AFRL-RI-RS-TP-2023-001

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 5

TWO SIX LABS

MARCH 2023

TECHNICAL PAPER

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency (DARPA) Public Release Center and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TP-2023-001 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
CHAD C. DESTEFANO
Work Unit Manager

/ S /
JAMES S. PERRETTA
Deputy Chief
Information Warfare Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
MARCH 2023	TECHNICAL PAPER	JANUARY 2021	JUNE 2021

4. TITLE AND SUBTITLE

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 5

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-19-C-0009	N/A	DoD, DARPA
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
		R2PB

6. AUTHOR(S)

Jared Ziegler, Will Huiras, & Irwin Ong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Two Six Labs
901 N Stuart Street, Suite 1000
Arlington VA 22203

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

DARPA/I20
675 North Randolph Street
Arlington VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RI, DARPA I20

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-RI-RS-TP-2023-001

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 36386
Date Cleared: 6/13/2022

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This Edge of the Art report is part of a series that aggregates the most recent advances in vulnerability research (VR), reverse engineering (RE), and program analysis tools and techniques.

15. SUBJECT TERMS

Vulnerability Research, Reverse Engineering, Program Analysis, Cyber, Fuzzing, Software Security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	94

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
CHAD DESTEFANO	N/A

Edge of the Art in Vulnerability Research

DARPA CHESS Program

VERSION 5.0
JUNE 2021
ACKNOWLEDGEMENT

Authors:
Jared Ziegler, Will Huiras, & Irwin Ong

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-19-C-0009.

DISCLAIMER

901 N Stuart Street, Suite 1000
Arlington, VA 22203
(703) 543-9662
info@twosixtech.com
www.twosixtech.com

Contents

Introduction

The DARPA CHESS program seeks to increase the speed and efficiency of software vulnerability discovery and remediation by integrating human knowledge into the automated vulnerability discovery process of current and next generation Cyber Reasoning Systems (CRS). As with most technological advancements that seek to supplant what was once the exclusive domain of human expertise, the best and the most convincing way to measure success is against a human baseline.

Combining Hacker Expertise Can Krush Machine Assisted Target Exploitation (CHECKMATE), the CHESS Technical Area 4 (TA4) control team, focuses on providing the CHESS program with a team of expert hackers with extensive domain experience as a consistent baseline to measure the TA1 and TA2 performers against.

Vulnerability research is a constantly evolving area of cyber security, making the baseline for measuring the success of the CHESS program a moving target. The control team must keep pace with the most recent advancements to remain an effective baseline for comparison. The CHECKMATE team not only needs to stay on top of the state-of-the-art research and technology solutions, but also capture key emerging and trending techniques across all relevant vulnerability classes, tools, and methodologies.

This Edge of the Art report is part of a series that aggregates the most recent advances in vulnerability research (VR), reverse engineering (RE), and program analysis tools and techniques that the CHECKMATE team considers when planning for the next CHESS evaluation event.

To stay on the Edge of the Art, a new edition of this report will be released every six months with enhancements in the current state-of-the-art and new tools and techniques emerging in the cyber security community.

1.1 Scope

The purpose of this Edge of the Art (EotA) report is to document tools and techniques that have come into existence (or significantly matured) since the last report.

The EotA reports are produced using an “aggregate and filter” approach. The CHECK-

MATE team constantly monitors many different sources in an attempt to aggregate all known and emerging tools and techniques. This information is then filtered into what the CHECKMATE team considers worth reporting. The definition of the “edge” is governed by the filter criteria, which differ across tools and techniques. It is anticipated that these criteria, and therefore the definition of “edge,” will evolve over the life of the CHESS program.

Naturally, this process is imperfect. Some tools or techniques may be overlooked during the writing of a particular report (potentially to be added in a later edition). Others that are included may turn out to be of diminished importance. All views expressed are those of the authors.

Additional information on the scope, organization, and criteria for the EotA report can be found in the appendix.

Static Analysis

Static analysis investigates a binary executable without running it. The most common forms of static analysis in reverse engineering and vulnerability research begin with disassembling and/or decompiling a binary executable. These transformations utilize several static program analysis techniques, which also underlie many of the other techniques discussed in this report. One of the most fundamental forms of static analysis is lifting a program to an intermediate representation (IR). IRs are used in many of the tools and techniques discussed throughout this report. Static analysis can be used for reverse engineering compiled programs, statically rewriting and instrumenting a binary executable, performing static vulnerability discovery on either source or binary code, etc.

A general overview of static analysis can be found in the appendix.

2.1 GEARSHIFT

Reference Link	https://github.com/grimm-co/GEARSHIFT
Target Type	Binary - Ghidra plugin
Host Operating System	Windows, MacOS, Linux
Target Operating System	N/A; any targets Ghidra supports
Host Architecture	Any host architecture Ghidra runs on
Target Architecture	Any target architecture Ghidra supports
Initial Release	17 May 2020
License Type	Open-Source (UIUC)
Maintenance	Maintained by GRIMM, last GitHub commit 22 Feb 2021

Overview

GEARSHIFT is a Ghidra plugin that performs structure recovery on a target binary using program analysis techniques. The user targets a function that has a structure as an argument, and the plugin will automatically generate a structure definition based on how the structure parameter is used. GEARSHIFT also generates C harness code that populates the function parameters using input from a file and calls the target function, which is useful for fuzzing code that takes structures as input.

Design and Implementation

GEARSHIFT attempts to recover structures by performing symbolic analysis over the data dependency graph from Ghidra's Intermediate Language (IL) [1]. Ghidra lifts operations

on all architectures into its own IL, called P-Code. Data dependency establishes that an instruction will affect the state of a register or memory value which will later be used by another instruction, so that the latter instruction ”depends” on the former. The combination of all dependencies is the data dependency graph, which Ghidra provides a view of via P-Code.

In order to recover structures, GEARSHIFT performs a variant of Value Set Analysis to discover accesses to locations that are at an offset into a struct. The basic idea is that by discovering the size and offset of accesses, one can infer the size and location of struct members (figure 2.1).

The plugin takes a function and its parameters as a starting point and performs a depth-first search of the data dependency graph, recording all loads and stores performed. The actual symbolic execution is performed by emulating the state of a function for all P-Code instructions it contains, starting with the function parameters as symbolic variables and collecting symbolic expressions in a binary expression tree.

In order to track struct usage across function boundaries, GEARSHIFT performs interprocedural analysis. This is important because structs are often accessed in multiple functions and some member accesses may only be discovered by interprocedural analysis. To do this, GEARSHIFT uses two types of analysis, forward (in order to both discover loads and stores in a single function) and backwards (to propagate symbolic definitions as they relate to function parameters).

```
void FUN_00100721(undefined4 *param_1)

{
  void *pvVar1;

  pvVar1 = malloc(8);
  *(void **)(param_1 + 6) = pvVar1;
  FUN_001006fa(param_1 + 4);
  FUN_001006fa(*(undefined8 *)(param_1 + 6));
  *param_1 = 2;
  *(undefined *)(param_1 + 1) = 7;
  *(undefined8 *)(param_1 + 2) = 8;
  return;
}
```

```
void FUN_00100721(S0 *param_1)

{
  S1 *pSVar1;

  pSVar1 = (S1 *)malloc(8);
  param_1->entry_6 = pSVar1;
  FUN_001006fa((S1 *)&param_1->entry_4);
  FUN_001006fa(param_1->entry_6);
  param_1->entry_0 = 2;
  param_1->entry_1 = '\a';
  param_1->entry_3 = 8;
  return;
}
```

Figure 2.1: Example of Improvement on Decompilation Readability

Once these analyses are performed GEARSHIFT can categorize loads as pointers to either a primitive, struct, or array; this information is used to categorize accesses to structs and recover the locations and sizes of struct members. The authors address the issue of differentiating structs and arrays by using accesses via indices inside of a loop as the sign of an array. Once struct interpolation is completed, GEARSHIFT defines the inferred structs inside Ghidra and automatically applies the new types to all uses of those structs.

In addition to struct discovery, GEARSHIFT generates a fuzzing harness that populates function parameters based on input. The generated harness is a template that is populated based on the function chosen as the target for GEARSHIFT and any structures it takes as input. This capability allows users to see the inferred structs in C as well as serving as a

good starting point for being able to fuzz a closed-source binary.

Use Cases and Limitations

GEARSHIFT's basic use case is struct recovery, which is a very common problem in reverse-engineering, especially in C++ code and firmware or operating system code. The use of multiple layers of structs is a challenge to reverse engineers and can be extremely tedious to analyze manually. GEARSHIFT's analysis starts from a function and its parameters and does interprocedural analysis in a way similar to how a human user would. Even in the case of imperfect recovery, an automated process that provides a partial solution can save the user a lot of time, especially in the case of large structures.

Automated struct recovery is a challenging topic, and GEARSHIFT focuses on solving the problem given a few assumptions. GEARSHIFT only attempts to recover structures that are passed via function parameters, not structs accessed via global variables or constructed within the function, which is a conscious design decision that requires the user to understand the usage pattern. Luckily Ghidra's "undo" feature makes it a simple thing to revert the analysis if the results are not as the user intended. This is illustrated in figure 2.2.

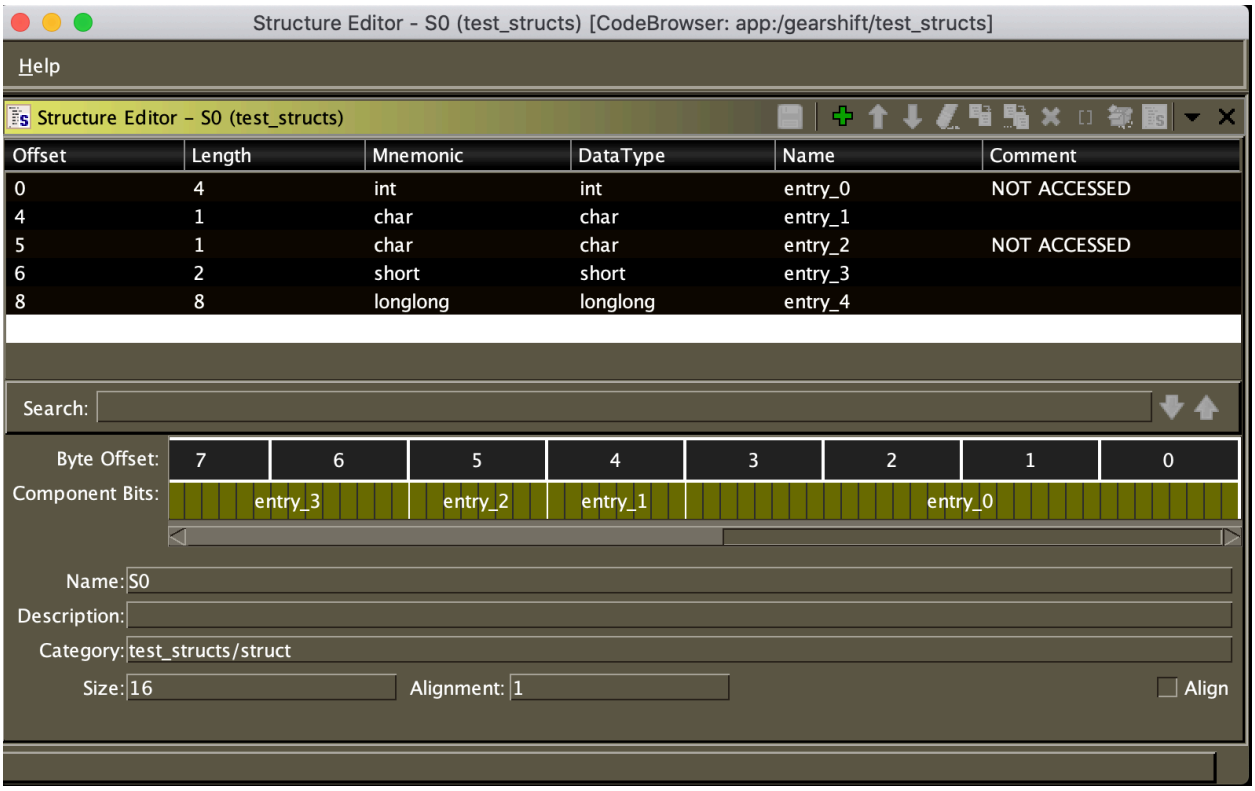

Figure 2.2: Example of Improvement on Decompilation Readability

If Ghidra's automatic analysis is not precise enough and inconsistencies hinder the analysis, GEARSHIFT will warn the user, who will have to adjust types manually. There are also

limits to what can be recovered about a structure given a particular section of code, so the user will still have to be engaged in understanding the target and applying structures where appropriate. Also, as with pretty much all symbolic analysis there are practical limits on the amount and complexity of analysis. There is still no silver bullet for reverse-engineering.

2.2 Ghidraaas

Reference Link	https://github.com/Cisco-Talos/Ghidraaas
Target Type	Binary
Host Operating System	Linux, Windows, macOS
Target Operating System	Linux
Host Architecture	Java Byte Code; Python3
Target Architecture	All architectures supported by Ghidra
Initial Release	09/2019
License Type	Apache-2.0 License
Maintenance	Stable release, last Github commit Dec 2020 by Cisco-Talos

Overview

As described on the project's Github page, "Ghidraaas is a simple web server that exposes Ghidra analysis through REST APIs."[2] It is designed to allow users to offload many generic reverse engineering tasks to a remote machine. Ghidraaas is also used as the back-end for GhIDA [3], which provides allows the functionality of Ghidra from within IDA Pro.[4] Ghidraaas provides s REST API that allows interfacing with an instance of Ghidra running on that server.

Design and Implementation

Ghidraaas is most simply described as a wrapper around Ghidra. It consists of a web server providing a REST API. The web server is written in Python 3. Ghidraaas uses

Ghidra Headless Analyzer to analyze the submitted sample submitted by the user. The tool automates some of the more routine tasks associated with binary analysis. Additional tasks can be added to the file `flask_api.py` by defining a new `@app.route()` and providing new scripts for Ghidra to run on a given binary. While the provided scripts are written in Python3, Ghidraaas can interface with Ghidra using any supported scripting language. (Ghidra supports Java and Python by default.) Out of the box, Ghidraaas provides the following APIs (figure 2.3):

Figure 2.3: Default APIs available with Ghidraaas [2]

Ghidraaas includes a `test.py` script that, with very little modification, can produce useful results. Figures 2.4 and 2.5 show the output of two such modifications.

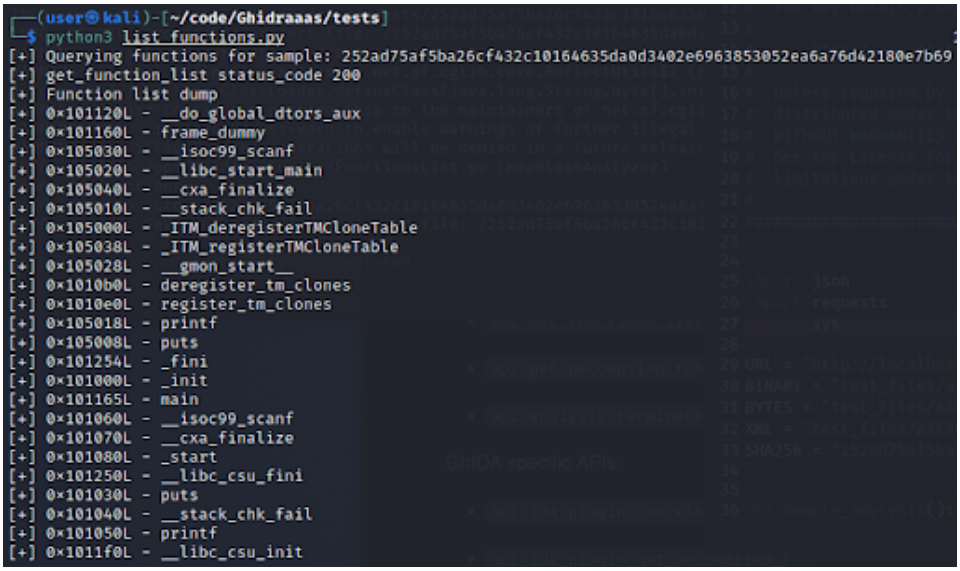

Figure 2.4: A simple script to list functions and their resolved names

```
┌──(user㉿kali)-[~/code/Ghidraaas/tests]
└─$ python3 decompile_func.py 0x101165L
[+] Querying for decompiled in sample 252ad75af5ba26cf432c10164635da0d3402e6963853052ea6a76d42180e7b69
for function at offset: 0x101165L
[+] Decompiling function ad address: 0x101165L
[+] get_decompiled_function status_code 200

undefined8 main(void)

{
  long in_FS_OFFSET;
  uint local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Hello, World!");
  puts("This is a super simple test program for Ghidraaas.");
  printf("Please, insert a number: ");
  __isoc99_scanf(&DAT_00102065,&local_14);
  printf("%d\n",(ulong)local_14);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

Figure 2.5: A simple script to request the decompilation of a function

Use Cases and Limitations

Ghidraaas does not attempt to be a substitute for a deep dive into a binary with Ghidra. Rather, it provides an interface for building custom analyses and applications that can easily make use of Ghidra's considerable capabilities. For example, here are a few ideas for useful applications that could be built using Ghidraaas:

- Search for I/O functions, e.g. read/write for easier input tracing
- Scan strings for:
  - DLL names with known hijack locations
  - Known vulnerable functions (e.g. strcat)
- Find loops with regex for human examination
- Compare the decompilation of multiple versions of binaries to help narrow down searches for N-day bugs

2.3 McSema and Remill

Reference Link	https://github.com/lifting-bits/mcsema https://github.com/lifting-bits/remill
Target Type	Binary (Binary Ninja Plugin)
Host Operating System	Windows, MacOS, Linux
Target Operating System	Windows, Linux
Host Architecture	x86, x64
Target Architecture	X86, x64, AArch64, SPARCv8+ (SPARC32), SPARCv9 (SPARC64), AArch32 (in development)
Initial Release	Jan 2015 (McSema) Dec 2017 (Remill)
License Type	Open-Source (McSema = GNU GPL 3.0, Remill = Apache 2.0)
Maintenance	Maintained by TrailOfBits

Overview

McSema is a binary lifting framework [5]. The goal of a binary lifting framework is to take, as an input, compiled binary code (typically an executable or library file) and produce a higher level, semantically equivalent, representation of that code. Remill is the library McSema uses to perform the translation from binary code to an intermediate representation (IR), specifically LLVM IR bitcode [6]. Once in bitcode form, this output can be fed into numerous other tools in the LLVM ecosystem. Since the LLVM compiler framework analysis and optimization passes operate over LLVM IR, the output of McSema (and any valid modifications) can be fed into Clang to produce updated binary code for any architecture for which a LLVM backend (i.e., the translator between LLVM IR and machine code) exists.

The result is the ability to take compiled code, decompile it, modify the decompiled code, then recompile the updated code for numerous target architectures.

Design and Implementation

McSema's architecture has two distinct components: instruction boundary isolation and instruction lifting. Instruction boundary isolation describes the process of taking a chunk of binary code and dividing that code into discrete instructions. For some architectures, like x86 and amd64, this can be a difficult task due to characteristics of the architecture's instruction set (e.g. embedding data in code segments and variable-length instruction encodings). McSema offloads this by relying on other tools for decoding instructions and generating control flow graphs (and the basic blocks that make up those graphs). Currently, it relies on IDA Pro for this functionality. However, some work has been done to support using Binary Ninja[7] and Dyninst for this purpose.

McSema depends on Remill for performing instruction lifting. Remill is a library that supports translating machine instructions to LLVM IR bitcode. Once McSema obtains a stream of bytes for a single machine instruction, it passes the stream of bytes to Remill to lift into LLVM IR bitcode. Remill creates its own internal data structure representation as illustrated by figure 2.6.

```
;; mov eax, 1
(X86 804b7a3 5 (BYTES b8 01 00 00 00)
  MOV_GPRv_IMMv_32
    (WRITE_OP (REG_32 EAX))
    (READ_OP  (IMM_32 0x1)))
```

Figure 2.6: String representation of a decoded MOV x86 instruction

Each of these data structures is then mapped to a template that implements the semantics of that instruction (figure 2.7).

```
template <typename D, typename S>
DEF_SEM(MOV, D dst, const S src) {
  WriteZExt(dst, Read(src));
  return memory;
}
```

Figure 2.7: Remill semantics for the MOV x86 instruction

For each decoded instruction, Remill creates a new function. The implementation of the semantics of the decoded instruction are placed inside this function. The CFG data from McSema is used to group these functions into basic blocks. Similar to a symbolic execution engine, each basic block tracks modifications to three values: state, program counter, and memory. The state structure tracks things like register values, the program counter tracks

execution in the program, and memory tracks updates to program memory. Each basic block contains templated code for handling operations common to all basic blocks, such as making values in the State structure available for use in the function (e.g. assigning register values to local variables) and updating the program's view of memory after the instructions in the basic block are executed (figure 2.8).

```
// Instructions will be lifted into clones of this function.
Memory *__remill_basic_block(State &state, addr_t curr_pc, Memory *memory) {

  ...

  auto &EAX = state.gpr.rax.dword;
  auto &EBX = state.gpr.rbx.dword;
  auto &ESP = state.gpr.rsp.dword;

  ...

  auto &SS_BASE = zero;

  ...

