?

Testing ☹

Verification 😉

Type-Based Verification Procedures

1) Annotate variables with security types (labels)
2) Provide (security) typing rules for expressions, commands, etc
3) Prove soundness of the type system against non-interference

---

1) Check whether the given program can be typed with typing rules
2) If the program is typable, it assures there is no information leakage

Existing Type System Issues

```
A.f(String name) { // L -> H
  String res; // H
  if(checkPermission(CONTACT)){
    res := info(name); // H
  } else {
    res := ""; // H
  }
  return res; // H
}

B.h() { // B: ∅
  String r; // L
  r := A.f("Bob");
}
```

$$\frac{\vdash e : t \qquad \vdash c_1 : t \qquad \vdash c_2 : t}{\vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : t}$$

- NO Leakage
- NOT Typable ☹

The Language of STA

Expressions:

$$e ::= n \mid x \mid e \ \textbf{op} \ e$$

Commands:

$$\begin{aligned} c \ ::= \ & x := e \\ \mid \ & \textbf{if } e \textbf{ then } c \textbf{ else } c \\ \mid \ & \textbf{while } e \textbf{ do } c \\ \mid \ & c; c \\ \mid \ & \textbf{letvar } x = e \textbf{ in } c \\ \mid \ & \textbf{test}(p) \ c \textbf{ else } c \qquad \text{(checkPermission)} \\ \mid \ & x := \textbf{call } A.f(\overline{e}) \qquad \text{(IPC calls)} \end{aligned}$$

Function:

$$F ::= A.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\}$$

Type Promotion & Demotion

Definition

Given a permission p, the ***promotion*** and ***demotion*** of a base type t with respect to p are:

$$(t\uparrow_p)(P) = t(P \cup \{p\}), \forall P \in \mathscr{P} \qquad \text{(promotion)}$$

$$(t\downarrow_p)(P) = t(P \setminus \{p\}), \forall P \in \mathscr{P} \qquad \text{(demotion)}$$

1. $t\uparrow_p$ specifies scenarios where p is known to be granted;
2. $t\downarrow_p$ specifies scenarios where p is known to be absent.

Type Merging

Definition

Given a permission p and types t_1 and t_2, the ***merging*** of t_1 and t_2 along p, denoted as $t_1 \rhd_p t_2$, is:

$$(t_1 \rhd_p t_2)(P) = \begin{cases} t_1(P) & p \in P \\ t_2(P) & p \notin P \end{cases} \qquad \forall P \in \mathscr{P}$$

1. when p is present, $t_1 \rhd_p t_2$ equals t_1;
2. when p is absent, $t_1 \rhd_p t_2$ equals t_2.

Typing for test Command

```
A.f(String name) {
  int r;
  // if (checkPermission(p)) {
  test(p) {
    r := Info(p);
  } else {
    r := 0;
  }
  return r;
}
B.h() { // B: ∅
  String r_L;
  r := A.f("Bob");
}
```

According to **(T-CP)**,

$$\frac{\Gamma\uparrow_p; A \vdash c_1 : t_1 \qquad \Gamma\downarrow_p; A \vdash c_2 : t_2}{\Gamma; A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \rhd_p t_2}$$

if the calling APP has been granted with permission set P, after executing **test**, the security label of r is:

$$\Gamma(r)(P) = \begin{cases} H & p \in P \\ L & p \notin P \end{cases}$$

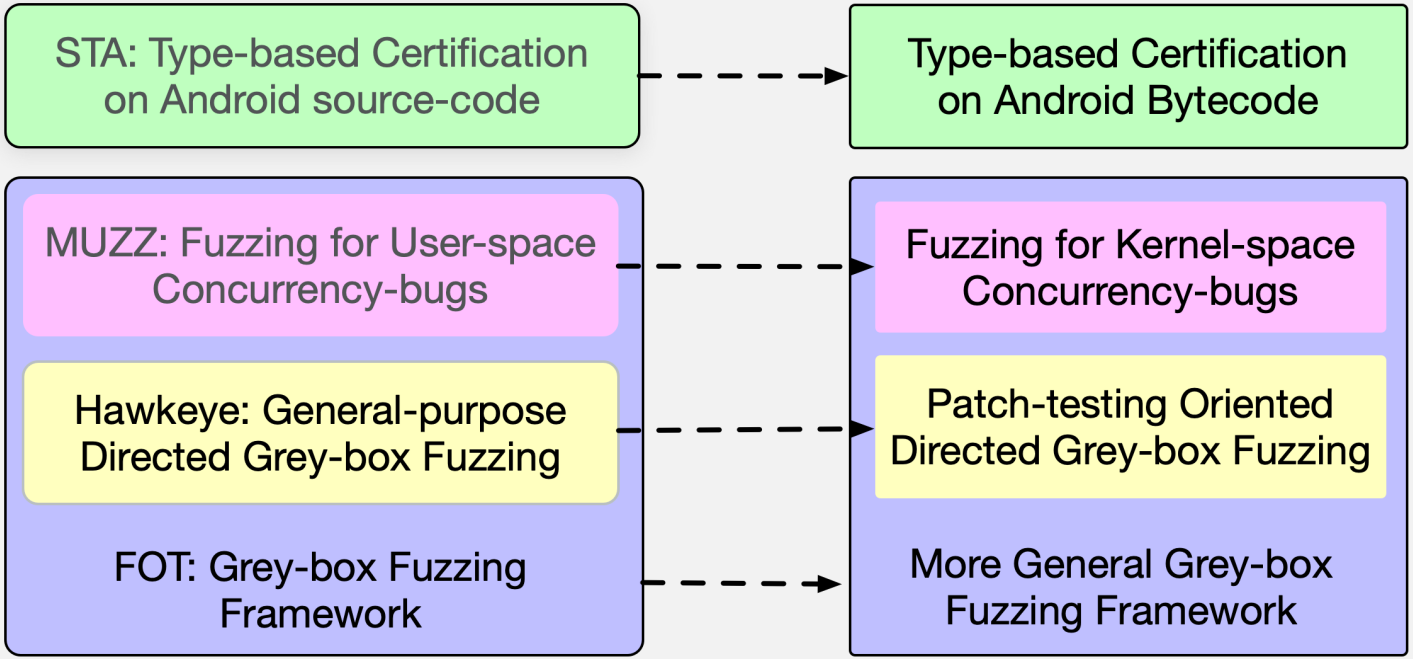
Since this is also the return type of function $A.f$, r := A.f("Bob") is typable when B does not have p.

Soundness Conclusion

Typable systems are non-interferent.

- typable: the target program can be checked with typing rules
- non-interferent: attackers know nothing about sensitive values

Summary & Future Work



Publications (thesis relevant)

- **Hongxu Chen**, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, Yang Liu, “Hawkeye: Towards a Desired Directed Greybox Fuzzer”, CCS’18
- **Hongxu Chen**, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, Yang Liu, “MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs”, (submitted).
- **Hongxu Chen**, Yuekang Li, Bihuan Chen, Yinxing Xue, Yang Liu, “FOT: A Versatile, Configurable, Extensible Fuzzing Framework”, ESEC/FSE ’18
- **Hongxu Chen**, Alwen Tiu, Zhiwu Xu, Yang Liu,“A Permission-Dependent Type System for Secure Information Flow Analysis”, CSF’18

Publications (others)

- Xiaofei Xie, **Hongxu Chen**, Yi Li, Ma Lei, Yang Liu, and Jianjun Zhao. Deephunter: A coverage-guided fuzzer for deep neural networks, ASE ’19.
- Yuekang Li, Yinxing Xue, **Hongxu Chen**, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection, ESEC/FSE’19
- Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, **Hongxu Chen**, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique, TIFS’17
- Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and **Hongxu Chen**. S-looper: automatic sum- marization for multipath string loops, ISSTA’15

Thank you!