  // Lifted code will be placed here in clones versions of this function.
  return memory;
}
```

Figure 2.8: Remil basic block template

The result of this process is a function representing a single basic block taken from the decoded program. Each decoded instruction that exists within the basic block is represented by a function call in the newly created basic block function. Each of these function calls encode the semantics of the instruction to which they correspond.

The LLVM IR bitcode produced by Remill has a few notable differences from the LLVM IR bitcode that would be produced for the same source code by one of LLVM's many frontends. One significant difference is how Remill handles memory accesses and some control transfer operations. In both cases, Remill utilizes a set of functions, referred to as "intrinsics." For example, when lifting a memory read, Remill will produce a call to `__remill_read_memory_*` (depending on the size of the read) instead of a call to LLVM IR's generic "load" instruction. Since these intrinsics can be defined by developers, they provide additional flexibility for generating accurate bitcode. Alternatively, these intrinsics can be defaulted to LLVM's store/load instructions. This is also the case with some control transfer mechanisms. For example, the implementation of handling interrupts (the semantics of which are architecture-dependent) is performed inside of intrinsic calls.

The LLVM IR bitcode produced by Remill also contains code that depends on the existence of a runtime environment. For example, Remill tracks changes to program state (e.g. register values) in the State structure. Examples of the use of this structure can be seen in the template function code for both Remill basic blocks and machine instructions.

As one would expect, the State structure didn't exist in the original binary, nor does it exist in the program's memory when executed. Instead, the structure is maintained by a runtime environment added by Remill.

Upon completion of the lifting process, the product is a semantically equivalent, but highly inefficient, representation of the original machine code. This inefficiency can be significantly reduced by the aggressive application of optimization passes, which substantially reduces the size of the bitcode.

Use Cases and Limitations

An advantage of using LLVM IR as a higher level representation is that there is a large ecosystem of existing tools that operate on LLVM IR bitcode. Since McSema produces LLVM IR bitcode, this ecosystem, which was once limited to developers with source code access, becomes available for use with blackbox binaries as well. This ecosystem includes symbolic execution tools like KLEE [8][9] and SymCC [10][11], static analysis tools like Sys [12][13], and fuzzing frameworks like libfuzzer [14]. Additionally, the entire LLVM framework is available to use on the generated code. Specifically, the bitcode produced by McSema can be fed into both the middle (optimizations) and back ends (code generation) of LLVM.

The major limitation of McSema is its reliance on other tools for performing instruction boundary isolation and CFG generation. The only officially supported tool is IDA Pro, which can be prohibitively expensive. While some support exists for Dynist, which is free and open source, this support is unofficial and, as such, not guaranteed to continue to work with each new release of McSema.

Depending on the use case, Remill's emphasis on recompilability may make consumption of the generated bitcode more difficult. Many state changes that occur in a compiled binary end up being represented as loads and stores into a runtime-managed data structure. This results in a very verbose IR that can be difficult to read if you are not interested in modeling machine-level state changes.

2.4 Preeny

Reference Link	https://github.com/zardus/preeny
Target Type	Binary
Host Operating System	Linux (debian, Arch, or Fedora-based) Partial OSX support
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	August 2016
License Type	Open-Source (BSD)
Maintenance	Last updated April 2021

Overview

Preeny is a binary analysis aid for disabling inconvenient program behaviors. CTF challenges often contain non-essential code that make binary analysis difficult. Two motivating examples are the C system calls `alarm()` and `fork()`. Occasionally, players will connect to CTF servers, move on something else, and leave their connection open. To prune these connections, binaries set timers via `alarm()` that upon expiration, close the dead connection. While this behavior protects binaries on the server, it disrupts analysis after users copy them onto their own local systems. To mitigate this, Preeny simply hooks the call to `alarm()`, and returns 0 as if no call was ever made. Second, CTF binaries may contain calls to `fork()` for various purposes. Unfortunately this disrupts dynamic reconnaissance without careful debugger management. GDB advises:

> Put a call to `sleep` in the code which the child process executes after the fork. It

> may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the ps program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see `Attach`). From that point on you can debug the child process just like any other process which you attached to. [15]

Preeny provides a greatly simplified solution. As with `alarm()`, Preeny hooks the `fork()` call and returns 0 as if the child process succeeded.
In addition to circumventing problematic syscalls, Preeny also offers functions for aiding general dynamic analysis. Scripts exist for binary patching, stack canary dumping, and RNG control.

Design and Implementation

For syscall disabling, Preeny hooks calls through the `LD_PRELOAD` environment variable. For example:

```
LD_PRELOAD=x86_64-linux-gnu/dealarm.so ~/alarmedBinary
```

Accomplishing this in code is straightforward:

Listing 2.1: Dealarm.c

```
unsigned int alarm(unsigned int seconds)
{
        preeny_info("alarm blocked\n");
        return 0;
}
```

More advanced scripts are ad-hoc, such as setcanary.c[16]:

Listing 2.2: setcanary.c

```
#ifdef __x86_64__
#define TONUMBER      strtoull
#define INSN_LOAD     "mov %0, %rax;"
#define INSN_WRITE    "movq %rax, %fs:0x28;"
#define REG           "%rax"
```

and patch.c[17]:

```
# tests/hello
Hello world!
# cat hello.p
[hello]
```

```
address=0x4005c4
content='4141414141'

[world]
address=0x4005ca
content='6161616161'
# PATCH="hello.p" LD_PRELOAD=x86_64-linux-gnu/patch.so tests/hello
--- section hello in file hello.p specifies 5-byte patch at 0x4005c4
--- section world in file hello.p specifies 5-byte patch at 0x4005ca
AAAAA aaaaa!
```

Use Cases and Limitations

Preeny supports the following operations:

Name	Summary
dealarm	Disables alarm()
defork	Disables fork()
deptrace	Disables ptrace()
derand	Disables rand() and random()
desigact	Disables sigaction()
desock	Channels socket communication to the console
desock_dup	Channels socket communication to the console (simpler method)
ensock	The opposite of desock -- like an LD_PRELOAD version of socat!
desrand	Does tricky things with srand() to control randomness.
detime	Makes time() always return the same value.
desleep	Makes sleep() and usleep() do nothing.
mallocwatch	When ltrace is inconvenient, mallocwatch provides info on heap operations.
writeout	Some binaries write() to fd 0, expecting it to be a two-way socket. This makes that work (by redirecting to fd 1).
patch	Patches programs at load time.
startstop	Sends SIGSTOP to itself on startup, to suspend the process.
crazyrealloc	ensures that whatever is being reallocated is always moved to a new location in memory, thus free()ing the old.
deuid	Change the UID and effective UID of a process
eofkiller	Exit on EOF on several read functions
getcanary	Dumps the canary on program startup (x86 and amd64 only at the moment).
setcanary	Overwrites the canary with a user-provided one on program startup (amd64-only at the moment).
setstdin	Sets user defined STDIN data instead of real one, overriding `read`, `fread`, `fgetc`, `getc` and `getchar` calls. Read here for more info
nowrite	Forces open() to open files in readonly mode. Downgrading from readwrite or writeonly mode, and taking care of append, mktemp and other write-related flags as well

Figure 2.9: Supported Functions

While Preeny was specifically designed for CTF binary analysis, many capabilities are useful for real-world analysis. Targets often contain similar, non-essential code. Preeny offers quick solutions for spinning up binary exploration. While Preeny is ideal for CTF binary analysis, it is important to note that Preeny is not suited for known-malicious or completely-unknown binary analysis. For example, preloading `desock` on a malicious bi-

nary does not guarantee that it will be unable to open a channel, cause damage, or carry out other undesired behaviors.

2.5 Rizin

Reference Link	https://github.com/rizinorg/rizin
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux, Win, iOS, MacOS, Android, *BSD, Solaris, QNX, Haiku, FirefoxOS
Host Architecture	x86_64
Target Architecture	x86_64, i386, ARM, MIPS, PowerPC, SPARC, RISC-V, SH, m68k, m680x, AVR, XAP, System Z, XCore, CR16, HPPA, ARC, Blackfin, Z80, H8/300, V810, V850, CRIS, XAP, PIC, LM32, 8051, 6502, i4004, i8080, Propeller, Tricore, CHIP-8, LH5801, T8200, GameBoy, SNES, SPC700, MSP430, Xtensa, NIOS II, TMS320 (c54x, c55x, c55+, c66), Hexagon, DCPU16, LANAI, MCORE, mcs96, RSP, SuperH-4, VAX, AMD Am29000
Initial Release	22 Jan 2021
License Type	Open-Source (LGPL-3.0)
Maintenance	Regular commits and releases with a large developer community (approximately 60 contributors).

Overview

Rizin is a command-line driven, open source reverse engineering (RE) platform. It includes a disassembler, hex editor, debugger, binary analysis features, patching features, and an extensible plugin architecture. As a fork of radare2 [18], Rizin shares many of the same features and interfaces. A subset of the radare2 contributors created Rizin in December 2020 as a response to disagreements with the technical direction of the project and some of the content in the codebase [19]. Work to date has focused on reimplementing features to improve testability and stability, as well as code refactoring to support maintenance[20]. Rizin also has a GUI version, called Cutter, which additionally includes a plugin that interfaces with the Ghidra decompiler[21]. Cutter's back-end was changed from radare2 to Rizin, as much of the core Cutter team has joined the project.

Design and Implementation

Rizin is primarily written in C and is built using the Meson build system. During evaluation, we found the Meson build process to be straightforward and surprisingly quick compared to an autotools build of radare2.

Like radare2, Rizin exposes features as command line tools to allow ease of scripting tasks [22]. Supported commands are:

- rz-ax - expression evaluation for converting between data types
- rz-find - search for string and byte sequences
- rz-run - initialize an emulation environment for debugging a program
- rz-bin - extract and output properties of binary files
- rz-diff - perform binary diffing
- rz-asm - assembler/disassembler
- rz-gg - construct basic blocks for injection into binaries
- rg-hash - compute hashes over files or sections of files

As a fork, Rizin's architecture is mostly similar to radare2's. One major distinction is the degree Rizin has been streamlined. To date, approximately 70 commands have been removed or consolidated from radare2 [23]. Additional changes include major work in the Java bytecode plugin, Lua 5.4 support, command parsing (figure 2.10), and reimplementation of saving and restoring projects.

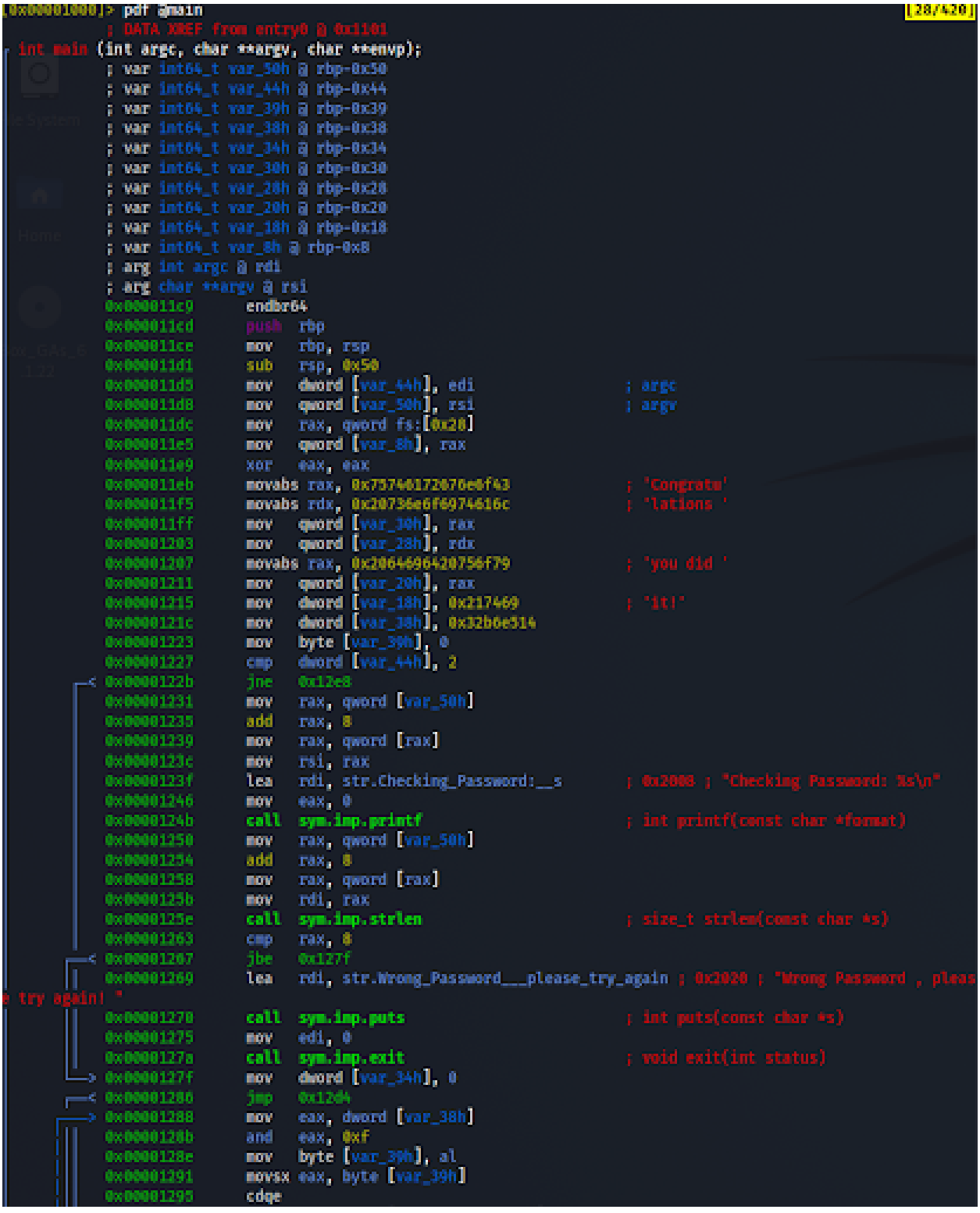

Figure 2.10: Rizin disassembly output of a program's main() function

Use Cases and Limitations

Rizin offers multiple options for scripting and integration with a focus on composability. Aside from the traditional Rizin shell environment, it's possible to create scripts using the individual command line tools, or integrate Rizin features into other programs using the rz-pipe project (also maintained by Rizinorg) [24]. As in radare2, rz-pipe provides a simple API to execute Rizin commands from other programs written in Python, Go, Haskell, OCaml, Rust, and Ruby. Rather than expose the backend API, rz-pipe accepts standard commands as they would be run in the Rizin environment, and the output is returned to the caller.

While not strictly a limitation, Rizin's inherent similarities to radare2 has its drawbacks. Users that are already familiar with radare2 commands will find Rizin easy to pick up. On the other hand, unfamiliar users should expect a steep learning curve. Because of this, at this stage in Rizin's development, the similarities to radare2 are a hindrance to wider adoption. So far, the new features do not offer a compelling reason for RE analysts to switch to Rizin. For upcoming releases, the developers are predominantly focused on bug fixes and code refactoring to support future enhancements and ease developer pain [25]. This focus on back-end improvements, addressing needed maintenance, and unit testing should yield stability and security benefits that may make it a better long term choice over radare2. Along with a stated emphasis on inclusivity and welcoming new developers, these changes may also encourage a larger base of regular contributors to the project than radare2. Hopefully once this tech debt has been addressed, more prominent user-facing changes will be made to the project, although the roadmap to 1.0 release has not been established as of this writing.

The only major user feature to differentiate it from radare2 is the completely rewritten, stable project support (compared to radare2's buggy implementation) that makes saving and loading the current project state painless. Project metadata and modifications are described using JSON files. This makes it straightforward for multiple analysts to collaborate on a project by committing project directories in git and merging their changes. A new API for developing and parsing user commands, called newshell, has been implemented to address longstanding inconsistencies in command parsing [26].

2.6 The Software Analysis Workbench (SAW)

Reference Link	https://saw.galois.com/
Target Type	C (via LLVM Bitcode) Java (via Java Bytecode) Cryptol (a domain-specific language related to SAW)
Host Operating System	Linux Windows macOS
Target Operating System	N/A
Host Architecture	N/A
Target Architecture	N/A
Initial Release	04/2015
License Type	Open Source BSD 3-Clause ”New” or ”Revised” License
Maintenance	Galois, Inc., on Github

Overview

The Software Analysis Workbench (SAW) is a tool intended to carry out formal verification of code (primarily Java and C) [27]. It provides a specification language (Cryptol) for creating verification properties to be proved as well as a scripting language for building up proofs of larger and more complex systems. Verification is carried out by constraint solvers (SAT/SMT), resulting in a potentially high level of automation.

Design and Implementation

SAW uses Cryptol, a domain-specific language for cryptography, as a specification language for describing properties of the code under analysis. Cryptol is a purely functional, lazily evaluated language with a strict type system that makes it compact and amenable to automated analysis. Here is a Cryptol implementation of the SHA-1 hash algorithm:

Listing 2.3: Cryptol example

```
f : ([8], [32], [32], [32]) -> [32]
f (t, x, y, z) =
  if (0 <= t)  && (t <= 19) then (x && y) ^ (~x && z)
   | (20 <= t) && (t <= 39) then x ^ y ^ z
   | (40 <= t) && (t <= 59) then (x && y) ^ (x && z) ^ (y && z)
   | (60 <= t) && (t <= 79) then x ^ y ^ z
   else error "f: t out of range"
```

SAW derives most of its verification power from external Boolean satisfiability (SAT) and satisfiability modulo theory (SMT) solvers. Properties amenable to solving by SAW are of two general forms:

- Find an input (or inputs) that meet some condition
- Show that some condition is met for all inputs

Some useful examples of properties specified in Cryptol (and therefore in SAW) are [28]:

- Function reversal: Find an input x that results in an output y
- Proof of inversion: Show that for all inputs x, g(f(x)) == x
- Proof of injectivity: Show that for all inputs x and y, x != y implies f(x) != f(y)
- Collision detection: Find inputs x and y where f(x) == f(y) and x != y
- Equivalence checking: Show that for all inputs x, f(x) == g(x)

SAW can be used to prove these and other properties. It does so by first symbolically executing the program in question. This generates a logical representation (Boolean variables, ands, nots, etc.) of the entire functionality of the program, which can then be passed to a SAT/SMT solver. These highly optimized tools are able to automatically solve many problems of surprising complexity over the entire input space of the program, resulting in an automated approach that provides confidence far exceeding any number of individual test cases.

Crucially, SAW extends this solving capability to other languages as well. Using a framework known as Crucible, SAW can also symbolically execute programs written in C and Java (the former by compiling to LLVM bitcode and the latter by compiling to Java bytecode.) This allows the analyst to prove properties about “real” software implementations,

potentially even highly optimized and difficult to read ones, in much the same way they would native Cryptol code.

In addition to proving user-defined properties, the symbolic execution process also generates additional proof obligations that must be met that can sometimes detect flaws such as out of bounds memory references.

As a compact example, consider this simple C program, assert-null.c, from the SAW example repository. [29]:

Listing 2.4: assert-null.c

```
int f(int *x) {
    return (x == (int *)0);
}
```

To analyze this with SAW, first compile to LLVM bitcode:

Listing 2.5: LLVM bitcode generation

```
llvm -c -emit-llvm assert-null.c
```

To confirm that this function returns 0 if a pointer is passed and 1 if null is passed, the following SAW script, assert-null.saw, can be used:

Listing 2.6: assert-null.saw

```
let f_spec1 = do {
    p <- llvm_alloc (llvm_int 32);
    llvm_execute_func [p];
    llvm_return (llvm_term {{ 0 : [32] }});
};

let f_spec2 = do {
    llvm_execute_func [llvm_null];
    llvm_return (llvm_term {{ 1 : [32] }});
};

m <- llvm_load_module "assert-null.bc";
llvm_verify m "f" [] false f_spec1 abc;
llvm_verify m "f" [] false f_spec2 abc;
```

To run the prover, execute the following:

Listing 2.7: SAW prover invocation

```
saw assert-null.saw
```

Use Cases and Limitations

Despite its name, the Software Analysis Workbench may not be appropriate for all analysis tasks. Its primary use case is cryptographic verification. There may be vulnerability research use cases, but its ability to find bugs depends on how well a program's intended functionality can be specified and, crucially, on whether the target is amenable to analysis by symbolic execution and SAT/SMT solving.

SAW has been used primarily to verify cryptographic implementations (including libgcrypt, Bouncy Castle, and Amazon s2n) which, while sometimes large and complex, have the advantage that their intended functionality is mostly well-specified. Other software may be difficult to verify, especially if it isn't well understood by the analyst and therefore the intended functionality isn't known. That said, it is possible to create partial specifications, such as specifying a range of return values and placing bounds on memory regions to ensure those bounds are respected.

Symbolic execution also has its limitations. Cryptographic algorithms tend to be highly structured, always-terminating, and composed of bounded looping constructs. These are useful properties for preventing the "path explosion" issue that can occur with more general programs. This means that symbolic execution may not be able to process certain functions, even for example small ones containing loops with an input data-dependent number of iterations. SAW does provide constructs for scaling to larger programs, such as breaking proofs down by function and building upon other proofs. In addition, there is the capability to carry out more complicated analysis, such as manually applying rewrite rules, in a similar style to interactive theorem provers, to crack particularly difficult problems. However, the rewrite capability is not well documented and is for advanced users only.

Finally, SAT/SMT solving is also limited. The Boolean satisfiability problem, the theory underlying all of these solvers, is in general NP-Complete, meaning that there are problem instances that cannot be solved in a tractable amount of time using known methods. Fortunately (and in part due to hard work by the SAT solver community), some classes of real-world problems, such as certain questions applied to cryptographic software, happen to be easily solvable by these tools. However, many real-world programs and real-world questions remain out of reach.

Dynamic Analysis

Whereas static analysis examines a binary without running it, dynamic analysis observes a binary as it executes. Dynamic analysis allows the inspection of actual runtime information about program state, including register and memory values. However, it cannot provide code coverage guarantees. Both approaches provide valuable insights into a program. Dynamic analysis techniques range from empirical observations of program execution to crafted instrumentation approaches that support a wide range of analyses.

A general overview of dynamic analysis can be found in the appendix.

3.1 Aurora

Reference Link	https://github.com/RUB-SysSec/aurora
Target Type	Binary (Source code may be used to enrich results)
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86_64
Target Architecture	x86_64
Initial Release	12 Aug 2020
License Type	Open-Source (AGPL-3.0)
Maintenance	Academic release, last Github commit Aug 2020 by @mrphrazer

Overview

Aurora is an automated root-cause analysis tool that attempts to determine the location and context that leads to a crash [30]. Starting with a single crashing input, it generates a population of crashing and non-crashing inputs. It then uses dynamic binary instrumentation to record detailed state information on the execution of each input. It derives Boolean predicates from the state information. Finally, it applies statistical analysis to determine which predicates best distinguish crashing from non-crashing inputs. Aurora is implemented in three components that can be built for a Linux system and is also available as a Docker container.

Design and Implementation

Aurora has three primary components: a modified AFL for crash exploration (figure 3.1), an Intel Pin[31] tracer, and a Rust-based root cause analysis (RCA) component. The crash explorer generates and saves both crashing and non-crashing inputs. The tracer is used to trace each input and save dynamic execution information such as register values to use as the basis for predicates. Finally the RCA component generates and evaluates the predicates to determine their ability to distinguish crashes and presents a ranked list of predicates for a user to review.

In order to do effective statistical analysis, Aurora requires a diverse but similar set of inputs. The inputs must be diverse enough to reveal measurable differences, but similar enough that they explore states close enough to the root cause. To achieve this, authors modified a version of AFL's "crash exploration" mode so that it saves both crashing and non-crashing inputs. Crashing inputs are saved for additional mutation, but non-crashing inputs are not mutated further, generating a population of inputs with behavior clustered around the original crash.

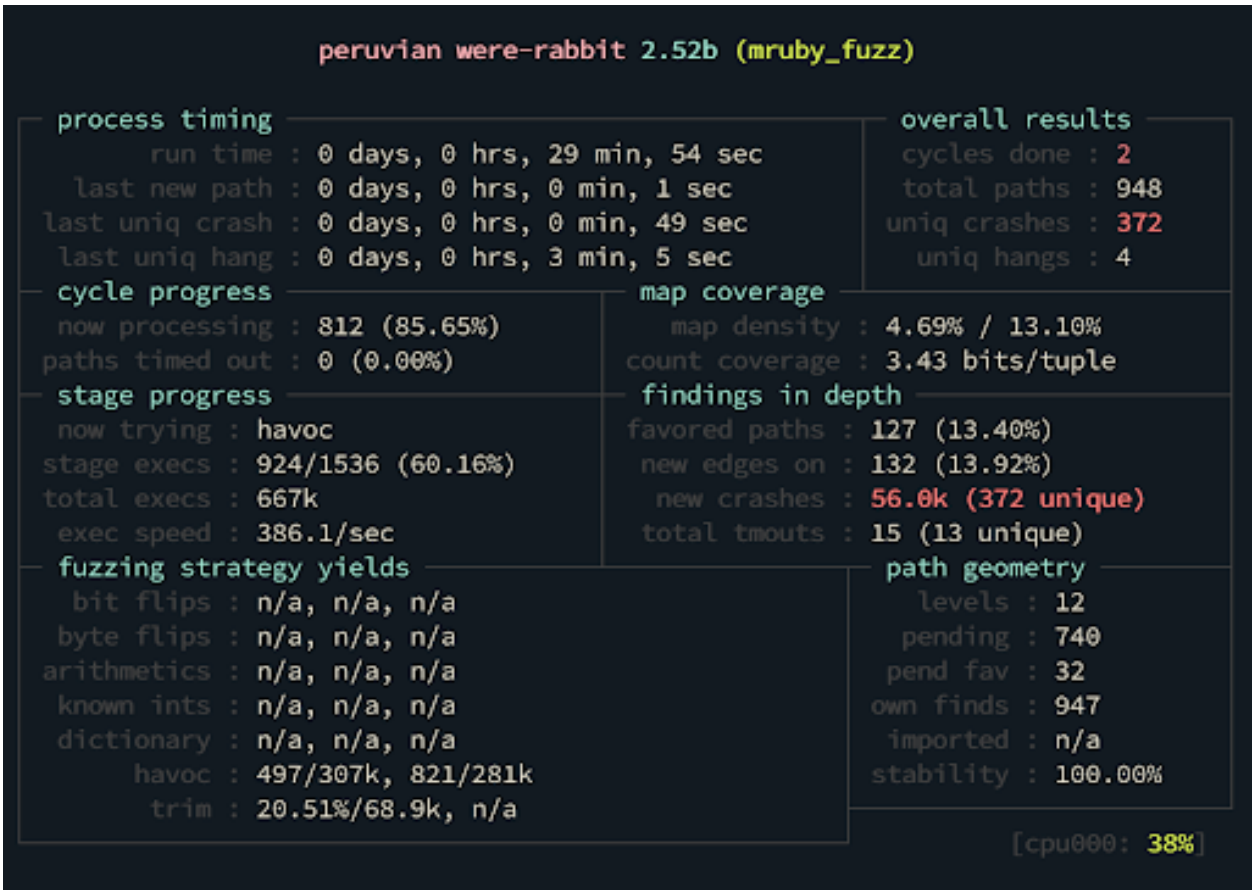

Figure 3.1: Modified AFL version used for crash exploration

Given the set of crashing and non-crashing inputs, the next step is to collect details of an

input's behavior in the form of control-flow edges and minimum and maximum values for registers and memory writes. Aurora implements a custom Pin tracer to record this information for every input. This collects the values of several expressions for each instruction executed in the target binary; this information is used as the basis of predicates which can be used in the next step: explanation synthesis.

The RCA component of Aurora is the most interesting contribution as its statistical analysis approach allows it to present boolean predicates for bug conditions, even in the case where there is no data dependency between the root cause and the crash. Most approaches before Aurora used techniques like backward slicing to follow data dependencies to a root cause of a value that leads to the observed bug condition, but some types of bugs are resistant to this type of analysis. Aurora's approach is to generate many predicates for each instruction, rank their ability to predict a crash, and then rank the best predicates from each instruction to present a list of simple boolean predicates to a human analyst to review.

Aurora generates three types of predicates: control-flow predicates, register and memory predicates, and flag predicates (from the flags register on x86/x64 architectures)(figure 3.2). The scoring of each predicate is done by averaging the percentage of correct predictions for crashing and non-crashing inputs, where a "perfect predicate" would correctly classify an input as crashing or non-crashing 100% of the time. Ties between predicates are broken by execution order, where predicates executed earlier in the program (as observed across all inputs) are favored as they are more likely to be the root cause.

Predicate Type	**Predicates**
Control-Flow	has_edge_to, always_taken
Register and Memory	$r < c$ for register and memory values r and constant c, is_heap_ptr(r), is_stack_ptr(r)
Flags	Carry flag, Zero flag, Overflow flag

Figure 3.2: Aurora Predicates

The output of Aurora is a ranked text file of predicates, where the authors' goal is to suggest locations (instruction addresses or lines in source code if debug symbols are available) and conditions that suggest where crashing and non-crashing behaviors diverge (figure 3.3). The output is functional, but not focused on aesthetics. Aurora's approach requires a human analyst to interpret and assess the suggestions, but this is a fundamental assumption that makes it possible to present a list of informed suggestions automatically rather than putting full responsibility on the automated analysis, which would lead to an unacceptable false-positive rate.

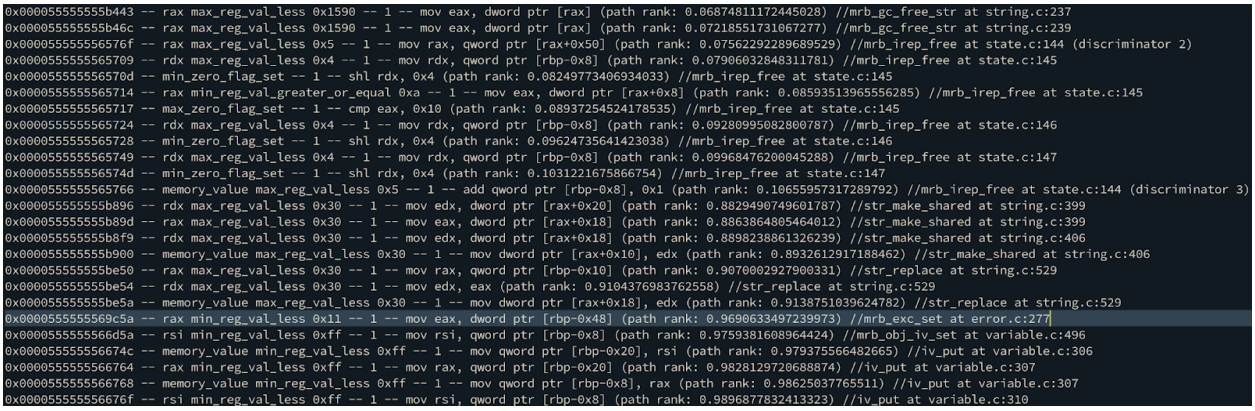

Figure 3.3: Root cause output excerpt with true root cause highlighted

Use Cases and Limitations

The nature of Aurora's analysis is that it synthesizes explanations for software bugs based on concrete observations and observed ability for a particular predicate to classify inputs into either crashing or non-crashing categories. This approach allows Aurora to infer root cause analysis of bug types without a priori knowledge of bug categories or behaviors. This flexibility comes at the cost of precision and speed, where other approaches might be able to directly detect a root cause. This suggests that the ideal use for Aurora is to provide supplemental data in the form of automatically generated starting points to human reviewers trying to understand the root cause of bugs found by automated fuzzing in continuous integration or continuous fuzzing frameworks such as Google's ClusterFuzz.

Bug Classes Aurora can detect:

- type confusion
- use-after-free
- uninitialized version
- heap buffer overflow
- null pointer deref
- stack-based buffer overflow

Given Aurora uses statistical analysis, it has the limitation that it is bound to report some predicates that are not related to bug condition. While the authors have used heuristics to reduce known false-positive patterns, it isn't perfect, and some bug patterns may not be simple enough to express in boolean predicates. In their experimental evaluation, they noted that some bugs were identified by the top-ranked predicate, while others required tens of predicates to be reviewed before finding the true root cause. That being said, it's unknown how much manual effort was involved in fixing the reference bugs used in the

authors' experiment, but they contend that the automated nature of Aurora would suggest it ends up saving developer time in most cases.

Aurora also relies on a fuzzer to create a diverse set of realistic inputs, which means it is only as good as the population the fuzzer can generate. The authors note that in the case of crashes found by grammar-based fuzzers, AFL's default mutation was often insufficient. It may also take a long time to generate a sufficiently large population of inputs for the statistical analysis to be effective, and the tracing and statistical analysis can also require large amounts of CPU time and RAM. Even with an extremely capable experimental setup, the authors noted that in their experiments Aurora ran for up to 17 hours using the timeout settings they chose. This seems like a long time, but the authors suggest this process should be fully automated as part of a fuzzing pipeline. Finally, Aurora was published to GitHub as a proof-of-concept for an academic paper, though the code has not seen updates or maintenance for several months at the time of this writing.

3.2 FANS

Reference Link	https://github.com/iromise/fans
Target Type	Binary
Host Operating System	Linux
Target Operating System	Android
Host Architecture	ARM (64-bit)
Target Architecture	Python 3
Initial Release	09/2020
License Type	Open-source (no specified license)
Maintenance	last updated 2020

Overview

FANS (Fuzzing Android Native Services) is a generation-based fuzzing tool for fuzzing Android native system services. To create inputs, FANS analyzes Android Remote Procedure Call (RPC) interfaces, and extracts corresponding semantic models. From these models, FANS then generates inputs to test target services. [32][33].

Design and Implementation

FANS is primarily built around how Android applications communicate with system services. To communicate with system services, Android applications query the Android Service Manager, to which all services are registered. In turn, the Service Manager provides remote procedure call (RPC) interfaces as an API. Services will then use the `onTransact`

function for processing application requests. These requests are referred to as ”transactions”.

FANS consists of four primary components which extend the transaction functionality:

- Interface collector
- Interface model extractor
- Dependency inferrer
- Fuzzer engine

These steps are shown in the figure 3.4.

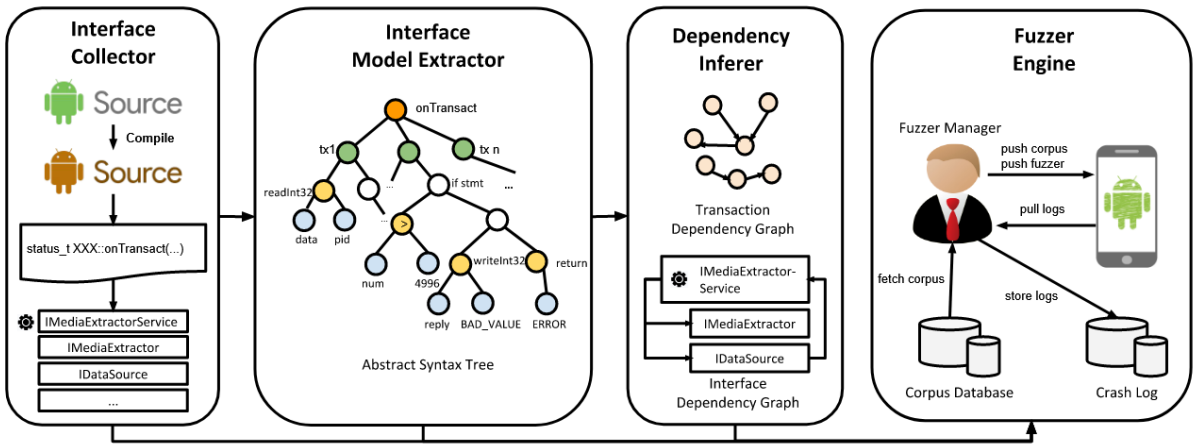

Figure 3.4: FANS design[32]

A more detailed explanation of this process is as follows:

1. The interface collector, `collector.py`, scans for the `onTransact` function in every C++ source file found in the compilation commands of the Android Open Source Project (AOSP). This way, both the interfaces generated from the Android Interface Definition Language files and also the interfaces initially written in C++ will be included in the model extraction.
2. The interface model extractor process will use the Clang compiler to convert the relevant C++ files to LLVM IR. It then uses the generated AST to identify the types of transactions based on the switch statement in the `onTransact` functions. Transaction inputs are extracted by analyzing the sequential reading of information from the data parcel, which is an argument to `onTransact`. E.g. `data.readInt32()`. Transaction outputs are extracted by analyzing the serialization of information in `reply`.
3. The dependency inferer will analyze the AST, discover dependencies between multi-level interfaces via the serialization of interfaces with `writeStrongBinder` and the deserialization of interfaces with `readStrongBinder`. Additionally, variable inter- and intra-transaction dependencies are inferred as well.

4. The fuzzer engine consists of two main parts, the fuzzer binary and the fuzzer manager. The fuzzer binaries are pushed directly onto the phones with adb and synced with the fuzzer manager. Transactions are randomly generated, checking for constraints on the transaction variables, checking if any dependent transactions are necessary, and then generating the variables themselves. Output is aggregated from ANRs, tombstones, and logcat. The Organization of aggregated output is shown in figure 3.5.

```
XXXX-XX-XX-XX-XX-XX
├── Device Serial        # the device serial
│   ├── 0                # how many times has this device been flashed
│   │   ├── anr          # Application Not Responding logs
│   │   ├── fuzzer_log   # fuzzer logs related with fuzzer itself
│   │   ├── logcat       # logcat logs generated by the smartphone
│   │   └── tombstones   # tombstones generated by the smartphone
│   ├── 1
│   │   ├── anr
│   │   ├── fuzzer_log
│   │   ├── logcat
│   │   └── tombstones
│   ├── 2
│   │   ├── anr
│   │   ├── fuzzer_log
│   │   ├── logcat
│   │   └── tombstones
│   └── device_manager.log # logs generated by the manager
├── Device Serial
...
```

Figure 3.5: FANS output.

Use Cases and Limitations

The primary use case of FANS is to have broad, scalable fuzzing of system services in the native environment, With the primary appeal being that fuzzing input is injected intelligently in the same manner as a potential adversary. The transaction generation method is novel, where fuzzers like BinderCracker generate input based on previously recorded transactions, the transactions generated by FANS use statically inferred grammar. This increases diversity and coverage by using infrequently used transactions and formats.

The primary target of FANS is the standard services that can be found in the Android Open Source Project (AOSP). This does not necessarily include vendor and hardware services. Additionally, due to the nature of fuzzing the system services, the phones are automatically reflashed when needed, slowing down the number of executions per second. Additionally there is a decent amount of overhead in setting up the environment for FANS. For example, AOSP must be compiled twice, once with address sanitization.

3.3 Frankenstein

Reference Link	https://github.com/seemoo-lab/frankenstein
Target Type	Binary - Broadcom Bluetooth processors
Host Operating System	Linux
Target Operating System	Broadcom proprietary OS built on ThreadX
Host Architecture	Python 3.6 and above
Target Architecture	ARM (32-bit) (generally)
Initial Release	09/2019
License Type	Open-Source
Maintenance	Maintained by Seemoo Lab

Overview

Frankenstein is a suite of tools for emulating and fuzzing Broadcom Bluetooth firmware [34]. It uses a complementary tool, InternalBlue, to collect firmware. Frankenstein patches the firmware, compiles ancillary files, and links all of these into a single, static ELF file. Subsequent instrumentation and QEMU emulation enables coverage based fuzzing. There are a variety of prepackaged, ancillary files and tools that enable fuzzing, emulation, and heap sanitization. A user with domain specific Bluetooth and coprocessor knowledge can efficiently write fuzzing harnesses in C that directly invoke code segments of the virtual chip.

Design and Implementation

Frankenstein collects firmware using the InternalBlue tool. Once collected, the firmware consists of segments that must first be reassembled. This firmware is designed to run on a bare-metal ARM 32 processor and must first be augmented to allow execution as a user space process. This is done through an ancillary C file that defines the entry point to the application. This can be seen in figure 3.6:

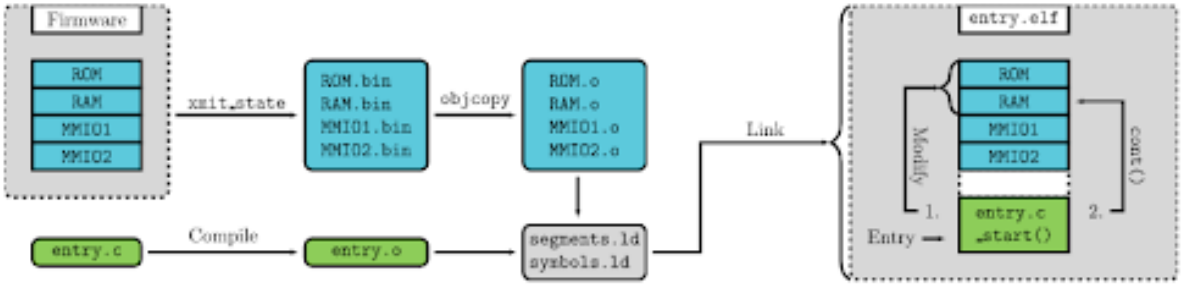

Figure 3.6: Components of the Frankenstein Emulation Process [35]

Frankenstein aids the user generating a mapping between function addresses and symbols. It uses this information to generate a linking file, `symbols.ld`, which allows the user to write test applications that link against the firmware. Frankenstein suggests the use of another tool, Polypyus [36], to discover differences between existing and new firmware in order to ease symbol creation. Once generated, the user can write custom, powerful test applications. There are a variety of prepackaged tools such as `execute` for testing non-interactable execution and `hci_attach` for full firmware, live fuzzing. Both of these tools can be used to target specific portions of the firmware. There is a prepackaged application `lmp_fuzz` for fuzzing the Bluetooth link manager. The code below, `hci_test.c`, demonstrates the ease with which test applications can be written. (Figure 3.7):

```
#include <frankenstein/utils.h>
#include "common.h"
#include "queue.h"
#include "lm.h"
#include "hci.h"

void hci_idle_loop() {
    while(1) {
        bcs_tick();
        contextswitch();
        check_and_handle_timers(1000);
        contextswitch();
        hci_rx_poll(1);
        contextswitch();
    }
}

void _start() {
    patch_code();
    idle_loop = hci_idle_loop;

    print("asd\n");
    hci_rx_fd = 0;
    hci_tx_fd = 1;
    print("asd\n");

    int rnd = open("/dev/urandom", O_RDONLY);
    acl_fd = rnd;
    inq_fd = rnd;
    page_fd = rnd;
    le_fd = rnd;

    //disconnect all connections
    //still do not know, why this crashes....
    print("asd\n");
    patch_jump(rm_getBBConnectedACLUsage, ret0);
    print("asd\n");

    //alarm(1);
    cont();
}
```

Figure 3.7: Fuzzing Harness for HCI Testing

The Frankenstein tool suite is a Django web application running on localhost that organizes work as projects. The web application helps generate build files, though compilation is accomplished independently from the command line. To use Frankenstein, a user must create or leverage an existing project, then load the appropriate binary segments and symbol data. A variety of existing Bluetooth projects and firmware come prepackaged with the tool, seen in Figure 3.8:

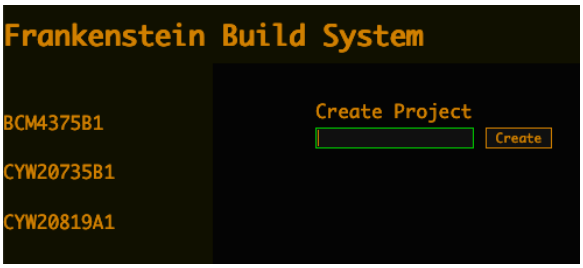

Figure 3.8: Project Creation Page of Frankenstein

The projects themselves map to folders on disk containing data files. Several existing sample projects are included with Frankenstein, as shown in Figure 3.9.

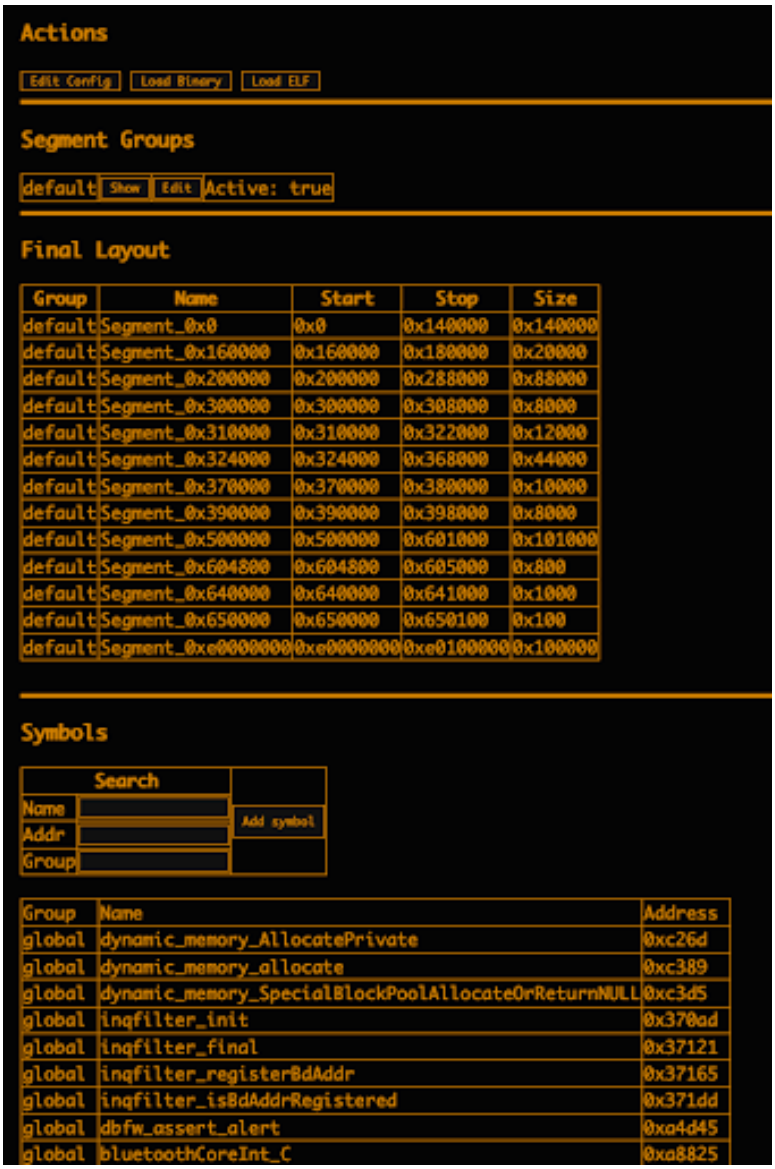

Figure 3.9: Existing Project Demonstrating Various Components

Using the web interface the user can specify segments, as shown in Figure 3.10.

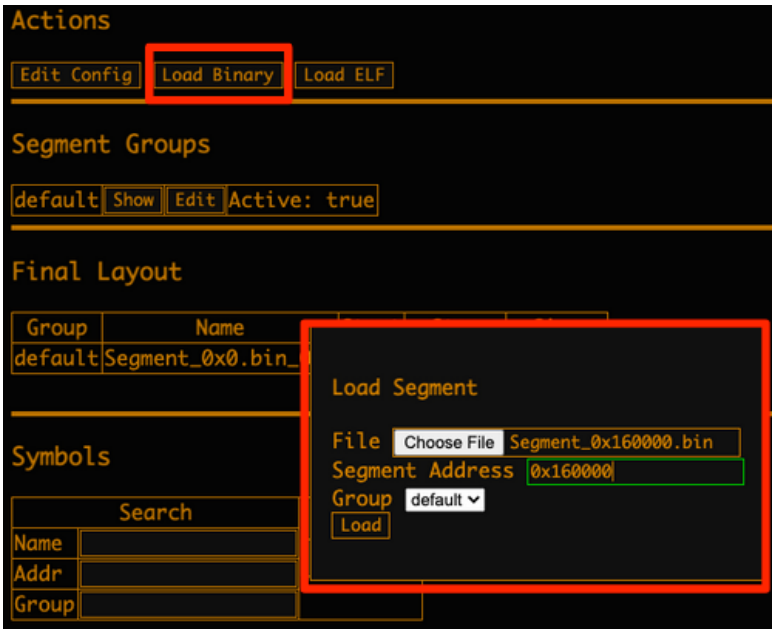

Figure 3.10: Loading Binary Segments into Frankenstein Project

Users may also specify symbols (Figure 3.11) and compiler flags (Figure 3.12).

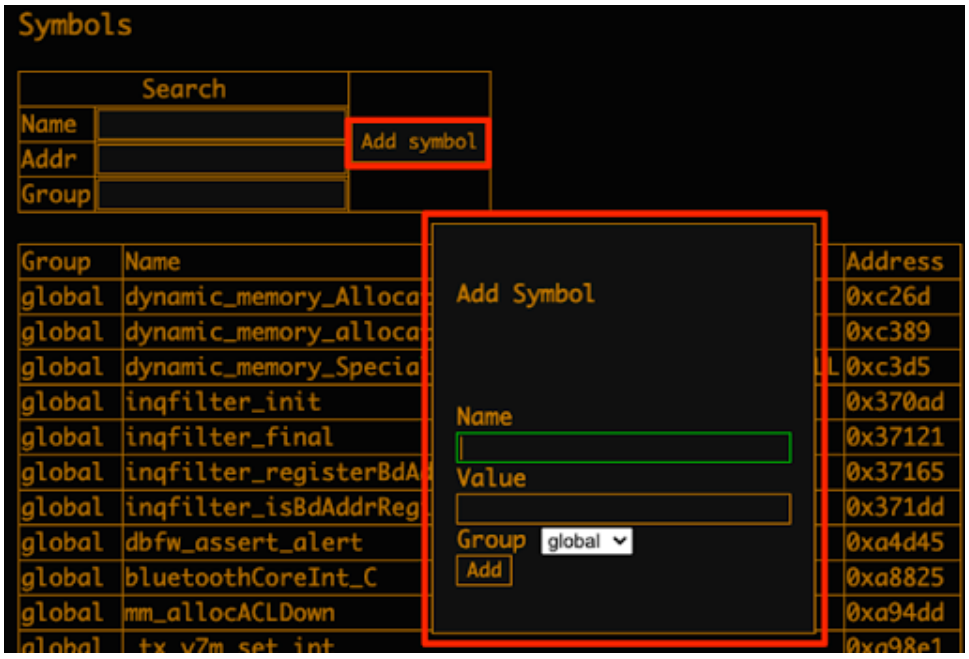

Figure 3.11: Adding Symbol Addresses to a Frankenstein Project

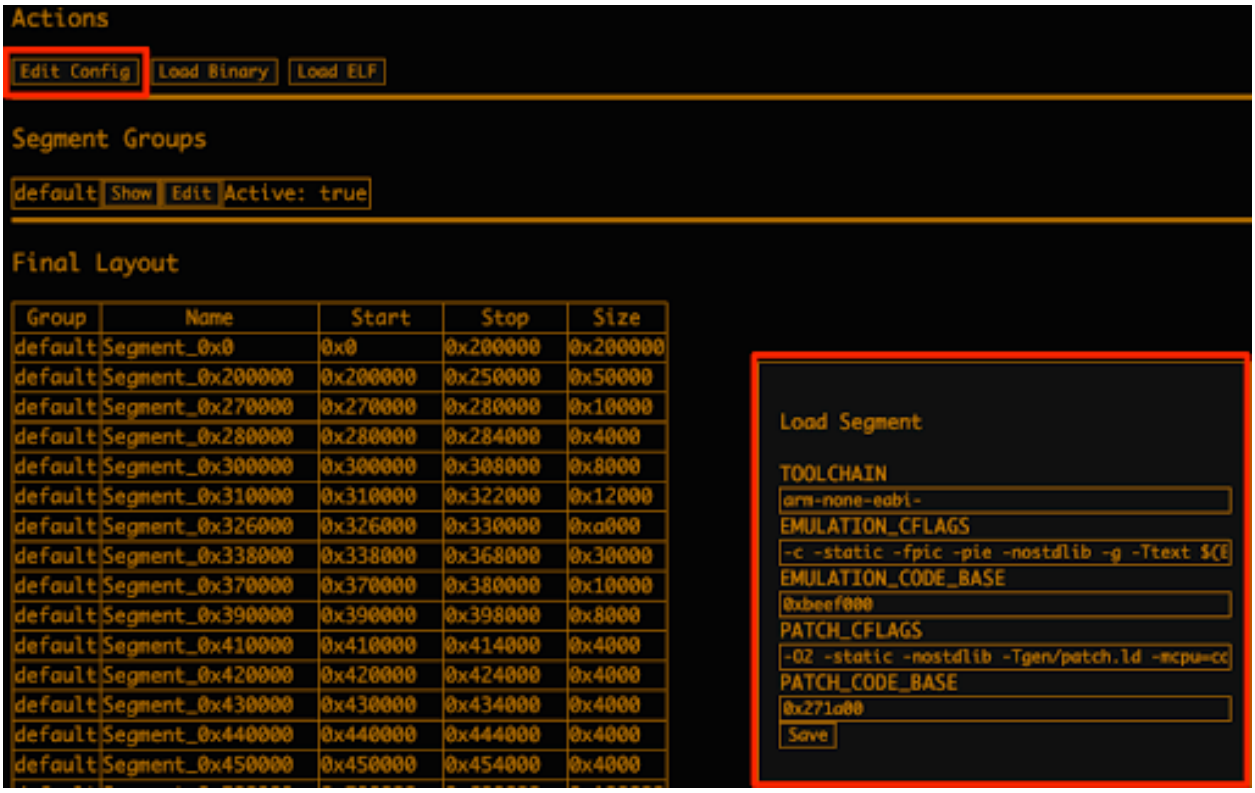

Figure 3.12: Specifying Compiler Flags for Generating the Frankenstein Emulator

The user must then invoke the compiler on the host with `make -C /projects/<Project Name>`. The resulting files are then visible and can be executed through the web application (Figure 3.13).

Figure 3.13: Components of Compiled Frankenstein Projects

The output, registers, and coverage metrics are displayed on the same page, seen in Figure 3.14.

Figure 3.14: Code Coverage Metrics of Frankenstein Emulator

Use Cases and Limitations

Frankenstein enables emulation of Broadcom Bluetooth firmware as virtual processors. The host operating system can register the emulated device as a Bluetooth processor, as shown in Figure 3.15, granting full stack fuzzing down to the hardware.

```
$ hciconfig
hci0:   Type: Primary  Bus: UART
        BD Address: 20:73:5B:17:69:31  ACL MTU: 1040:20  SCO MTU: 64:1
        UP RUNNING
        RX bytes:864 acl:0 sco:0 events:60 errors:0
        TX bytes:3273 acl:0 sco:0 commands:60 errors:0
```

Figure 3.15: Emulated Bluetooth Co-processor Registered to Operating System

The firmware is patched to allow remote excitation of transmission routines granting the user direct interaction with the firmware. Further, introducing symbols allows for flexible test application development. The prepackaged tools `hci_attach` and `lmp_fuzz` allows a user to send arbitrary packets to the Bluetooth co-processor through a UART[37] interface or by simulating over-the-air traffic, as seen in Figure 3.16.

```
$ cat /dev/urandom | ./hci_attach.exe
0x08Pts:/dev/pts/8
0x08Attaching Primary controller to /dev/pts/8
Switched line discipline from 0 to 15
Device index 0 attached
lr=0x024ea5 dynamic_memory_AllocateOrDie(0x14)lr=0x9a69 dynamic_memory_AllocatePrivate(0x2004
 = 0x21fb50;
lr=0x0252ed bthci_event_AttemptToEnqueueEventToTransport(0x21fb50)lr=0x024ce9 mpaf_hci_EventF
lr=0x024df7 bttransport_SendMsgToThread(0x21fb50, 0x0)Context switch lm -> bttransport
lr=0x01960d btuarth4_RunTxStateMachines(0x249e58, 0x01, 0x0, 0x0)lr=0x019265 uart_SendAsynch(
ynch)04040abf5684c795f80c025a01
lr=0x03fd03 uart_SendSynchHeaderBeforeAsynch(0x249f70, 0x249e10, 0x01, 0x0c);
lr=0x062c1d uart_DirectWrite(0x21fb58, 0x0c)HCI Event (Direct Write)040abf5684c795f80c025a01
 = 0x07;
 = 0x0;
lr=0x0193fb dynamic_memory_Release(0x21fb50) = 0x01;
 = 0x80000000;
lr=0x01de2d btuarth4_RunRxStateMachines(0x249e58) = 0x88000000;
Context switch bttransport -> lm
;
;
Context switch lm -> idle
```

Figure 3.16: Feeding Random Input to Frankenstein Fuzzing Harness

Unfortunately, this powerful emulation capability requires substantial domain specific Bluetooth and Broadcom knowledge. Generating symbols for the firmware also requires reverse engineering skills or leveraging a diffing tool. A working understanding of Bluetooth HCI tooling and QEMU emulation is necessary for sanity checking running firmware. Most importantly, this knowledge is necessary to differentiate bugs in Frankenstein from those in the emulated firmware. An example bug is show in Figure 3.17.

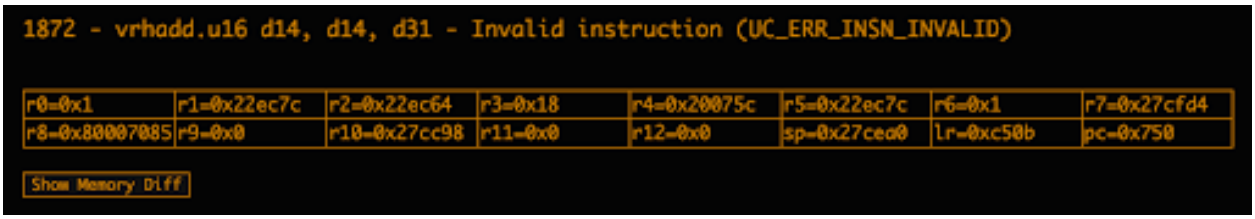

Figure 3.17: Invalid Instruction Encountered during Coprocessor Emulation

Collecting binary segments requires use of the tool InternalBlue and reintroducing symbols to segments that have not been previously examined requires reverse engineering or diffing existing binaries. Finally, writing test cases requires understanding control flow of the Broadcom operating system to properly handle fuzzing target functions while appropriately managing context switches and timers. That said, Frankenstein and its dependency InternalBlue are the most readily available tools for carrying out this type of analysis.

Minor Bugs While using the tool a variety of minor bugs were encountered. First, the tool needed more dependencies than listed - pip modules ans2html, unicorn, and capstone. The tool by default sets up a server on localhost but can be modified for remote use if desired. To do so, edit the file frankensteinWebUI/settings to change:

`ALLOWED_HOSTS=["127.0.0.1", <host name or public facing IP>]`

Then run: `python3 manage.py runserver 0.0.0.0:8000`

Another bug involved deleting symbols and segments. If there was no item to delete, it instead deletes the entire submenu. Finally, the tool doesn't properly render in Firefox as the showModal function is not supported, as shown in Figure 3.18. Therefore, it is recommended to use a different browser than Firefox.

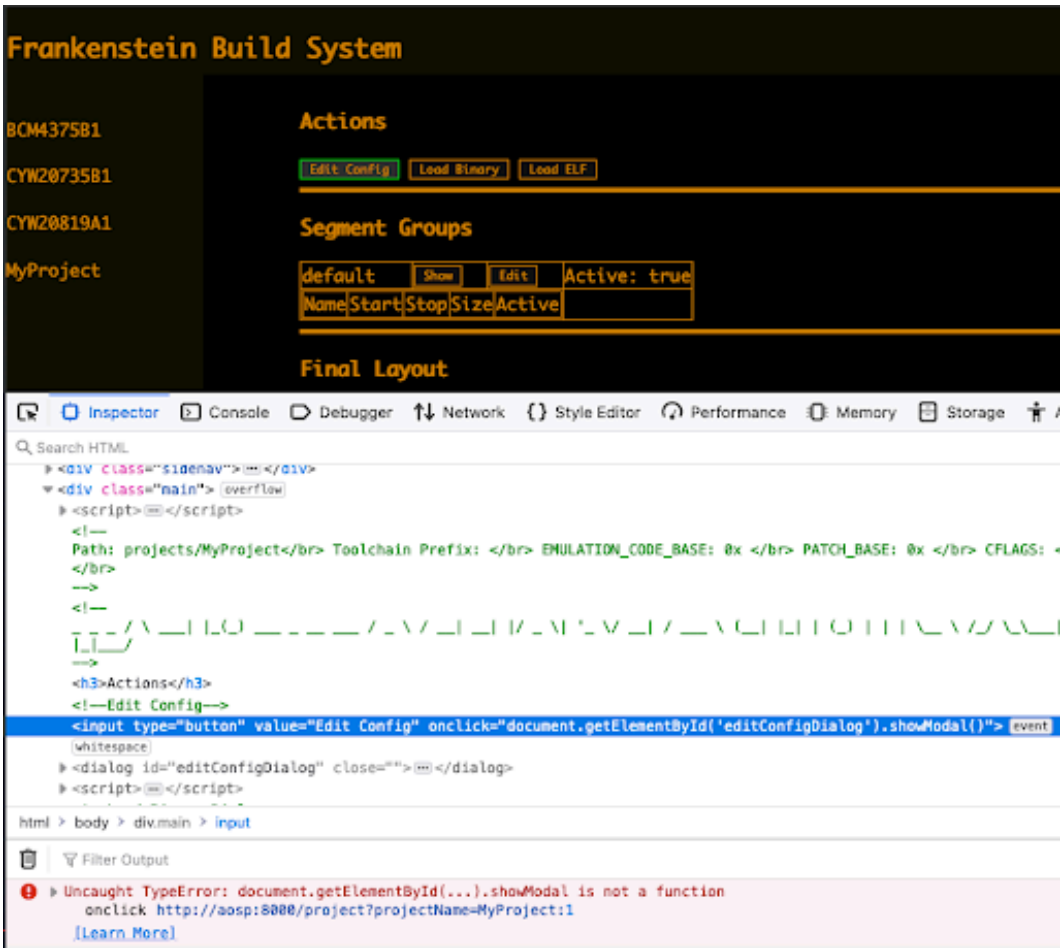

Figure 3.18: Rendering Errors in Frankenstein Project

3.4 FuzzGen

Reference Link	https://github.com/HexHive/FuzzGen
Target Type	Library source code
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86_64
Target Architecture	LLVM Bitcode
Initial Release	November 2019
License Type	Apache 2.0 (Open Source)
Maintenance	Last commit November 2020

Overview

FuzzGen[38] automates the creation of LibFuzzer[14] test harnesses for shared and static linked libraries. It shares similarities with other automatic driver/harness generation work. [39][40][41][42][43] FuzzGen scales the fuzz test harness generation in three distinct steps.

FuzzGen has been tested on the Android Open Source Project (AOSP) and with Debian source repository packages, and it found 17 previously undisclosed vulnerabilities. FuzzGen's performance was measured with code coverage as a proxy for efficacy, and the auto-generated fuzz test harnesses were compared against human built test harnesses. An overall observation was that FuzzGen created more fuzz test harnesses in a shorter amount of time than the humans, and the code coverage of the auto-generated test harnesses was usually, but not always, greater than that of the human generated ones.

Design and Implementation

FuzzGen generates a representation of binaries using a library. To do this, it performs a whole-system analysis to identify all code in the system that uses the fuzz target's external interface, and it generates a control flow graph (CFG) for these binaries. The fact that FuzzGen makes use of all available information from the system about how a library is used is significant because it means that the tool can build a more complete understanding of how to interact with the library. FuzzGen is built on top of LLVM and relies on having access to source code, so it has insight into primitive and complex data types and knows where and how they are initialized and used. FuzzGen prunes a binary's CFG to remove basic blocks (BB) to create a graph that only has one target library function call per node. It also encapsulates the data dependency between the function calls. FuzzGen calls the resulting graph an Abstract API Dependence Graph (A2DG). This graph, in essence, is a topological ordering that describes how to initialize data types and how to sequence the target library API calls. This mimics how a human would analyze how to set up a fuzzing harness after manual analysis.

To make use of the whole-system analysis, FuzzGen coalesces A2DGs from different binaries by merging common nodes (i.e., same function call with the same data types). This process is done starting from the root node of each graph. If a node is merged then its descendants are also migrated from one graph into the merge destination graph. Migrated nodes can also be merged later and topological ordering is maintained while merging nodes between graphs. If there are no common nodes, then the two A2DGs are kept separate. The coalesced A2DG builds a more complete picture of how to interact with a library's interface as it was intended.

FuzzGen creates fuzz test harness stubs using the coalesced A2DG. It first flattens the A2DG by creating a direct acyclic graph (DAG) and grouping nodes topologically by hierarchical level in the graph. While traversing the graph, FuzzGen randomly selects a node from a single group of nodes, and it ties the nodes together using the data dependency information encoded in the A2DG nodes. Fuzzing stub headers and linking information are also extracted from the A2DG.

Use Cases and Limitations

FuzzGen is currently limited to source-code based analysis, and it works best when it has a corpus of programs that interface with the fuzz target. These programs must be compiled into LLVM bitcode. The intuition that drove the creation of FuzzGen seems reasonable. Each stage in the FuzzGen workflow is essentially how a human would perform the work when investigating a single binary.

It is likely that there is a version of the tool that works as was claimed in the paper; however, the version that is publicly available on GitHub.com[38] seems to be brittle. Tooling to create the FuzzGen components could be more reliable, and the results in the paper could be made more easily reproducible. For example, the paper claims that the tool has been successfully used on Debian source packages (but AOSP is a first tier target); however,

the project maintainers say that using FuzzGen on Debian source is ”an exercise left to the reader” because FuzzGen is ”... research-quality code that requires some hand holding and very specific LLVM and system versions to compile (and likely a complete rewrite for production).” Currently, it may not be worth the time investment to make the FuzzGen tooling and to get FuzzGen to work on code outside of the AOSP. That said, it may be possible to get FuzzGen to work on other projects that build from source such as Buildroot or Yocto Project. It may also be possible to extend FuzzGen for binary-only fuzzing; however, it will suffer from the imprecision of missing or incorrect data type recovery and incomplete control flow graph recovery.

3.5 Ghidra Debugger

Reference Link	https://github.com/NationalSecurityAgency/ghidra
Target Type	Binary
Host Operating System	Windows, Linux
Target Operating System	Windows, Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	05/2021
License Type	Open-Source
Maintenance	Maintained by the National Security Agency

Overview

Since late 2020, Ghidra's native debugger has been available from the Ghidra repository [44]. As of May 2021, the debugger tool is now included in the Ghidra 10.0 Beta release.

Design and Implementation

The Ghidra debugger is a front-end for external debuggers, allowing users to drive debugging sessions directly from Ghidra. Ghidra debugger currently supports user-mode Windows applications on 64-bit hosts via dbgeng.dll/WinDbg and 32-bit applications via WoW64. Additionally it supports Linux applications on amd64/x86_64 hosts via GDB, including 32-bit i686/x86 applications [45]. Ghidra debugger works by connecting to the local gdb or dbgeng.dll agent over Ghidra Asynchronous Debugging Protocol (GADP).

A review of the Ghidra interface is as follows:

Objects The Objects panel allows the user to interact with the connected debugger. Figure 3.19 displays a hierarchy that describes all the objects in a program's execution. In addition, the objects panel is where the user will enter common debugging commands such as "step over" and "resume".

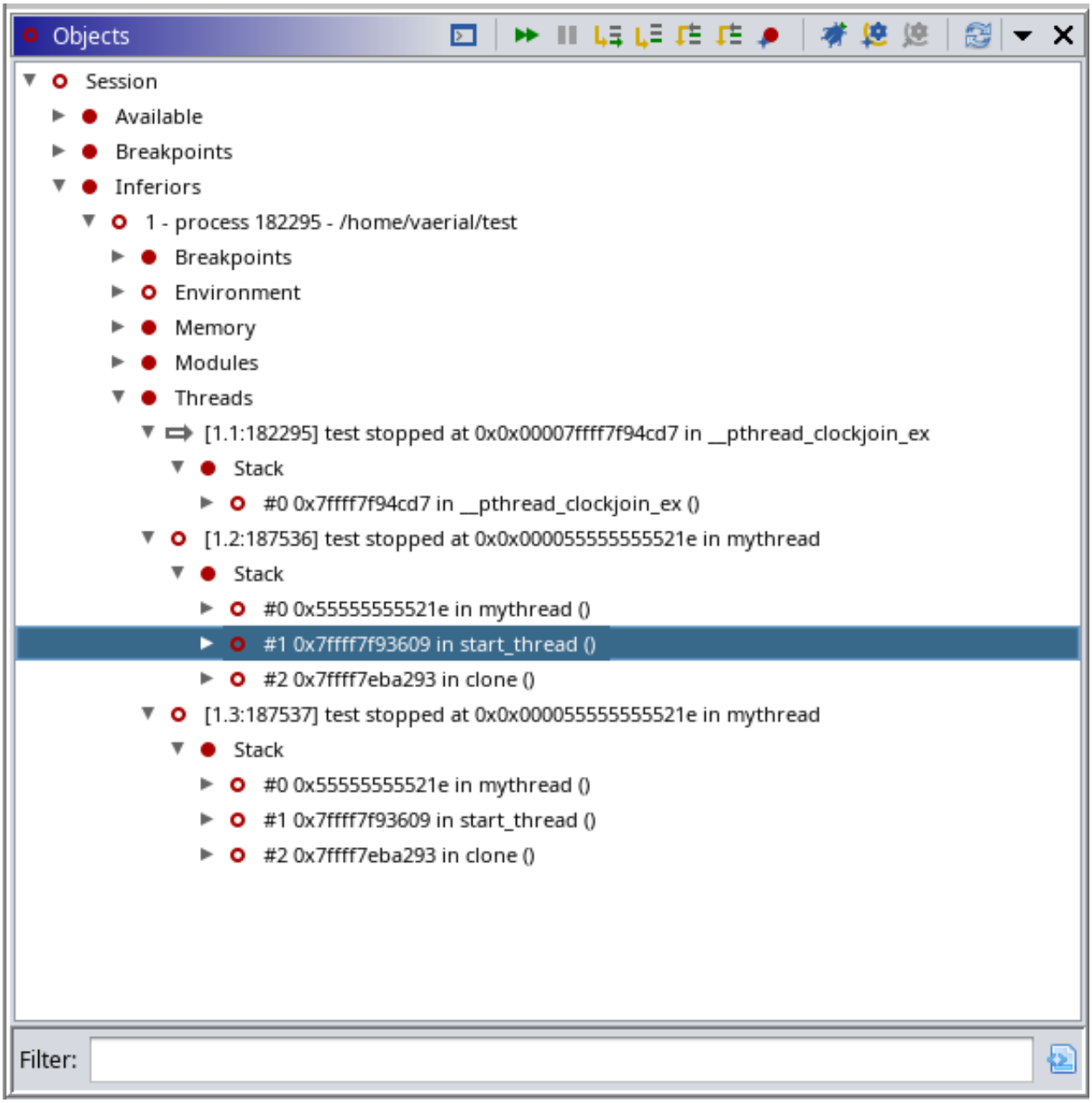

Figure 3.19: The objects panel

Dynamic listing Similar to Ghidra's static listing, the dynamic listing displays the target's memory. However, the dynamic listing will update with the memory contents of the execut-

ing program when debugging certain triggers cause debugging to pause. Just like during static analysis with Ghidra, the user may patch instructions in the dynamic listing after the program has been loaded in. Also, memory regions that have not been read yet will be greyed out (figure 3.20).

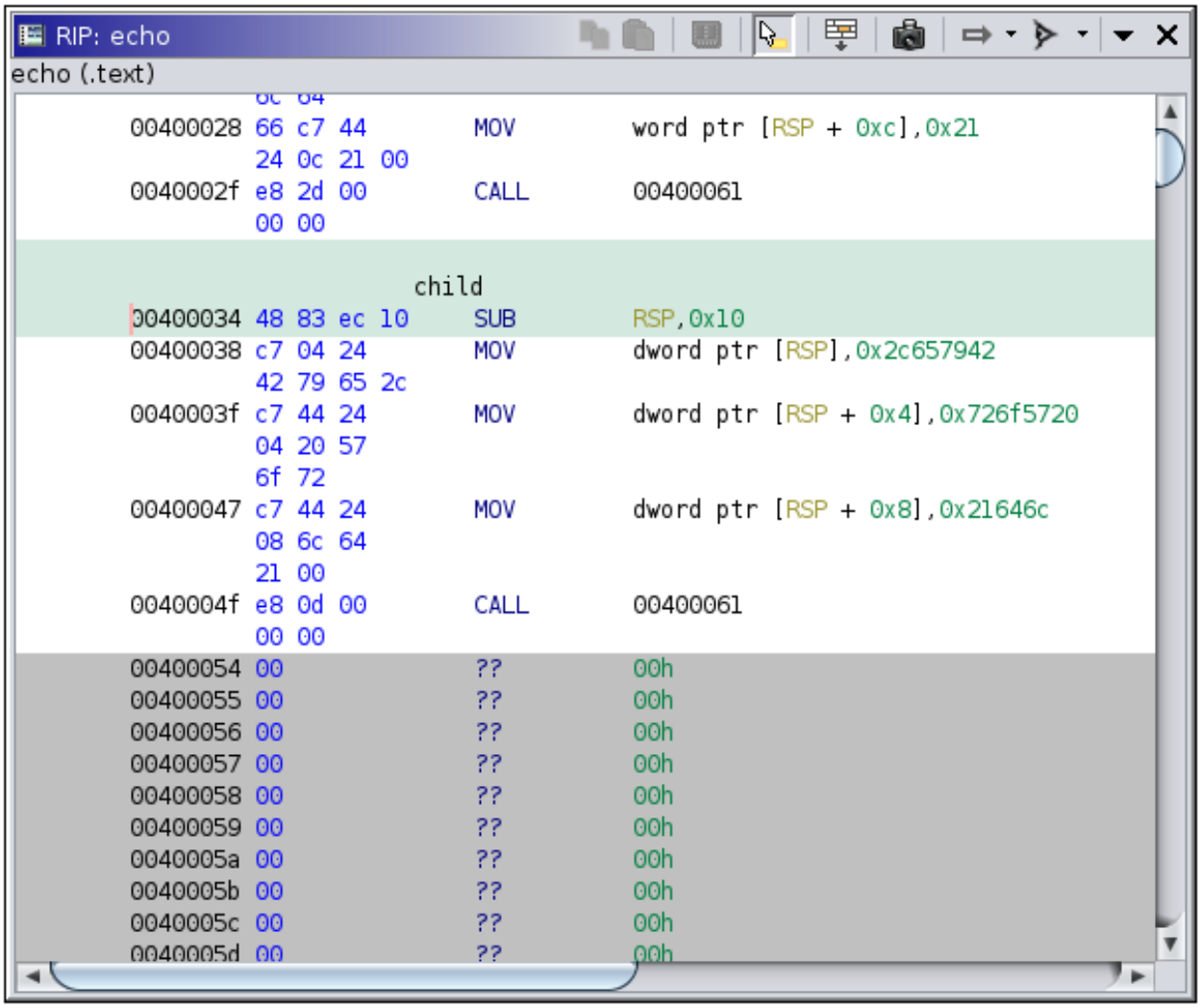

Figure 3.20: The dynamic listing panel

Conveniently, the static listing panel is synchronized with the dynamic listing panel when running the debugger. Selecting a location, current instruction, or breakpoint in one panel will update the other and vice versa. This additionally applies to the decompile pane (figure 3.21).

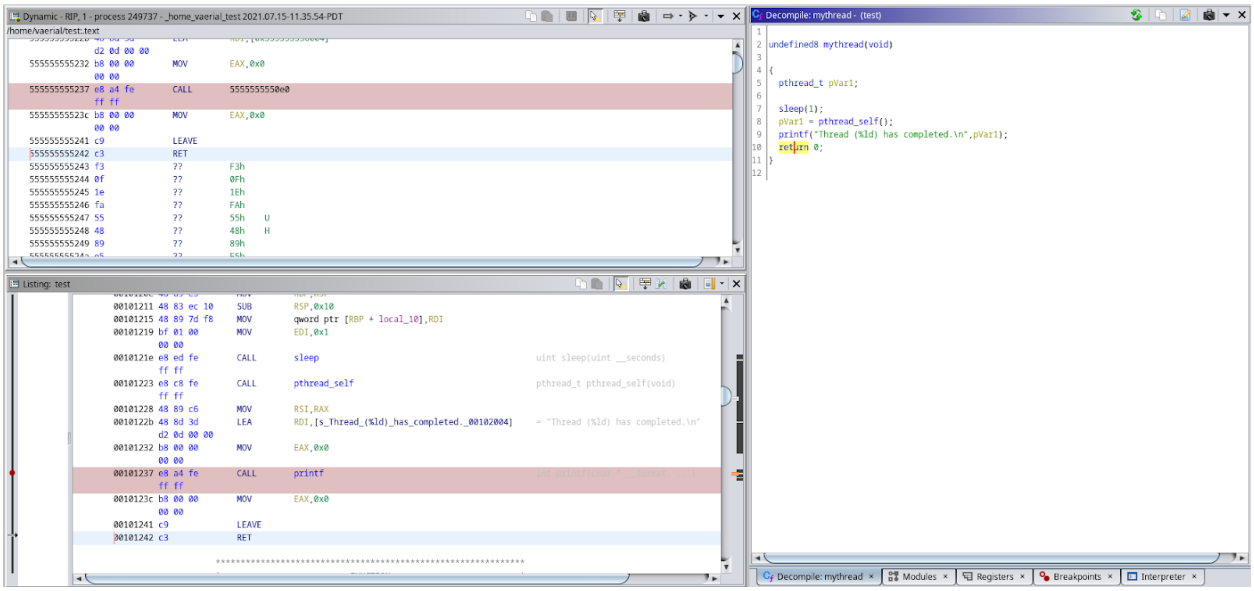

Figure 3.21: The dynamic listing, static listing, and decompile panes when selecting the return instruction and displaying a breakpoint on the printf call

Memory regions Different memory regions and their details can be found in the regions pane. These regions generally consist of pages allocated for the stack, the heap, etc. Details about these regions include when the region was created or destroyed as well as its location in memory, size, and permissions (figure 3.22).

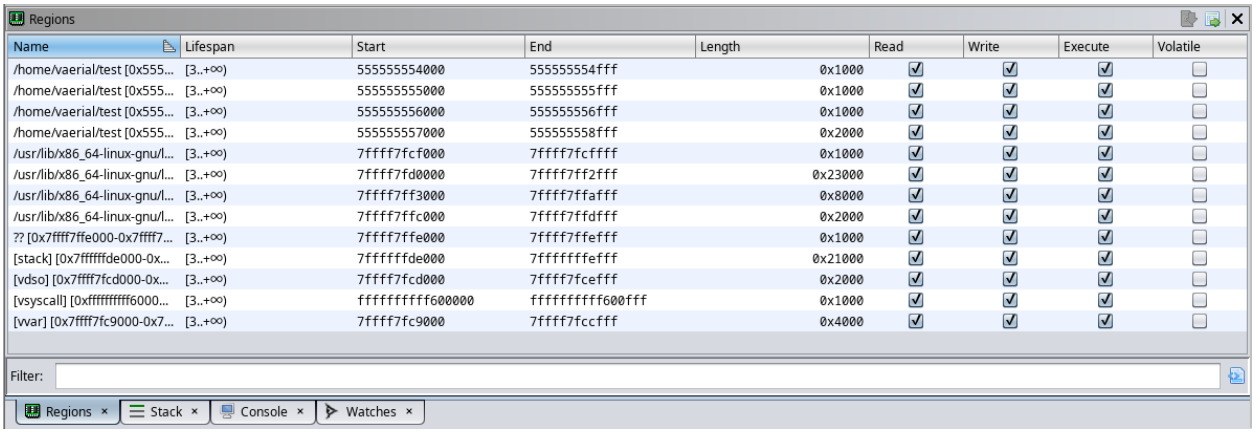

Figure 3.22: The Regions pane

Threads Ghidra's debugger allows navigating thread contexts, viewing their state, and stepping back and forth through the execution of those threads. Traces can also be saved and loaded into the debugger (figure 3.23).

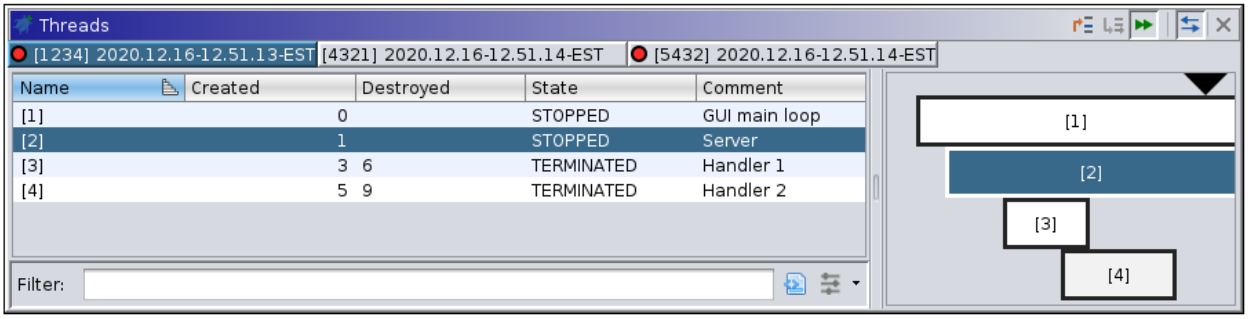

Figure 3.23: The threads panel

Traces The Ghidra debugger offers a ”time travel” functionality by recording traces and being able to rewind execution through these recorded traces. This functionality is available in the trace pane (figure 3.24).

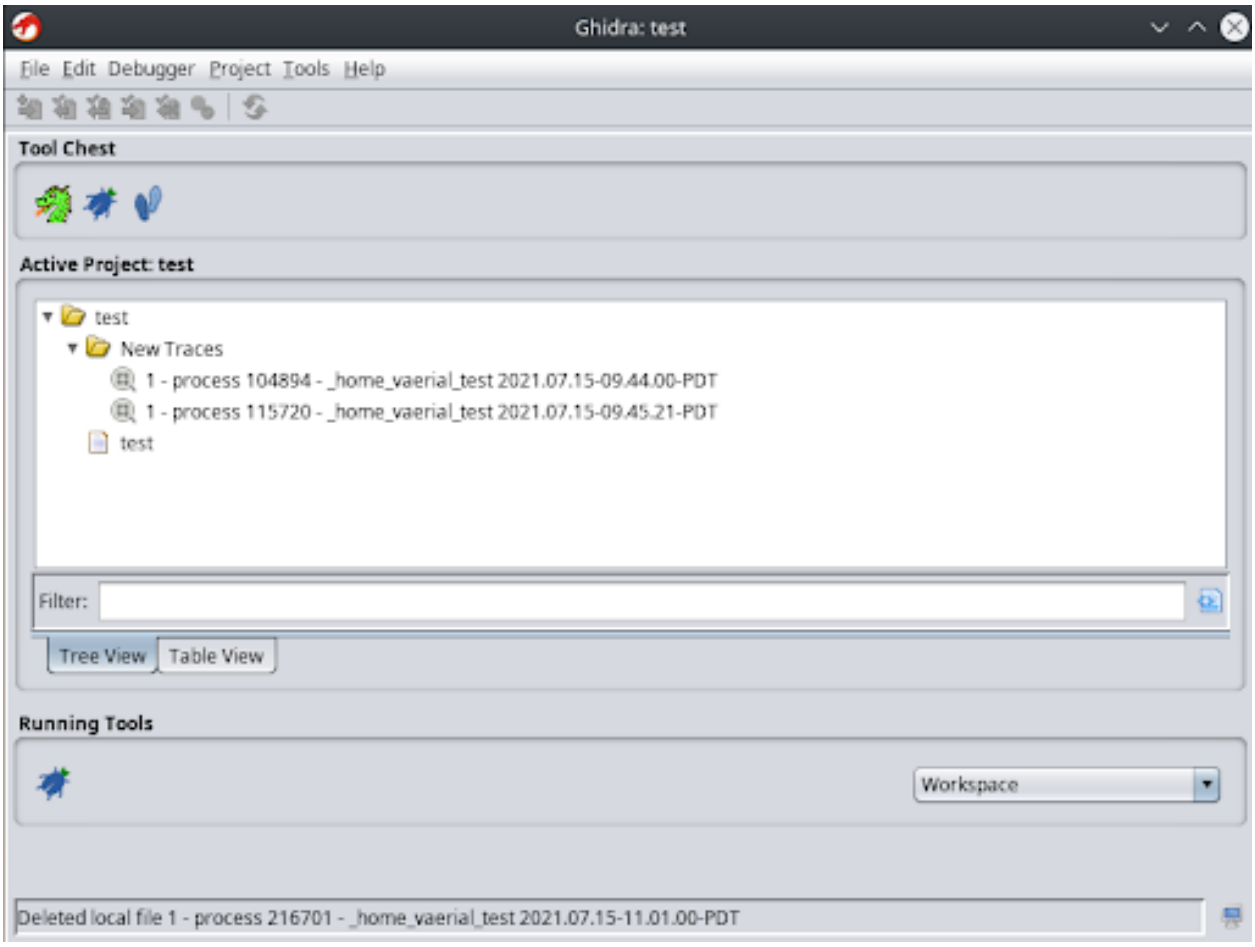

Figure 3.24: The Ghidra project window displaying saved traces

P-code debugging The Ghidra debugger also includes a Pcode stepper. (Pcode is Ghidra’s intermediate representation of machine instructions used in decompilation and other anal-

yses.) The Pcode stepper provides a means of executing instructions at the Pcode level and displaying details of the machine state (figure 3.25). This could be useful for debugging new architectures that are built using Ghidra's SLEIGH specification language.

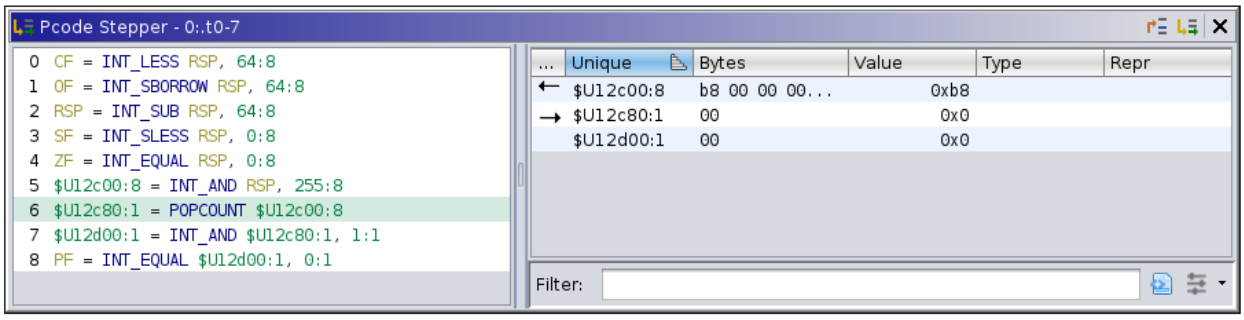

Figure 3.25: Ghidra Pcode Stepper Window

Use Cases and Limitations

Most of the Ghidra debuggers limitations stem from its relative newness. Many bugs exist, such as crashing when switching between traces. The tool is also somewhat unintuitive to use by those used to terminology used in other debuggers. For example, watchpoints are also classified as READ breakpoints. In addition, breakpoints can only be added after the target program has been mapped to memory. Another drawback is lack of support for foreign architectures. With some extra work, users may mitigate this by telling Ghidra to execute IN-VM and connect to an instance of gdb-multiarch. Additionally, there is no tracing for some architectures including, but not limited to, MIPS, PowerPC, and ARM [45].

3.6 Internal Blue

Reference Link	https://github.com/seemoo-lab/internalblue
Target Type	Binary - Broadcom Bluetooth processors
Host Operating System	Linux; Android 6-11 (jailbroken); iOS 12-14 (jailbroken); macOS (High Sierra -BigSur)
Target Operating System	Broadcom proprietary OS
Host Architecture	Python 3.6 and above
Target Architecture	N/A but generally targets ARM (32)
Initial Release	01/2018
License Type	Open-Source
Maintenance	Maintained by Seemoo Lab

Overview

Internal Blue is a python tool that enables reverse engineering and vulnerability analysis of Broadcom Bluetooth processors. It is a ”Swiss Army Knife” of capabilities. Of its many operations it can: send and monitor host to Bluetooth communication (HCI); establish Bluetooth connections; inject Bluetooth protocol packets (LMP, LCP); and invoke a variety of proprietary processor commands.

Design and Implementation

Internal Blue is designed to interface with peripheral Broadcom Bluetooth chips on a variety of host operating systems [46]. The tool itself is a python framework that can invoke a

variety of capabilities from its own command line interface. A subsection of capabilities can be seen in figure 3.26:

Figure 3.26: Internal Blue Command Line Interface and Options

The tool can run on a variety of host operating systems including Linux and MacOS. For these operating systems it uses BlueZ sockets on Linux and undocumented IOBluetooth API on MacOS. It can target handsets directly as well, though each may require unique setups whose capabilities vary by privilege and device instrumentation. Internal Blue uses Android Debug Bridge to interrogate a rooted Android device with an instrumented Bluetooth stack. It targets rooted iOS devices using a special daemon installed onto the device. In either method, the tool makes the handset listen on a TCP socket on which the user can monitor, or inject a variety of Bluetooth protocols into the Bluetooth processor.

Use Cases and Limitations

Internal Blue's main limitation is less associated with the tool itself, than with the high degree of prerequisite knowledge required for reverse engineering Bluetooth processors.

To leverage InternalBlue's most interesting capabilities, users must first modify the software under test to install instrumentation, a non-trivial process. For example, on Android, the entire Bluetooth stack must be recompiled with instrumentation, patched if it is particularly recent, and re-flashed onto the device. While documentation the Internal Blue distribution includes documentation that describes the relevant steps, it is nonetheless complicated for the uninitiated.

Fortunately, it may be the case that a researcher uses or can acquire a host device that shares a similar Bluetooth co-processor as the target. If so, the researcher can use Internal Blue on their host rather than instrument the target device itself. For Linux and BlueZ,

Internal Blue works out of the box.

Three complicating factors include the Host Controller Interface (HCI) protocol, Link Management Protocol (LMP), and Link Controller Protocol (LCP). Fortunately, Internal Blue abstracts the required knowledge of these protocols for Broadcom chipsets. Complex and proprietary capabilities are simplified into commands directly accessible to the user.

These capabilities include: reading and writing RAM and ROM as shown in figure 3.27, patching 4 bytes of data at a specified addresses, reading entire memory spaces, adding debugging breakpoints, and many more. Researchers can also inject Bluetooth traffic to send data to connected peripherals.

Internal Blue's novelty is that it greatly lowers the barrier to entry for researching Bluetooth processors. Further benefits include simple installation, high robustness, and a responsive development team.

Figure 3.27: Reading ROM from Bluetooth Coprocessor

3.7 JMPscare

Reference Link	https://github.com/fgsect/JMPscare
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (64)
Target Architecture	x86 (64); ARM(32); MIPS (32)
Initial Release	02/2021
License Type	Open-source (MIT License)
Maintenance	Not currently maintained - Last updated 02/2021

Overview

JMPscare assists fuzzing closed-source binaries by taking fuzz results and identifying interesting branches which are never taken. Such branches would otherwise prevent the fuzzer from exploring deeper into that part of the program. By identifying these branches, the human can then modify the target or harness to force exploration beyond the existing frontier of explored basic blocks.

Design and Implementation

Presented at NDSS 2021 Binary Analysis Research (BAR) Workshop, JMPscare is composed of three main components:

1. Collecting traces from emulated executions of the fuzz target. The primary mechanism currently is unicornafl.
2. Analyzes these traces using a standalone JMPscare tool
3. Using the output of the standalone JMPscare tool to annotate conditional branches in the disassembly view of Binary Ninja and enable users to quickly patch the binary to enable the fuzzer to make faster progress over “roadblock” conditionals [47] (figure 3.28).

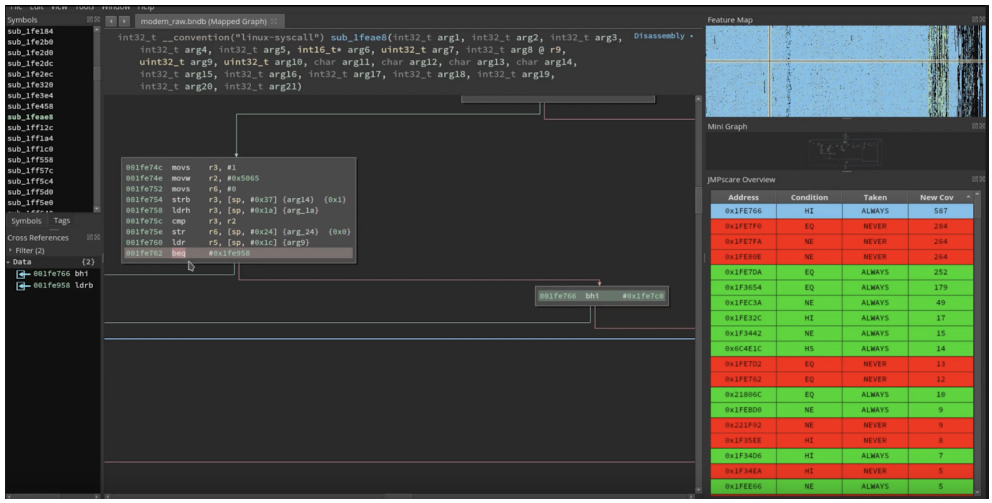

Figure 3.28: JMPscare Binary Ninja plugin identifying a “roadblock” 4-byte comparison [48]

Use Cases and Limitations

JMPscare is suited for use on targets of sufficient size to preclude easy manual analysis of fuzzer progress. In closed source fuzzing, it is often difficult to introspect fuzzer behavior using tools like Lighthouse alone. JMPscare should ideally enable the human-in-the-loop to identify high-value frontiers beyond which are completely unexplored basic blocks, as shown in figure 3.29.

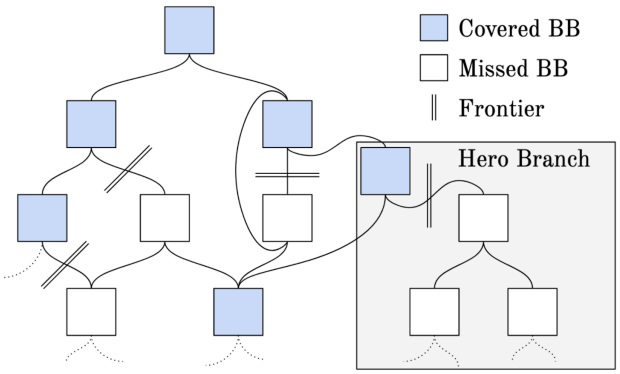

Figure 3.29: JMPscare’s goal in identifying interesting basic blocks [47]

JMPscare is still partly in the proof of concept stage. Its “Potential New Coverage (PNC) Analysis” is ARM only; the authors flatly state it is at PoC level currently. Other limitations include the fact that it does not yet work with qemu-mode. Lastly, run-time jumps are not resolved by JMPscare static analysis and instead a human assigns a placeholder weight. This is not a scalable bandaid for reachability analysis.

JMPscare has not had any commits to its repo since NDSS 2021 [49]. If this represents only a brief hiatus and active development resumes, it could become a more oft-used technique to ensure fuzzers aren’t becoming stuck on roadblock conditionals that are difficult to solve a very common problem with off-the-shelf fuzzing approaches.

3.8 KRACE

Reference Link	https://github.com/sslab-gatech/krace
Target Type	Linux Kernel
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	N/A
Target Architecture	N/A
Initial Release	05/2020
License Type	Open-Source
Maintenance	Not currently maintained - last updated 2020

Overview

KRACE is a coverage guided kernel file system fuzzer that reasons about concurrency to find data races. More specifically, KRACE uses a novel alias coverage metric (in addition to standard branch coverage) to track threaded memory access instructions that may cause concurrency bugs.

Design and Implementation

Like many other fuzzers, KRACE has two distinct steps: instrumenting and fuzzing. (See figure 3.30).

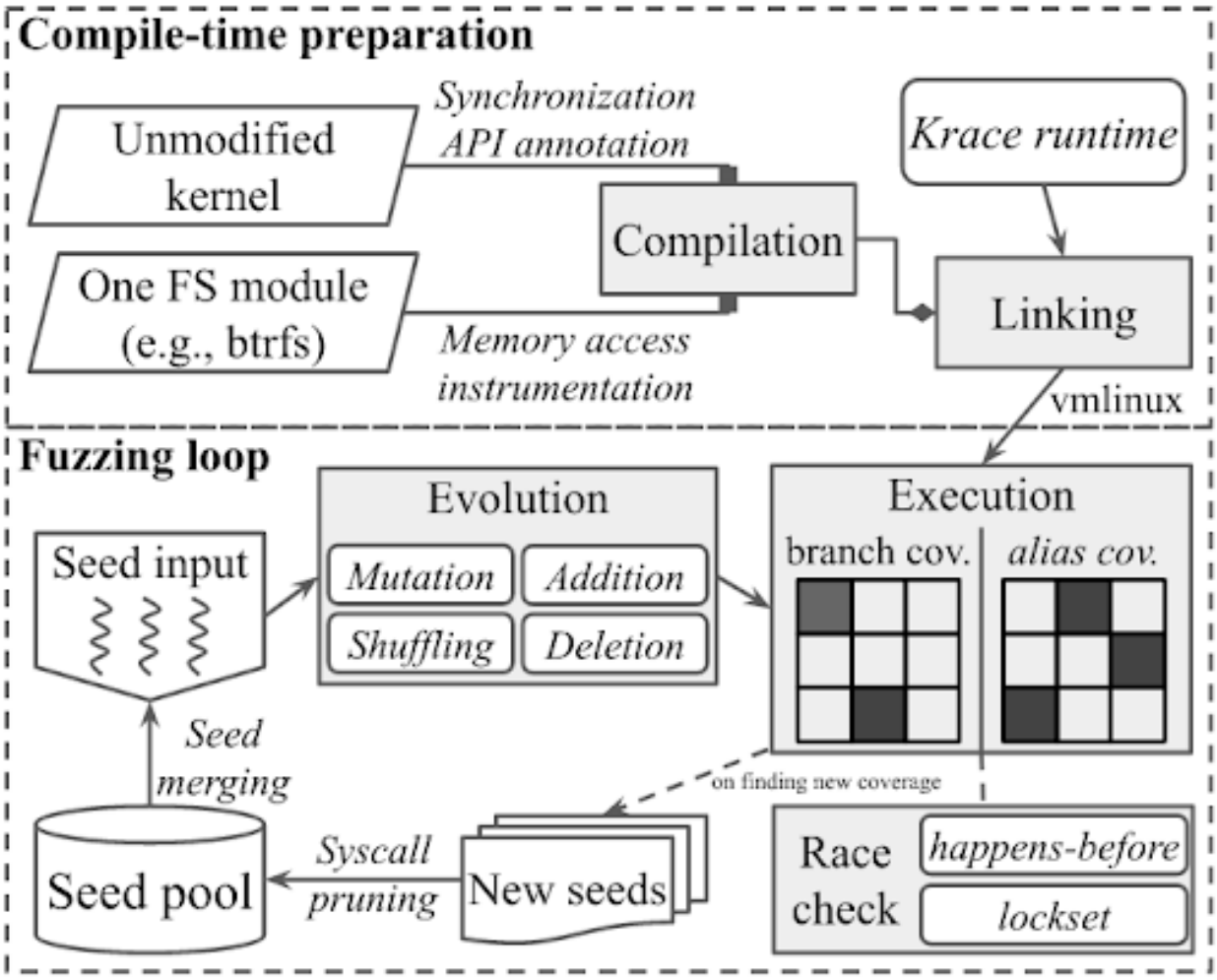

Figure 3.30: KRACE's architecture and components [50]

For the instrumentation step, KRACE instruments memory access instructions in a specified file system module along with that module's associated journaling (error correcting) module. This is carried out through kernel annotations, an LLVM [51] instrumentation pass, and the KRACE library compiled in the kernel itself. KRACE also provides coverage tracking and logging.

For the fuzzing step, KRACE first creates an execution environment consisting of multiple VM instances running inside QEMU. Each instance has a private memory mapping for test cases to be executed and uses the Plan 9 Filesystem Protocol (9P) [52] for sharing file system images and execution logs. Coverage bitmaps are stored in globally shared memory that is accessible to the host and all VM instances. This is illustrated in figure 3.31 and 3.32.

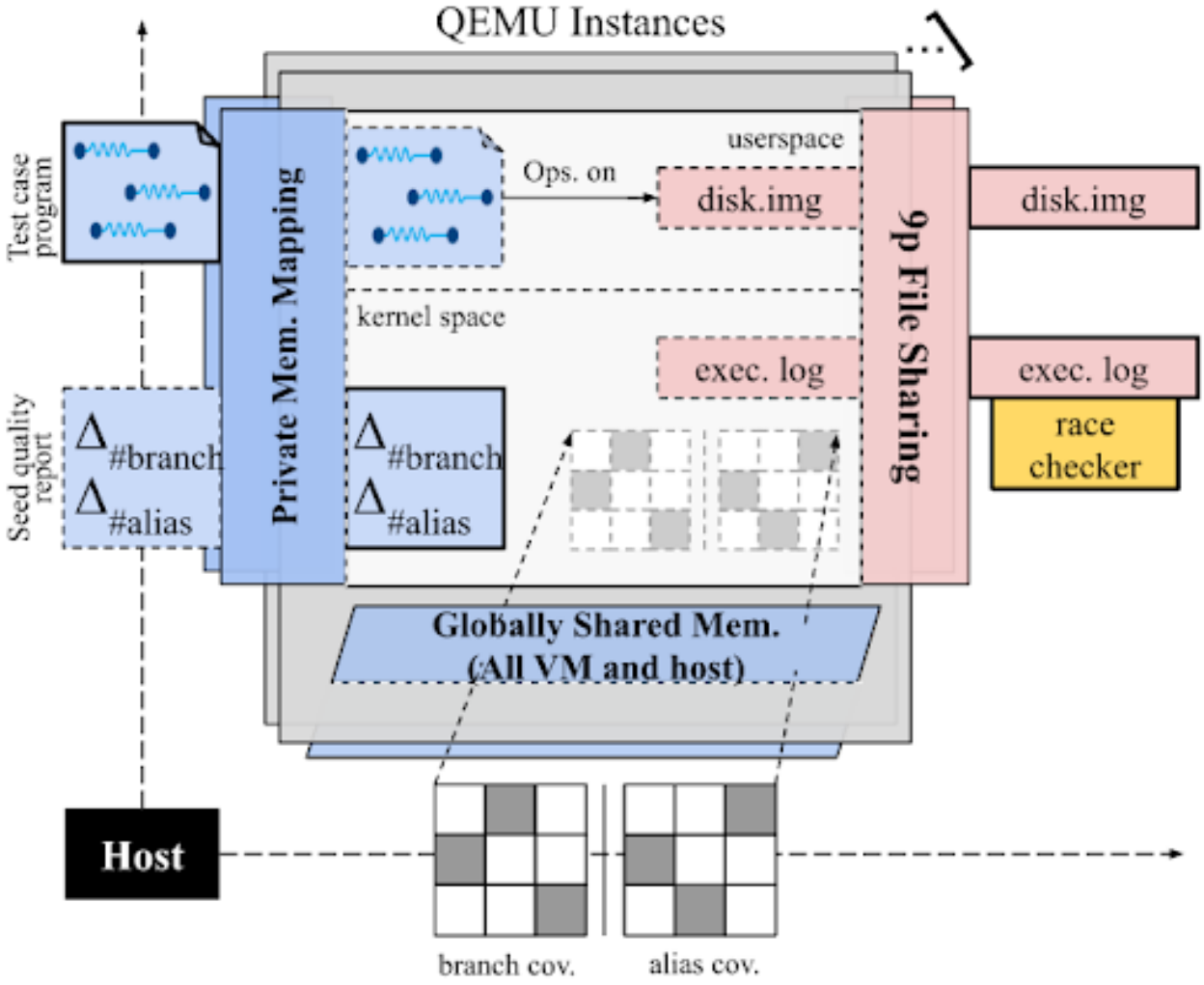

Figure 3.31: KRACE's implementation of the QEMU fuzzing executor [50]

```
ivshmem-mapped memory
|   4 MB | -> header
|   4 MB | -> cov_cfg_edge
|   4 MB | -> cov_dfg_edge
|   4 MB | -> cov_alias_inst
|240 MB | -> (reserved)
--------- (256 MB) header
|   2 MB | -> metadata (userspace: mount options, etc)
|  48 MB | -> bytecode (userspace: program to interpret)
|  12 MB | -> strace   (userspace: syscall logs)
|   2 MB | -> rtinfo   (kernel   : runtime info)
|  64 MB | -> rtrace   (kernel   : racing access logs)
--------- (128 MB) instance
```

Figure 3.32: Inter-VM shared memory, pass/dart/dart_common.h [53]

After the execution environment is set, KRACE uses branch coverage to track sequential execution paths, and alias coverage to track multiple sequences of memory access instructions in concurrent threads. During runtime, each memory address is mapped to its last write operation. When another thread reads from that same memory address, that ordered pair is recorded as new alias coverage.

The seed input for KRACE is multi-threaded sequences of filesystem related syscalls. The input arguments to each syscall are generated and tracked using a specification, where basic dependencies are maintained, e.g., closing on a previously opened file descriptor. KRACE will then mutate the seed by adding new syscalls, reordering syscalls, and modifying arguments. To produce a new seeds, KRACE will merge two seeds together while preserving the order for syscalls relative to their respective original seed (figure 3.33).

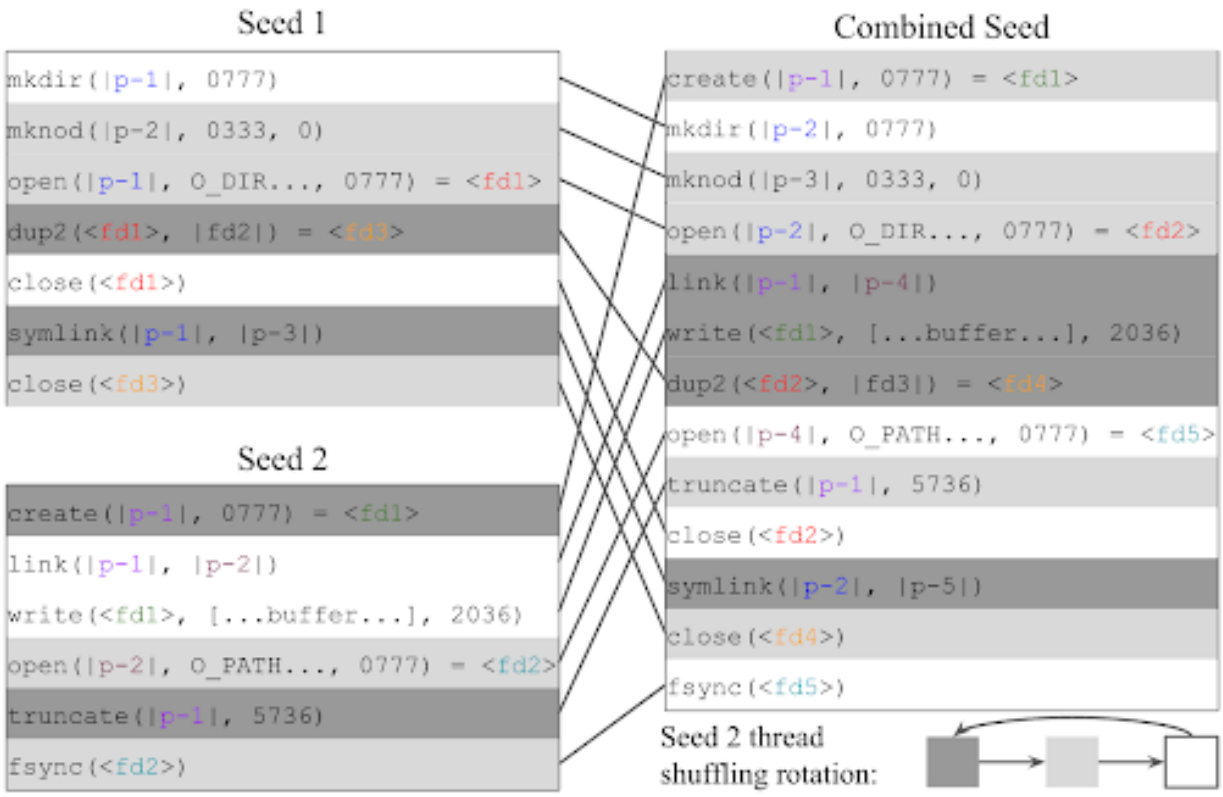

Figure 3.33: Merging two seeds of multi-threaded syscalls. Threads are marked by a shade of greyscale [50]

Given that KRACE operates in the concurrent dimension, thread scheduling may also be used as a form of input. Rather than hooking the scheduler, delay injection is used. Before execution, random integers are generated and mapped in kernel space. The instrumented code around each memory access fetches one of the integers and delays for that number of memory accesses observed in other threads across the entire system.

Whenever a new seed is discovered, KRACE performs the data race checking process in separate threads, when resources are available. This is more efficient than checking during execution, as the analysis is costly. This data race check is performed in three steps.

First involves finding the race candidates to check. This is done during runtime via the memory access hooks. A pair of memory operations is a data race candidate if, at runtime, they access the same memory location, they are issued from different contexts, and at least one of them is a write operation [50]. To remove any false negatives, a lockset analysis is performed. This checks to make sure no two contexts hold the same lock when they perform a memory operation.

Finally, KRACE recursively performs a happens-before analysis [53] to help filter out false positives. It does this by ensuring that parallel instructions involved in a race candidate are able to happen safely. KRACE performs this check by hooking kernel synchronization

interfaces dynamically.

Use Cases and Limitations

KRACE's primary use case is to find data races in ext4, btrfs, and xfs filesystem implementations. Additionally, it can find data races in the virtual file system layer.

In comparison to other fuzzers, KRACE has a significantly slower execution speed, though it compensates with superior coverage. According to the paper, it outperforms Syzkaller [54] in the coverage metric. [50]

While KRACE does not offer deterministic replay, it provides a full stack trace, conflicting source code lines, and the callback graph in its report. This is similar to other tools such as Razzer [55].

The main limitation of KRACE is that there is no documentation for building or running the project. The lack of documentation also means any missing dependencies will need to be resolved iteratively while attempting to build. Additionally, when cloning the repository, several of the submodules fail to initialize properly.

Some missing dependencies include, but are not limited to:

- sparse
- cmake>=3.13
- texinfo
- cgcc
- libattr1-dev
- libcap-ng
- nlohmann-json-dev

3.9 PyPANDA

Reference Link	https://panda.re/
Target Type	Binary
Host Operating System	Windows (under WSL), macOS, Linux
Target Operating System	Windows; macOS; Linux; BSD; etc. (anything that QEMU can emulate)
Host Architecture	x86 (32, 64) receives first-class support. Others that run QEMU, LLVM11and Python3 may work.
Target Architecture	Whole system or application binary-only analysis: x86 (32, 64); arm (32, 64); ppc (32, 64); mips (32, 64); etc. (anything that QEMU can emulate)
Initial Release	02/2021 (for the Python bindings)
License Type	Open-Source (GPLv2)
Maintenance	MIT Lincoln Laboratory, New York University, and Northeastern University

Overview

PyPANDA was presented at the NDSS BAR 2021 workshop. it integrates stages of PANDA's whole-system dynamic analysis [56] into Python. It provides Python bindings to a whole-system analysis framework and provides syntactic sugar for ancillary tasks that would normally have to be performed external to the analysis framework. This allows for more unified analysis workflows than with separate tools.

PANDA is a whole-system dynamic analysis framework. It is built on top of different technologies; however, it is powered by QEMU at its core. PANDA benefits from the flexibility of the QEMU emulator to dynamically analyze whole systems or a userspace application across multiple architectures. PANDA is packaged with a library of built-in analyzers and it has a C++ API (and now Python) for users to extend and customize PANDA's capabilities. PANDA differentiates itself from other dynamic analysis engines in that recorded concrete executions can be later replayed deterministically. This is important when analyzing binaries that interact with hardware, as hardware internal state machines and system external interactions affect hardware determinism.

Design and Implementation

PyPANDA provides Python bindings for the underlying PANDA architecture-neutral extensible dynamic analysis framework. The PANDA framework, simplistically, is the QEMU emulation engine with some additional hook points and data capture capabilities for an analyst to record and deterministically reconstruct a system state for offline analysis. PyPANDA (either as a plugin or as a standalone script) simplifies and tightens the analysis ideation and implementation loop while keeping the Python interpreter overhead to a minimum.

PyPANDA has some syntactic sugar to allow an analyst to programmatically interact with the guest system, interact with the underlying QEMU framework itself (e.g., monitor/instrumentation functions and virtual machine states), and replace GDB hooks with native Python callback functions. This is what the PANDA maintenance team calls a "unified analysis."

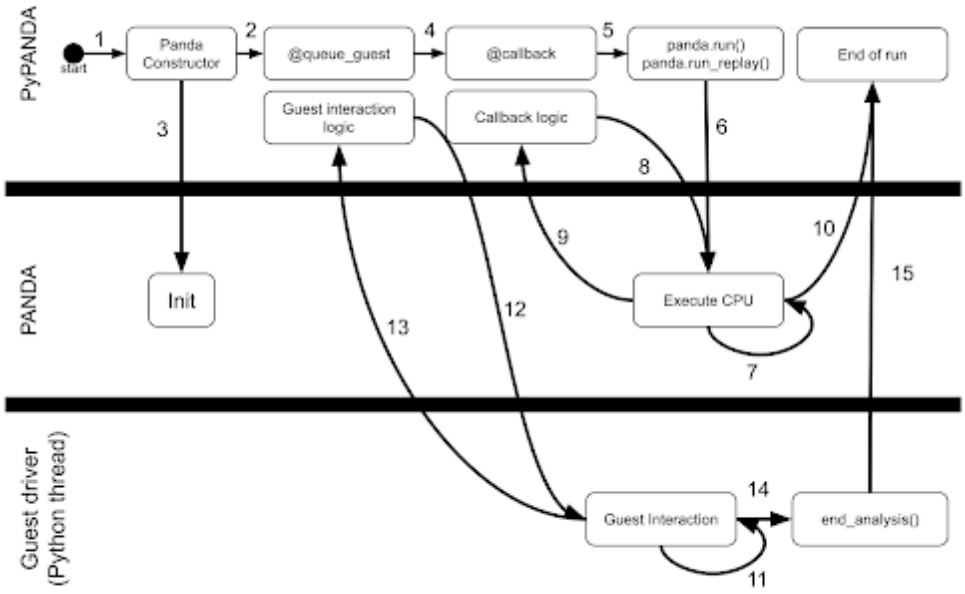

Figure 3.34: PyPANDA's Unified Analysis [56]

Some additional features of the underlying PANDA framework that are worth mentioning are the deterministic replay of a previously recorded execution, lifting QEMU TCG instructions to LLVM instructions, and a library of existing plugins (e.g., ”timeless” debugging). The determinism is especially useful for whole system analysis where hardware states may affect the overall system behavior (e.g., network packets), and the LLVM lifting and the Python bindings makes it possible to re-use existing LLVM IR analysis tools and to interface with other analysis tools with a Python interface.

Use Cases and Limitations

Deterministic Record-Replay

Other tools may be more suitable for application-level dynamic analysis, but PANDA's deterministic record and replay serves an important, niche role in dynamic analysis of binaries that interact with hardware. The determinism gives researchers a consistent baseline to check a system's security properties. To put it in terms of software engineering, PANDA makes it easier to understand transient and race condition behaviors once the behavior is recorded.

Timeless Debugging

Hand-in-hand with PANDA's deterministic record-replay is a capability called reverse (a.k.a., timeless) debugging. PANDA uses VM snapshots–each as large as the VM's memory–to enable reverse debugging. For reverse instruction stepping, PANDA restore's the VM's last snapshot and executes until it reaches the desired instruction. This seems like a PANDA design tradeoff in reducing engineering complexity at the expense of storage space. PANDA's reverse debugging seems to re-use QEMU's GDB interface. This differs from QIRA's approach of logging execution trace transactions–tracer specific–to a log file that is later ingested into a database back-end. It also differs from RR's approach of limiting traces to user-space transactions, system call inputs and outputs, and processor specific instructions (e.g., rdtsc, number of instructions executed, etc.). All of the above-mentioned approaches differ from GDB's in-memory trace logging for reversible debugging, which limits the utility of GDB reverse debugging in vulnerability research. Again, PANDA's whole-system deterministic replay capabilities differentiates its timeless debugging offering from the rest, and it also borrows some ideas from rr's GDB interface.

It may be interesting to investigate how to combine PANDA with rr's compactness or QIRA's user interface to make it even more approachable. PANDA can already analyze interactions between two processes on the same emulated machine. It would be interesting to see if (Py)PANDA can be used to analyze system-to-system interactions (e.g., an HCI device, a sensor, and a controller system).

As of April 2021, PyPANDA is unable to instantiate two Panda objects at the same time because each Python Panda object is a full instance of a Panda (C-implementation) that is loaded with a guest architecture specific shared object. Because of this, attempting

to instantiate more than one object causes some Panda internal data structures to be instantiated more than once resulting in a fatal error. The PyPANDA maintainers looked at potentially segregating multiple Pandas into different namespaces; however, attempts to use dlmopen have not been fruitful.

Non-Intel architectures and non-PANDA generic platforms are not as well supported because of the level of effort in wrapping all possible non-deterministic devices for PANDA's deterministic record-replay. This upfront manual effort may be alleviated if more people contribute the to project and if tooling or documentation is made available.

It should also come as no surprise that PANDA's record-replay functionality is separate from QEMU's record-replay, and they do not mix well together. Additionally, controlling the QEMU guest machine clock rate with QEMU's `icount` and other similar command-line arguments has the unfortunate side-effect of making a PANDA recording not replay-able.

The PyPANDA maintainers estimate a runtime overhead of well-engineered Python plugins at 110 per cent that of a native C-compiled plugin. They have suggested ways of mitigating the problem, but writing a PANDA native plugin may be worth the effort if performance is an issue.

PyPANDA relies on QEMU, which is a great basis for system-level emulation of generic devices. However, for custom or proprietary systems, there is an upfront cost of identifying all emulated hardware components needed for the analysis target. This is referred to as the ”re-hosting” problem. Using generic devices, if they work, is a time saver. The time cost to partially or to fully emulate hardware for an analysis would need to be evaluated against the potential payoff or the potential re-usability of the emulated device.

Appendix

This appendix provides additional background information on the Edge of the Art project as well as more in-depth discussion of vulnerability research technologies. This information gives context to the rest of the document, can provide useful information to those new to the field, and should remain largely the same from one EotA edition to the next.

4.1 Resources

Staying current with the ongoing advancements of such a fast-moving field requires constant engagement with the cyber security community. The contents of this report are drawn from four specific areas of engagement:

- **Social Media** - Participating in social media platforms, including online forums and chat applications, to identify key influencers, build relationships, and identify new research directions.
- **Online Code Repositories** - Monitoring code repositories for new tools and deciding when a tool has reached a baseline level of maturity for our team to evaluate and include in our toolset.
- **Top Security Conferences** - Attending a selected set of top cyber security conferences that focus on VR, RE, and program analysis to provide a formal venue for learning and exchanging new techniques.
- **Academic Literature** - Surveying academic literature frequently to ensure complete coverage of novel algorithms and approaches driven by academic research.

4.2 Tools Criteria

The following criteria govern which tools are included in this report:

- **Year Released** – “Cutting edge” has an obvious temporal component, but it is less obvious where the cut-off should lie. Every tool in this report has been introduced

within the last five years (i.e., first released in 2016 or later). Those released earlier are included either because they have significantly matured since their initial release and now contain notable features or have otherwise recently become of increased interest to the community.

- **Capability** – New tool capabilities, and how they compare to the current state-of-the-art, are a primary consideration for inclusion in this report. The novel aspect of a new tool capability is dependent on the category of tool, and each section of this report starts with an introduction that lays out its specific considerations.
- **Theory and Approach** – Tools which offer novel ideas, approaches, or new research are important even when the tools have poor implementations or do not necessarily outperform the current state-of-the-art.
- **Usability** – In contrast with *Theory and Approach*, Usability considers tools which may not represent groundbreaking research, but enable the user to harness existing capabilities more effectively.
- **Current State-of-the-Art** – The line between edge-of-the-art and state-of-the-art is hazy. There is rarely a single moment where a tool or technique definitively transitions from one category to another. In some cases, including a tool that one might consider state-of-the-art is necessary to compare to the edge-of-the-art. In other cases, the tool has new capabilities which keep it on the edge-of-the-art.

4.3 Techniques Criteria

Most techniques are implemented by at least one tool and are documented in that tool's description.

Workflows – One area of techniques that is complementary to (rather than implemented by) tools is that of workflows. This includes techniques that define effective strategies to better leverage existing tools or improve the performance of teams of analysts.

4.4 Tool and Technique Categories

There are many ways to categorize the tooling and techniques used for vulnerability discovery and exploitation. Cyber Reasoning Systems (CRS) tend to view the problem as a combination of analytical techniques, such as dynamic analysis, static analysis, and fuzzing. These analytical techniques are a bit too broad to use as tool categories because each technique summarizes a set of actions that are performed by different tools. Some tools may utilize multiple analytical techniques and thus fall in multiple categories. Alternatively, existing tool categorizations, like the Black Hat Arsenal tool repository, are both too specific (e.g., "ics_scada"), or include categories that are irrelevant to VR, RE, and exploit development (e.g., "phishing").

The CHECKMATE team has adopted a tool categorization that encompasses the VR and exploit development process followed by most researchers. Broadly, this process involves three overarching steps: 1) find points of interest (PoI) that may contain a vulnerability; 2) verify the existence of a vulnerability at each PoI; and 3) build an input that triggers the vulnerability to generate a specific effect (e.g., crash, info leak, code execution, etc.). As part of this process, the researcher will typically engage in six types of activities: Comprehension, Translation, Instrumentation, Analysis, Fuzzing, and Exploitation. These activity classes form the basis for the tool categorization used in this report.

4.5 Static Analysis Technical Overview

4.5.1 Disassembly

An assembler converts a program from assembly language to machine code, and a disassembler performs the reverse: it converts machine code to assembly language. Since there is often a one-to-one correspondence between machine instructions and assembly instructions, this translation is much less complicated than decompilation. However, disassembly can pose challenges, especially with architectures like x86 which have variable length instructions. When overlapping sequences of bytes could themselves be valid instructions, one cannot just disassemble an instruction at random. Several approaches to disassembly address this challenge, including linear sweep (which disassembles instructions in the order they appear starting from the first instruction) and recursive descent (which disassembles instructions in the order of their control flow) [57]. Many popular disassemblers including IDA Pro [4] and Binary Ninja [58] use the latter technique.

Disassembling machine code is often the first step in binary analysis. There are currently a variety of disassemblers available, ranging from simple command line utilities to proprietary platforms with capabilities far beyond basic disassembly. A simple example is objdump [59], a standard tool on Linux operating systems, which given a target binary, will output its disassembly. Tools like debuggers often rely on more sophisticated disassembly frameworks like Capstone [60] which has features complementary to its core disassembler and is designed to be used via an API. The disassembly framework Miasm [61], which is a tool included in the second version of this report, can be used similarly to Capstone.

In contrast to frameworks, disassembly platforms are designed primarily for humans to analyze disassembled code through a graphical user interface (GUI). These are often sophisticated user applications which offer a significant range of features beyond disassembling code. For example, many of these applications have built-in APIs that can be used as frameworks for custom, automated analyses. Several of these tools were discussed in the first version of this report: IDA Pro [4], Ghidra [62] and Binary Ninja [58].

Reassembleable Disassembly The disassembly techniques discussed until this point are only concerned with moving from machine code to assembly, however reassembly (automatically reassembling disassembled code) has recently become an area of academic interest, in part to support static instrumentation. A 2015 paper, Reassembleable Dissas-

sembling [63], claims that at the time “no existing tool is able to disassemble executable binaries into assembly code that can be correctly assembled back in a fully automated manner, even for simple programs. Actually, in many cases, the resulting disassembled code is far from a state that an assembler accepts, which is hard to fix even by manual effort. This has become a severe obstacle [63, p. 1].” The paper presented a tool that could disassemble a binary using a set of rules that made the resulting disassembly relocatable, which they assert is the “key” to reassembling [63, p. 1]. Since 2015, this technique has been improved, notably by the creators of angr who built a reassembling tool called Ramblr [64]. More recently, the tool DDisasm [65] was introduced. (DDisasm was discussed in the first version of this report.)

Static Binary Rewriting and Static Instrumentation Binary rewriting modifies a binary executable without needing to change the source code and recompile. One use case is for binary instrumentation, which is often thought of as a dynamic technique. While many dynamic binary instrumentation (DBI) techniques exist, there are also methods for statically instrumenting binaries. Many of these rely on reassembling or relinking the binary. Retrowrite [57], a tool designed to statically instrument binaries for dynamic analysis like fuzzing and memory checking, also uses a reassembleable disassembly technique that builds on previous research. The tool LIEF [66], discussed in the first version of this report, allows the user to statically hook into a binary, or statically modify it in a variety of ways.

Intermediate Representation (IR) An intermediate representation is a form of the program that is in-between both its source language and target architecture representations. IRs may be expressed using a variety of formats. However, most often they take the form of a parseable Intermediate Language (IL), defined by a formal grammar. IRs are designed to enable analyses and operations that would be more difficult to perform on the original representation by converting it to a common form that is architecture agnostic. Different IRs have different attributes and features, depending on their intended use. For example, some transform machine code to make it human readable, others layer on additional operations making the resulting representation less readable but amenable to analyses and optimizations.

Intermediate Representations are commonly used in compilers; a familiar example being LLVM [67], the IR used in the Clang compiler [68]. LLVM is helpful as an example not just because it is well known, but because it demonstrates the range of features a well-designed IR can offer. The instruction set and type system for LLVM is language independent, which means there are no high-level types and attributes. This allows LLVM to be ported to many architectures. Although the type system is low-level, by providing type information, LLVM enables a target program to be optimized through various analyses [51]. Unlike machine code however, LLVM is designed to be human readable [51].

LLVM uses a technique called Single Static Assignment (SSA), a form common in compilation and decompilation in which each variable is only assigned a value once. SSA form enables analysis such as variable recovery but by its nature maps one instruction to many and generates output not intended for human consumption.

These traits are not specific to LLVM but are attributes of many IRs discussed in this report. Clang's compiler works by translating source code languages to LLVM, performing optimizations in this form, and then translating the LLVM bitcode to many possible architectures[69]. The Ghidra decompiler does something similar but in reverse: a binary program is first lifted (converted to a higher-level representation) to an IR called P-Code [44], on which Ghidra can perform analyses and then decompile by converting the program to pseudo source code. Therefore, Ghidra can decompile anything it can lift to P-Code, because decompilation is performed on a language agnostic IR and not the original machine language [70].

Ghidra uses SSA form in its decompilation. However, unlike LLVM, P-Code is not in SSA form by default [70]. Other IRs also have an SSA and non-SSA form, for instance, Binary Ninja's IRs offer the ability to toggle between non-SSA and SSA form [58].

SSA demonstrates one of the trade-offs that inform IR design. The developers of Binary Ninja created the charts in Figure 24 and Figure 25 to show the tension between different features of IRs.

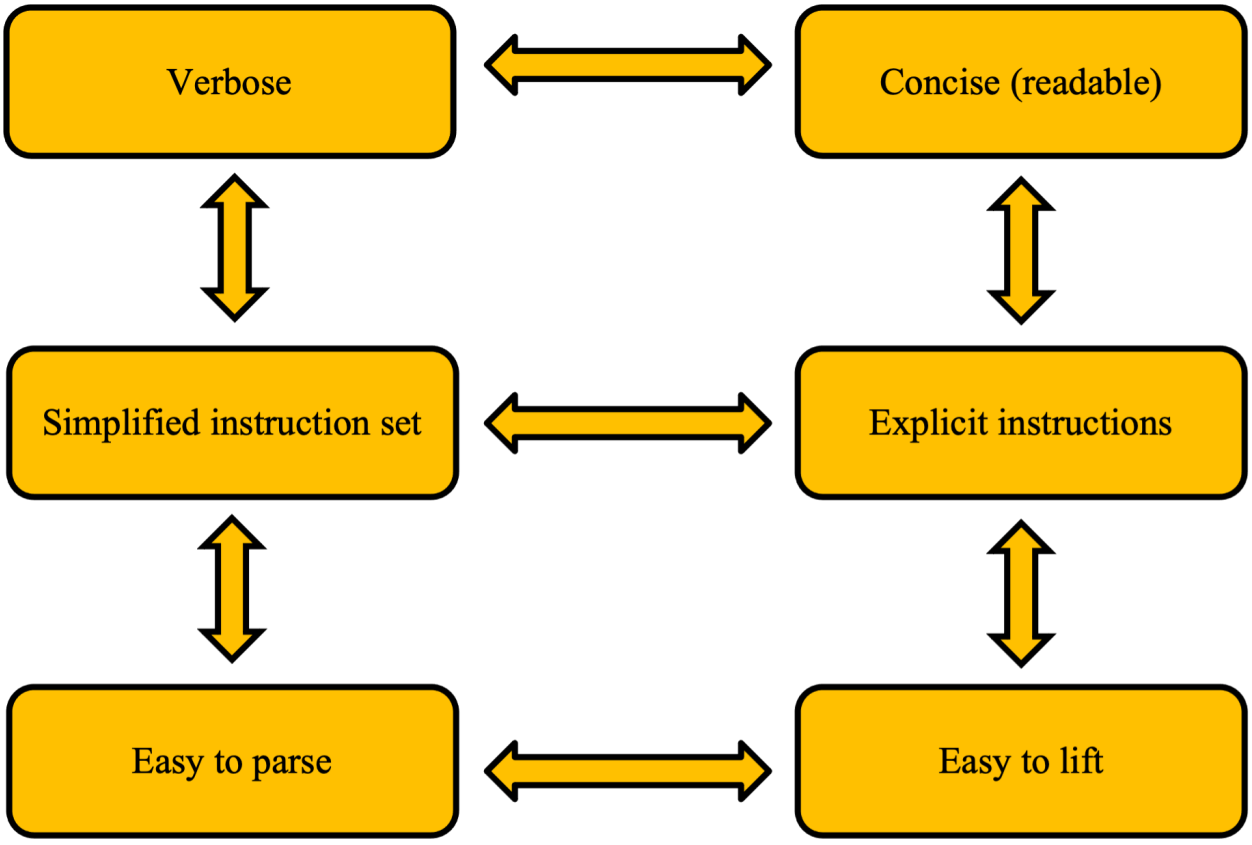

Figure 4.1: Tradeoffs of IRs, Pt. 1 [71, p. 29] The double arrows imply that emphasizing one makes the other more difficult.

Intermediate Representations, each with their own mix of features, are used extensively throughout the tools in this report. Decompilers such as IDA Pro, Ghidra and Binary Ninja (which has developed a decompiler that is not yet released) each have their own IRs. These are used not just for decompilation, but also exposed via APIs that allow the user to

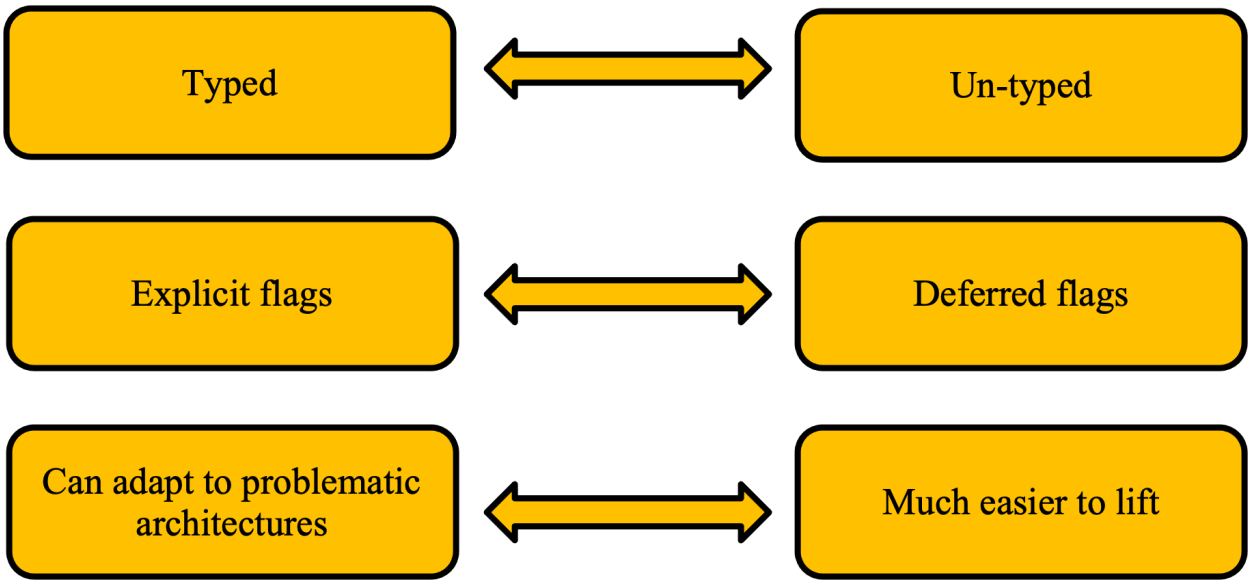

Figure 4.2: Tradeoffs of IRs, Pt. 2 [71, p. 30] The double arrows imply that emphasizing one makes the other more difficult.

utilize it for their own analyses. IDA Pro only recently documented their API [72], whereas the public release of Ghidra's P-Code included an API. However, out of these three platforms, Binary Ninja's IRs are designed with the greatest degree of user capability. They offer three levels of IRs each with an optional SSA-form and a feature-heavy API [73]. Their third level serves as a decompilation level.

Other IR frameworks discussed in the second version of this report can be used in the same manner, but each offer their own set of features. Binary Analysis Platform (BAP) [74] is a framework designed for program analysis which is built around the BAP Intermediate Language (BIL), which has a formally defined grammar [74]. Miasm has an expression-based IR that facilitates tracking memory and registry values. Miasm also has a JIT engine for emulation and has built in support for symbolic execution [61].

Certain IRs are tailored for specific use cases. For example, Fuzzilli, a fuzzer which targets Javascript JIT engines, uses a custom IR called FuzzIL. Seeds are constructed and mutated in FuzzIL then translated into Javascript before being fed into the engine [75]. This approach has the benefit of being able to theoretically explore all possible patterns given enough computing power, unlike a JIT fuzzer working from hardcoded Javascript samples.

In contrast some tools in this report use existing IRs rather than creating their own. The symbolic execution tool angr uses Vex, which is the IR implemented by the memory debugger Valgrind [76]. WinAFL, a version of the AFL fuzzer for the Windows operating system, uses DynamoRIO, a dynamic binary instrumentation engine with its own IR [77].

The variety of IRs discussed thus far show the versatility of IRs and their applications. They can be used for decompilation, semantic analysis, emulation, symbolic execution, fuzzing seed generation, and more. The abundance of intermediate representations offers a range

of choices and satisfies differing use cases, but also results in compatibility issues. GTIRB [78], which is discussed in the previous version of this report, is an IR designed to convert between different IRs. It is also the IR used in DDisasm.

4.5.2 Decompilation

A disassembler translates a program's machine code into assembly language instructions, whereas a decompiler converts a program's machine code into pseudo-code resembling a high-level language, such as C or C++. The goal of both is to transform a compiled program into a more human readable form, but the output of a decompiler is far closer to the original source code. It is significantly more difficult to create a semantically faithful representation of the underlying binary instructions in high-level pseudo-code.

Whereas compiler theory has been a popular area of computer science for decades, its reverse has received far less attention. In 1994 Christina Ciafuentes published her PhD thesis on the subject, Reverse Compilation Techniques [79]. This work went on to inform the development of multiple decompilers, including Hex-Rays, the decompiler of choice for over a decade. This tool is part of IDA Pro, a disassembler which has been commercially available since 1996, however Hex-Rays was not released until 2005 [80]. Until recently, it was one of the few decompilers available, and the most technically sophisticated.

As of 2019, the United States National Security Agency (NSA) released Ghidra, a disassembler and decompiler with comparable performance to IDA Pro [81]. In March 2020, Vector35 released a decompiler for their tool, Binary Ninja. Binary Ninja not only exposes its IRs to the user, but makes them a fundamental part of its design, with this new decompilation acting as a third layer in their three-tiered IR system. Their decompilation is available in both SSA and non SSA form.

4.5.3 Static Vulnerability Discovery

There are a number of tools and techniques intended to statically discover vulnerabilities. Many are designed for source code, including tools such as Coverity [82], CodeSonar [83], and CodeQL [84]. These use a number of static analysis algorithms to find possible vulnerabilities and common vulnerability patterns in a code base. Additionally, there are program analysis techniques designed to statically identify vulnerabilities in binary code, such as graph-based vulnerability discovery and value-set analysis (VSA) [76, p. 5].

Static Program Analysis Disassembly and decompilation, as well as static vulnerability discovery methods, are predicated on several program analysis techniques. One of the most basic forms of static analysis is pattern matching, simply scanning through code to find known vulnerabilities (e.g., using the C library function gets()). However, many of these techniques rely on far more sophisticated forms of program analysis, some of which are as follows:

Control Flow Recovery: A binary program can be broken into basic blocks separated by

branches: a basic block is a sequence of instructions that contains no jumps, except at the entry and exit. A control flow graph (CFG) models a program as a graph in which the basic blocks of the program are represented as nodes, and the jumps, or branches, are represented as edges. A CFG is instrumental to many forms of static program analysis and vulnerability discovery. Recovering one is done by disassembling the program and identifying the basic blocks and the jumps between them (both direct and indirect) [76, p. 4].

Variable and Type Information Recovery: Variable and type information is used by the compiler but is not present in final binary executable form (unless the binary is compiled to explicitly include this information for debugging purposes). Therefore, it is often necessary to recover this information when analyzing a binary [57]. One attribute of many IRs is that their lifters will recover variable and type information and include it in the IR form. This is also necessary for decomplication.

- **Function Identification:** Function information is also often left out of the final binary form of a computer program, and it is also necessary in various forms of analysis. Methods have been developed to identify distinct functions within a binary [57].
- **Value Set Analysis (VSA):** VSA is a form of static analysis which attempts to track values and references throughout a binary [76]. This analysis has a variety of uses, including identifying indirect jumps or find vulnerabilities such as out of bound accesses.
- **Graph-based vulnerability discovery:** This form applies graph analysis to a CFG to identify vulnerabilities [76].
- **Symbolic Execution:** Symbolic execution replaces program inputs with symbolic values, and then symbolically executes over the program. Symbolic execution be done either entirely statically or in conjunction with dynamic analysis. See page in the dynamic analysis section for a more in-depth discussion.
- **Abstract Interpretation, data-flow analysis, etc.:** There are many types of formal static analysis which apply mathematical approaches to program analysis. These include abstract interpretation and data-flow analysis. The tools BAP has implemented support these forms of analysis [85].

4.6 Dynamic Analysis Technical Overview

4.6.1 Debuggers

Among other uses, interactive debuggers can pause a program during execution and step through one instruction at a time, to inspect the current state of registers and memory at a specific point and see the upcoming instructions. Debuggers can be used to reverse engineering a program to determine how it operates, to inspect a crash found by a fuzzer, or to debug an exploit. Like many dynamic analysis tools, debuggers utilize both static

and dynamic techniques (e.g., the popular debugger GDB uses a disassembler and the tracing utility ptrace [86]) to implement its functionality.

Recordable, replayable debugging is one of the most powerful additions to modern debugging. This allows a user to record a program execution and then replay while debugging the process. In addition to forward debugger actions like step and continue, replayable debugging allows the user to step backwards and continue backwards, etc. TTD, a tool discussed in a previous edition, allows for replayable debugging from within the Windows debugger Windbg [87]. The tool rr [88], which was discussed in the first version of this report, enables recordable replayable debugging on Linux.

4.6.2 Dynamic Binary Instrumentation (DBI)

DBI, which underlies many dynamic binary analysis techniques, entails modifying the binary, either before or during execution, often by hooking into the binary at specific points and injecting code. DBI frameworks implement custom instrumentation which the user can access through an API to perform dynamic analysis. These include Intel Pin [31] and DynamoRIO [89], which underlie many of the tools discussed in these reports. Both can be used to drive the Windows fuzzer WinAFL [77], and the dynamic binary analysis tool Triton is built around Intel Pin [90]. DBI frameworks are implemented in a variety of ways. Intel Pin works by intercepting instructions before they are executed and recompiling them into a similar, but Intel Pin-controlled instruction, which is then executed [31]. It is analogous to Just-In-Time (JIT) compilers. DynamoRIO operates similarly in that it sits in between the application and the kernel, like a “process virtual machine,” to observe and manipulate each instruction prior to execution [89]. Other DBI options are less granular and intrusive and rely on hooking into the program through dynamically loaded libraries (e.g., this is how the tool Frida [91] operates).

4.6.3 Dynamic Fuzzing Instrumentation

Although fuzzing is discussed at length in the next section, fuzzing often requires dynamic binary instrumentation to enable input to easily and quickly be fed to the program. This can be done with various tools (Frida, Qiling, etc.) that allow the user to hook into the binary at the point of input and redirect it. The binary may also calculate checksums, or other functionality that can inhibit fuzzing; these tools can hook into the binary and redirect execution around the problematic code. The fuzzer Frizzer, reviewed in a previous edition of this report, uses Frida to instrument it.

4.6.4 Memory Checking

Memory checking, whether to find memory bugs or analyze them is a valuable form of dynamic analysis in vulnerability research. To do this, a program is instrumented such that if a memory error is triggered during runtime (e.g., an out of bounds access, null dereference, or segmentation fault) it will be recorded, along with additional contextual

information. Several tools exist to do this, such as Valgrind [92], Dr. Memory (a part of the DynamoRIO framework) and LLVM's Sanitizer Suite which includes Address Sanitizer (ASAN) [67].

4.6.5 Dynamic Taint Analysis

Dynamic taint analysis is a form of dynamic binary analysis in which data within a program (often some kind of input) are "tainted" such that their flow throughout the program can be traced. This can be done on the byte or bit-level with a tradeoff between the fidelity of the analysis and the time and memory resources required. Dynamic taint analysis is often built on top of dynamic binary instrumentation to hook into data transfer instructions to check whether the source memory or register value is tainted and then taint the subsequent destination (or conversely, remove a taint from a destination if the source lacks a taint). Dynamic taint analysis is not only useful for tracking values throughout a program, but also identifying instructions not affected by user input, which can be used for concolic execution. Triton is one tool that implements dynamic taint analysis.

4.6.6 Symbolic and Concolic Execution

Symbolic analysis is a method of program analysis which abstracts a program's inputs to be symbolic values. A symbolic execution engine "executes" the program with these symbolic values, and records the constraints placed on them for each possible path they could take. Subsequently, a constraint solver takes these constraints for a specific path and attempts to find a value which satisfies them. Consider a program which takes an input as an integer and exits if it is less than 10. That input would be assigned a symbolic value, a, and then the symbolic execution engine would record a constraint of $a < 10$ for the path that reached that exit call. Then a constraint solver would find a value for a that satisfied the path constraints, $a < 10$.

Symbolic execution can be performed "dynamically," and this is called dynamic symbolic execution, or DSE. However, throughout literature on symbolic execution there are generally two competing definitions of DSE. The first kind of DSE refers to any form of symbolic execution which "explores programs and generates formulas on a per-path basis [93]". This does not mean that only one path is followed, just that a distinct formula is generated for each path. When a branch condition is reached, and both branches are feasible, execution will "fork" and follow both possible paths [93, p. 3]. In the paper (State of) The Art of War: Offensive Techniques in Binary Analysis [76], Shoshitaishvili et al. describe this kind of DSE:

"Dynamic symbolic execution, a subset of symbolic execution, is a dynamic technique in the sense that it executes a program in an emulated environment. However, this execution occurs in the abstract domain of symbolic variables. … "Unlike fuzzing, dynamic symbolic execution has an extremely high semantic insight into the target application: such techniques can reason about how to trigger specific desired program states by using the accumulated path constraints to retroactively produce a proper input to the application

when one of the paths being executed has triggered a condition in which the analysis is interested. This makes it an extremely powerful tool in identifying bugs in software and, as a result, dynamic symbolic execution is a very active area of research. [76, p. 6]"

Symbolic execution can be combined with concrete execution in a variety of ways and this is often referred to by the portmanteau "concolic" execution. "Concolic" is another term with competing definitions but is often used as a synonym for DSE. Concolic execution can refer to the kind of DSE described in the previous excerpt, in which symbolic (not concrete) inputs are used, and all possible paths are explored, but the program execution will switch between concrete and symbolic emulation, depending on whether the instruction handles symbolic values [76].

The other common definition of DSE and concolic execution refers solely to symbolic execution which is "driven by a specific concrete execution [94, p. 6]." A program will be executed both concretely and symbolically using a chosen concrete input, and the symbolic execution will only follow the specific path taken by the concrete input [95], [94, p. 5-6]. After doing this, additional paths can be explored by negating one (or more) of the collected branch conditions for the path of the concrete input, and then solve for the new path with these negated conditions using an SMT solver in order to generate a new input [94, p. 6]. This kind of DSE or concolic execution is often used in symbolic assisted fuzzing, also known as hybrid fuzzing, which use symbolic techniques to gain semantic insight while fuzzing a program. QSYM [96] (a fuzzer discussed in the first version of this report) is an example of hybrid fuzzing.

There are many tools for symbolic execution, including Triton and Miasm. The tool angr [76] (discussed in the first version of this report) is one of the most popular, publicly available tools, and uses emulation to perform symbolic execution.

While symbolic execution does provide powerful insights into program semantics, it is greatly limited by space and time complexity issues. Path explosion is one of the challenges in symbolic execution: unbounded loops might result in exponentially many new paths. Symbolic execution is also hindered by the memory needed to store a growing number of path constraints. It is also difficult to apply to real world systems, because system calls and library calls can be hard to manage with symbolic values, among other environmental concerns [94]. Additionally, constraint solving is often a difficult and time-consuming task. As such, symbolic execution is in many cases not a feasible option or must be constrained to a small area of the program.

Constraint Solving Symbolic execution relies on the ability to solve for the collected path constraints, which is a challenging problem. These constraints can be modeled by satisfiability modulo theories (SMT) which generalize the Boolean satisfiability problem (SAT). SAT is an NP-complete problem which looks for a set of values which will satisfy the given Boolean formula. An SMT formula models a SAT problem with more complex logic that involves constructs like inequalities or arrays, whereas a SAT formula is limited to the realm of Boolean logic. While SAT solvers perform well on some problems, because SAT is NP-complete, some problem instances remain out of reach, limiting their scalability.

Bibliography

[1] Jeffball and A. Lin. (2020) Automated struct identification with ghidra. [Online]. Available: https://blog.grimm-co.com/2020/11/automated-struct-identification-with.html

[2] Cisco-Talos. (2021) Ghidraaas. [Online]. Available: https://github.com/Cisco-Talos/Ghidraaas

[3] Cisco-Talos. (2021) Ghida. [Online]. Available: https://github.com/Cisco-Talos/GhIDA

[4] Hex-Rays. (2021) Ida pro. [Online]. Available: https://hex-rays.com/ida-pro/

[5] Trail of Bits. (2021) Mcsema. [Online]. Available: https://github.com/lifting-bits/mcsema

[6] ——. (2021) Remill. [Online]. Available: https://github.com/lifting-bits/remill

[7] P. Goodman. (2018) Heavy lifting with mcsema 2.0. [Online]. Available: https://blog.trailofbits.com/2018/01/23/heavy-lifting-with-mcsema-2-0

[8] C. Cadar, D. Dunbar, and D. Engler. (2020) Klee. [Online]. Available: https://klee.github.io/

[9] ——, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[10] eurecom-s3. (2021) Symcc: efficient compiler-based symbolic execution. [Online]. Available: https://github.com/eurecom-s3/symcc

[11] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[12] F. Brown, D. Stefan, and D. Engler. (2021) Sys: A static/symbolic tool for finding good bugs in good (browser) code. [Online]. Available: https://github.com/eurecom-s3/symcc

[13] ——, "Sys: A static/symbolic tool for finding good bugs in good (browser) code," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/brown

[14] K. Serebryany, “libfuzzer–a library for coverage-guided fuzz testing,” *LLVM project*, 2015.

[15] Debugging forks. [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Forks.html

[16] Y. Shoshitaishvili and S. Verma. (2019) setcanary.c. [Online]. Available: https://github.com/zardus/preeny/blob/master/src/setcanary.c

[17] Y. Shoshitaishvili and Mrmaxmeier. (2018) patch.c. [Online]. Available: https://github.com/zardus/preeny/blob/master/src/patch.c

[18] radare org. (2021) Radare2. [Online]. Available: https://rada.re/n/radare2.html

[19] rizinorg. (2020) Frequently asked questions. [Online]. Available: https://rizin.re/posts/faq/

[20] ——. (2021) rizin. [Online]. Available: https://github.com/rizinorg/rizin/releases

[21] ——. (2021) Cutter. [Online]. Available: https://cutter.re/

[22] ——. (2021) Tools. [Online]. Available: https://book.rizin.re/tools/intro.html

[23] ——. (2021) Releases. [Online]. Available: https://github.com/rizinorg/rizin/tags

[24] ——. (2021) rz-pipe. [Online]. Available: https://github.com/rizinorg/rz-pipe

[25] ——. (2021) 0.3.0 issues. [Online]. Available: https://github.com/rizinorg/rizin/issues?q=is%3Aopen+is%3Aissue+milestone%3A0.3.0

[26] ——. (2021) newshell issues. [Online]. Available: https://github.com/rizinorg/rizin/issues?q=is%3Aopen+is%3Aissue+newshell

[27] Galois. (2021) The software analysis workbench. [Online]. Available: https://saw.galois.com/

[28] ——. (2021) Cryptoproofs. [Online]. Available: https://github.com/weaversa/cryptol-course/blob/master/labs/CryptoProofs/CryptoProofs.md

[29] ——. (2021) Llvm examples. [Online]. Available: https://github.com/GaloisInc/saw-script/tree/master/examples/llvm

[30] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, “AURORA: Statistical crash analysis for automated root cause explanation,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[31] Intel. (2021) Pin - a dynamic binary instrumentation tool. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[32] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, “FANS: Fuzzing android native system services via automated interface analysis,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 307–323. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/liu

[33] Baozheng Liu. (2020) Fans. [Online]. Available: https://github.com/iromise/fans

[34] seemoo-lab. (2021) Frankenstein. [Online]. Available: https://github.com/seemoo-lab/frankenstein

[35] J. Ruge, J. Classen, F. Gringoli, and M. Hollick. (2020) Advanced wireless fuzzing to exploit new bluetooth escalation targets. [Online]. Available: https://www.usenix.org/system/files/sec20_slides_ruge.pdf

[36] seemoo-lab. (2021) Polypyus. [Online]. Available: https://github.com/seemoo-lab/polypyus

[37] (2021) Universal asynchronous receiver-transmitter. [Online]. Available: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[38] (2021) Fuzzgen: Automatic fuzzer generation (github). [Online]. Available: https://github.com/HexHive/FuzzGen

[39] (2021) Deepstate (github). [Online]. Available: https://github.com/trailofbits/deepstate

[40] P. Goodman and A. Groce, "Deepstate: Symbolic unit testing for c and c++," in *NDSS Workshop on Binary Analysis Research*, 2018.

[41] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[42] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[43] M. Kelly, C. Treude, and A. Murray, "A case study on automated fuzz target generation for large codebases," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.

[44] National Security Agency. Ghidra software reverse engineering framework. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[45] ——. (2021) Ghidra 10.0 beta - what's new. [Online]. Available: https://ghidra-sre.org/releaseNotes_10.0beta.html

[46] seemoo-lab. (2021) Internalblue. [Online]. Available: https://github.com/seemoo-lab/internalblue

[47] M. Dominik and L. Seidel, "JMPscare: Introspection for binary-only fuzzing," 2021, preprint. [Online]. Available: https://bar2021.moyix.net/bar2021-preprint3.pdf

[48] ——. Ndss 2021 bar - jmpscare: Introspection for binary-only fuzzing. Youtube. [Online]. Available: https://www.youtube.com/watch?v=dtOR6UEJOYI

[49] ——. (2021) Jmpscare. [Online]. Available: https://github.com/fgsect/JMPscare

[50] M. Xu, S. Kashyap, H. Zhao, and T. Kim, “Krace: Data race fuzzing for kernel file systems,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 05 2020, pp. 1643–1660.

[51] LLVM Foundation. Llvm language reference manual. [Online]. Available: https://llvm.org/docs/LangRef.html

[52] (2021) 9p: The simple distributed file system from bell labs. [Online]. Available: http://9p.cat-v.org/documentation

[53] sslab gatech. (2020) Krace. [Online]. Available: https://github.com/sslab-gatech/krace

[54] D. Vyukov, “Syzkaller: an unsupervised, coverage-guided kernel fuzzer,” 2019.

[55] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, “Razzer: Finding kernel race bugs through fuzzing,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 754–768.

[56] T. J. Ott and N. Aggarwal, “PyPANDA: Taming the PANDAmonium of whole system dynamic analysis,” 2021, preprint. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-152/

[57] S. Dinesh, N. Burow, D. Xu, and M. Payer, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1497–1511. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00009

[58] V. 35. (2019) Binary ninja. [Online]. Available: https://binary.ninja/

[59] I. Free Software Foundation. (2019) Gnu binutils. [Online]. Available: https://www.gnu.org/software/binutils/

[60] N. A. Quynh. (2013) Capstone engine. [Online]. Available: https://www.capstone-engine.org/

[61] C. I. Security. (2019) Miasm. [Online]. Available: https://github.com/cea-sec/miasm

[62] N. S. Agency. (2021) Ghidra. [Online]. Available: https://ghidra-sre.org/

[63] S. Wang, P. Wang, and D. Wu, “Reassembleable disassembling,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15. USA: USENIX Association, 2015, p. 627–642.

[64] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, “Ramblr: Making reassembly great again,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/ramblr-making-reassembly-great-again/

[65] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[66] Quarkslab. LIEF. [Online]. Available: https://lief.quarkslab.com/

[67] LLVM Foundation. The llvm compiler infrastructure. [Online]. Available: https://llvm.org/

[68] ——. Clang: a C language family frontend for llvm. [Online]. Available: https://clang.llvm.org/

[69] ——. “clang” CFE internals manual. [Online]. Available: https://clang.llvm.org/docs/InternalsManual.html

[70] A. Bulazel. (2019) Working with ghidra’s p-code to identify vulnerable function calls. [Online]. Available: https://www.riverloopsecurity.com/blog/2019/05/pcode/

[71] P. LaFosse and J. Weins, “Modern binary analysis with il’s,” presented at the BlueHat Seattle, 2019.

[72] R. Rolles. (2018) Hex-rays microcode api vs. obfuscating compiler. [Online]. Available: https://hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/

[73] V. 35. Binary ninja intermediate language series, part 1: Low level il. [Online]. Available: https://docs.binary.ninja/dev/bnil-llil.html

[74] CMU Cylab. (2015) Carnegie mellon university binary analysis platform (cmu bap). [Online]. Available: https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf

[75] S. Groß, “Fuzzil: Coverage guided fuzzing for javascript engines,” Master’s thesis, KIT, Karlsruhe, 2018.

[76] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[77] Google Project Zero. (2019) WinAFL. [Online]. Available: https://github.com/googleprojectzero/winafl

[78] GrammaTech. (2020) GTIRB. [Online]. Available: https://github.com/GrammaTech/gtirb

[79] C. Cifuentes, “Reverse compilation techniques,” Ph.D. dissertation, QUT, Brisbane, 1994.

[80] I. Guilfanov, “Keynote: The story of IDA Pro,” presented at, CODE BLUE, Tokyo, Dec 2014.

[81] L. H. Newman. (2019) The NSA makes ghidra, a powerful cybersecurity tool, open source. [Online]. Available: https://www.wired.com/story/nsa-ghidra-open-source-tool/

[82] Synopsys. Coverity scan static analysis. [Online]. Available: https://scan.coverity.com/

[83] GrammaTech. Codesonar. [Online]. Available: https://www.grammatech.com/codesonar-cc

[84] Semmle. CodeQL. [Online]. Available: https://semmle.com/codeql

[85] I. Gotovchits. (2019) [ANN] BAP 2.0 Release. [Online]. Available: https://discuss.ocaml.org/t/ann-bap-2-0-release/4719

[86] Free Software Foundation, Inc. GDB: The GNU project debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[87] Microsoft. (2020) Debugging using windbg preview. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview

[88] Mozilla. rr. [Online]. Available: https://rr-project.org/

[89] D. Bruening. DynamoRIO. [Online]. Available: https://dynamorio.org/

[90] Quarkslab. Triton - a DBA framework. [Online]. Available: https://triton.quarkslab.com/

[91] O. A. V. Ravnås. Frida. [Online]. Available: https://frida.re/

[92] T. V. Developers. Valgrind. [Online]. Available: https://valgrind.org/

[93] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser, "Veritesting challenges in symbolic execution of java," *SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, p. 1–5, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3149485.3149491

[94] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018. [Online]. Available: https://doi.org/10.1145/3182657

[95] CEA IT Security. (2017) Playing with dynamic symbolic execution. [Online]. Available: https://miasm.re/blog/2017/10/05/playing_with_dynamic_symbolic_execution.html

[96] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun