AFRL-RI-RS-TM-2020-001
Version 4 of 4

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 4 OF 4

TWO SIX LABS

MARCH 2021

TECHNICAL MEMORANDUM

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-TM-2020-001 Version 4 of 4 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
TODD N. CUSHMAN
Work Unit Manager

/ S /
JAMES S. PERRETTA
Deputy Chief, Information
Exploitation & Operations 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

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2021	TECHNICAL MEMORANDUM	SEP 2019 - JAN 2021

4. TITLE AND SUBTITLE

Edge of the Art in Vulnerability Research
Version 4 of 4

5a. CONTRACT NUMBER
FA8750-19-C-0009

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Jared Ziegler

5d. PROJECT NUMBER
CHES

5e. TASK NUMBER
S4

5f. WORK UNIT NUMBER
01

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)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/RIGA
525 Brooks Road
Rome, NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RIGA

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TM-2020-001 Version 4 or 4

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 34204
Date Cleared: 10 MAR 2021

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This Edge of the Art report aggregates the most recent advances in vulnerability research (VR), reverse engineering (RE), and program analysis tools and techniques that Two Six Labs considers when planning for the next CHESS evaluation event.

15. SUBJECT TERMS

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

16. SECURITY CLASSIFICATION OF: UNCLASSIFED			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**TODD N.CUSHMAN**
U	U	U	UU	63	19b. TELEPHONE NUMBER (*Include area code*) **NA**

TABLE OF CONTENTS

LIST OF FIGURES

1 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 CHECKMATE 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 on page 41.

2 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 on page 43.

2.1 Static Analysis Tools

2.1.1 AngryGhidra

Reference Link	**https://github.com/Nalen98/AngryGhidra**
Target Type	● Binary ● Ghidra Plugin
Host Operating System	Linux; BSD; macOS; Windows
Target Operating System	N/A
Host Architecture	x86 (16, 32, 64)
Target Architecture	N/A
Initial Release	08/23/2016
License Type	Open-Source (MIT)
Maintenance	Maintained by *Nalen98*

Overview

As the name suggests, AngryGhidra is a tool that brings some of the power of angr to the Ghidra GUI. Specifically, it allows the user to perform symbolic execution, a static analysis technique that can be used to determine what inputs cause different parts of a program to execute.

Design and Usage

Installation is quite simple, as documented in the GitHub README. The key information to an easy installation: **You must use a downloaded Release Zip file for your version of Ghidra**. Installing from a cloned repo will not work without unintuitive changes.

Step 1: Install angr:

```
$ pip3 install angr
```

Step 2: Make sure python3 is in the PATH
Step 3: Install the Extension from within Ghidra:
File → Install Extensions → +

Step 4: Restart Ghidra and "configure" the new extension (this is straightforward).

Once the extension is installed and you have a binary open, getting started with the plugin is as easy as right-clicking on an address in the disassembly viewer and selecting the AngryGhidraPlugin submenu. From there, you can choose to **Set/Unset** key addresses and **Apply Patched Bytes**.

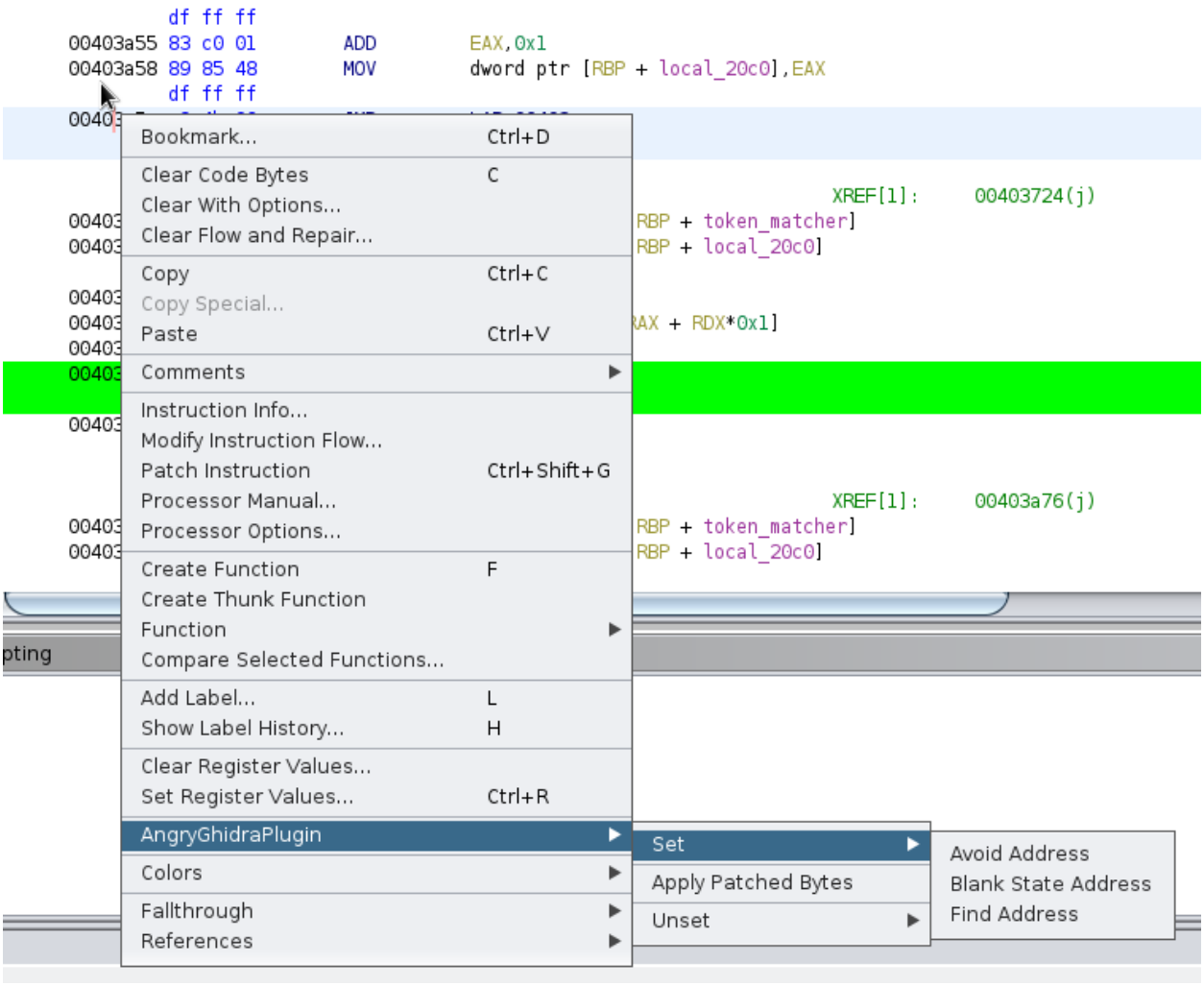


Figure 1 Screenshot from Ghidra using the AngryGhidra plugin.

If you choose to Set/Unset addresses, you can select the "**Blank State Address**", "**Avoid Address**", and "**Find Address**" fields. Activating any of them brings up the single UI panel:

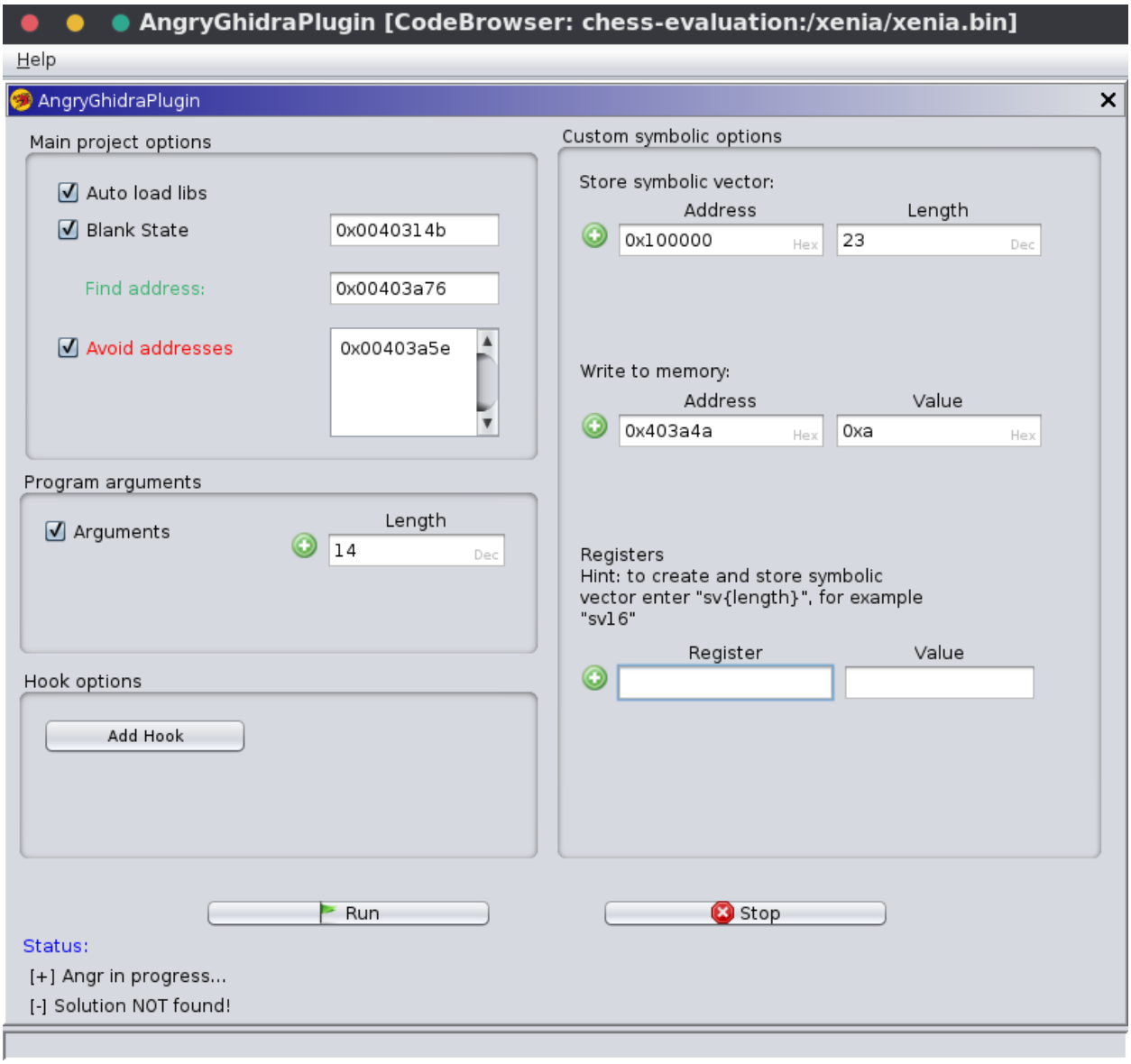


Figure 2 Screenshot from Ghidra using the AngryGhidra plugin.

- The Blank State Address indicates where to start symbolic analysis with a "blank state."
- The Avoid Address field holds multiple entries and causes angr to only analyze paths which do **not** include those addresses.
- The Find Address field allows you to determine where a "successful" solution should finish.

Additional features include the ability to set program arguments, hook addresses to modify the symbolic state, and a few other options enabling greater control over angr's analysis.

Because of the nature of symbolic execution with Z3, the output is either "**Solution:**" (including a single satisfying input that reaches the desired state) or "**Solution NOT found!**"

Use Cases and Limitations

AngryGhidra provides a subset of angr's symbolic analysis capabilities to Ghidra.

The tool provides a rudimentary interface to angr. The UI consists of a single panel and a Context Sub-Menu "AngryGhidraPlugin". Some pieces of the UI are simple to understand while others take a bit more investigation. Documentation is very limited for anything other than installation.

For simple problems, the UI makes it easy to solve problems using the power of angr. For example, the main page shows the plugin being used (in an exceedingly fast gif) to solve the "SecurityFest 2016" challenge named "fairlight". [1]

Symbolic execution works well for programs of limited size and complexity. However, it has difficulty scaling to large problems due to the path explosion problem; large programs simply have too many paths to analyze in a reasonable amount of time.

Finally, while angr supports a variety of binary formats and Ghidra supports even more, AngryGhidra will only work on the x86 family of architectures.

2.1.2 BNCov

Reference Link	**https://github.com/ForAllSecure/bncov**
Target Type	Binary (Binary Ninja Plugin)
Host Operating System	Windows, MacOS, Linux
Target Operating System	N/A
Host Architecture	x64, AArch64 (MacOS only)
Target Architecture	Any supported by Binary Ninja
Initial Release	Aug 2019
License Type	Open-Source (MIT)
Maintenance	Maintained by ForAllSecure

Overview

bncov is a plugin for Binary Ninja that provides visualization and a scripting interface for block coverage information. This provides coverage information visually, and enables scripting to link dynamic analysis information with the Binary Ninja's analysis of the target. The end result is a framework enabling automation of analysis tasks, such as understanding the progress of a fuzzer in terms of what code it was able to cover, and what interesting code the fuzzer may not have been able to reach.

Design and Implementation

bncov is a plugin that provides a simple abstraction layer over block coverage information, which must be collected separately. The key insight provided is a crucial bridge between dynamic information (what happened during execution of the target) and static analysis information (everything Binary Ninja can deduce about the target without execution). bncov's core contribution is a scripting interface that allows users to build new analyses.

Once the plugin is installed, new entries for bncov will be available in the context menu (via right-click or "Tools" dropdown) which allows users to import block coverage information. Users must generate block coverage information with tools such as Pin, DynamoRIO, or Frida, and then those files (in one of two formats) can be imported and processed.

Once coverage data is imported, basic blocks that were covered are highlighted and bncov's CoverageDB data structure are populated with the relevant information. This CoverageDB abstraction interaction with the coverage data either via Binary Ninja's built-in Python console or via scripts which can be executed either from the console or headlessly. There are a number of helper functions, but the primary data structures representing mappings from each coverage file to the basic blocks it includes and vice versa. This scripting interface provides block coverage information and linkage with static. Examples of building automated scripts on top of it are included with the repository.

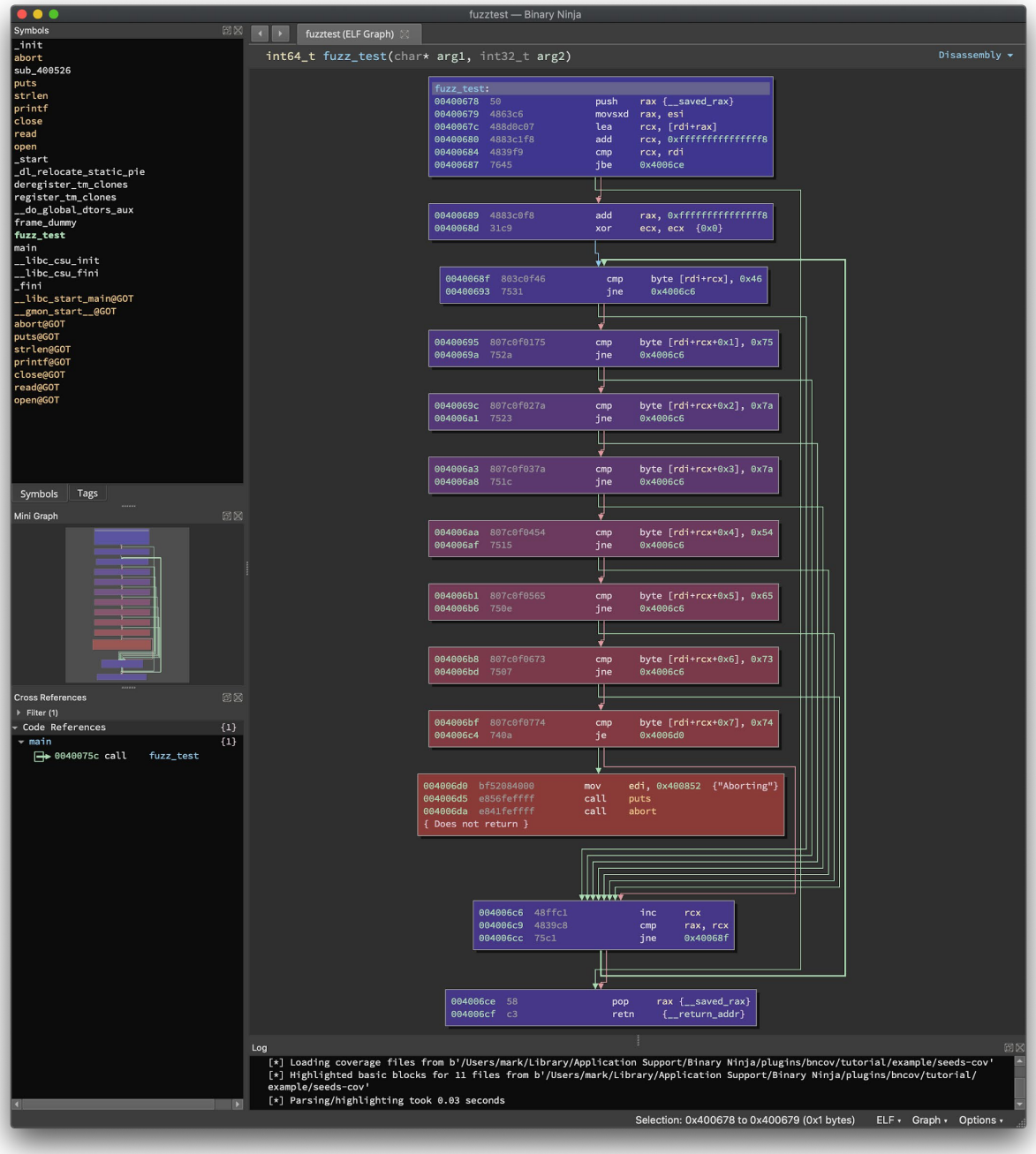
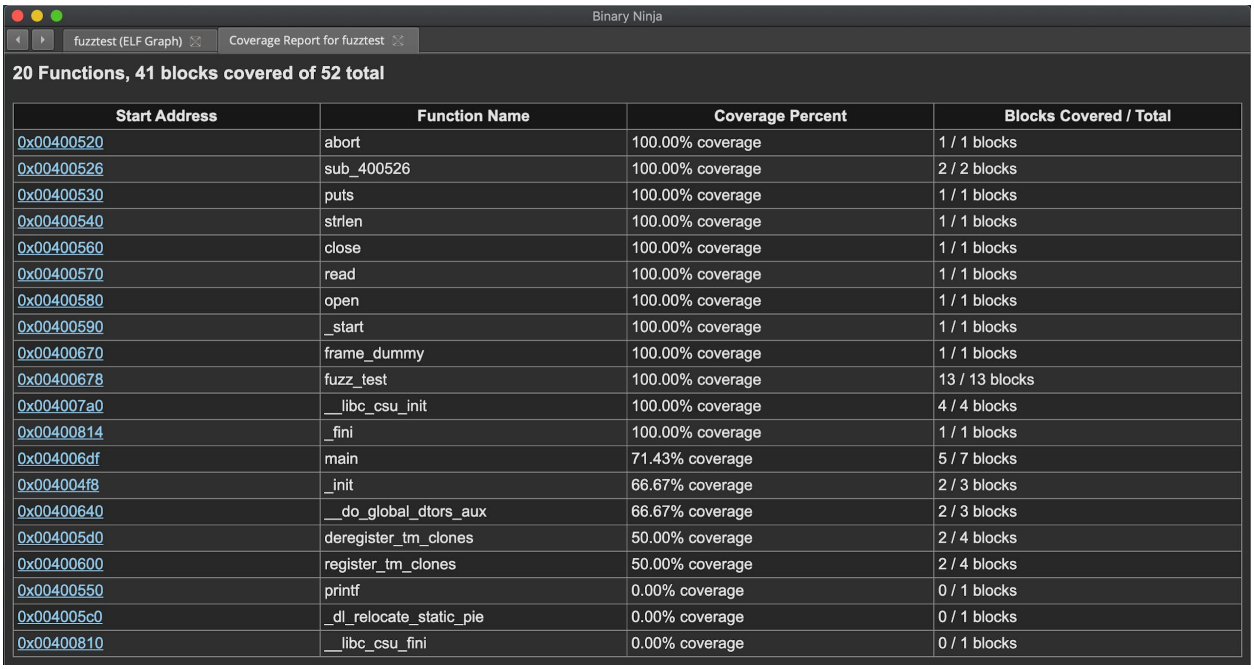


Figure 3 Screenshot from Binary Ninja using the BNcov plugin. The default highlighting scheme is an inverse heatmap: redder or "hotter" blocks have fewer inputs covering them

Use Cases and Limitations

Basic use cases involve scripting to answer common questions, such as "which functions could the fuzzer reach" or "what inputs executed this specific basic block". These can be performed within the built in Python console as part of an interactive reverse-engineering/vulnerability analysis workflow, or scripted and executed headlessly to automate report generation.

Generating reports for viewing within the Binary Ninja UI is also possible, and a default coverage report capability is included.



Start Address	Function Name	Coverage Percent	Blocks Covered / Total
0x00400520	abort	100.00% coverage	1 / 1 blocks
0x00400526	sub_400526	100.00% coverage	2 / 2 blocks
0x00400530	puts	100.00% coverage	1 / 1 blocks
0x00400540	strlen	100.00% coverage	1 / 1 blocks
0x00400560	close	100.00% coverage	1 / 1 blocks
0x00400570	read	100.00% coverage	1 / 1 blocks
0x00400580	open	100.00% coverage	1 / 1 blocks
0x00400590	_start	100.00% coverage	1 / 1 blocks
0x00400670	frame_dummy	100.00% coverage	1 / 1 blocks
0x00400678	fuzz_test	100.00% coverage	13 / 13 blocks
0x004007a0	__libc_csu_init	100.00% coverage	4 / 4 blocks
0x00400814	_fini	100.00% coverage	1 / 1 blocks
0x004006df	main	71.43% coverage	5 / 7 blocks
0x004004f8	_init	66.67% coverage	2 / 3 blocks
0x00400640	__do_global_dtors_aux	66.67% coverage	2 / 3 blocks
0x004005d0	deregister_tm_clones	50.00% coverage	2 / 4 blocks
0x00400600	register_tm_clones	50.00% coverage	2 / 4 blocks
0x00400550	printf	0.00% coverage	0 / 1 blocks
0x004005c0	_dl_relocate_static_pie	0.00% coverage	0 / 1 blocks
0x00400810	__libc_csu_fini	0.00% coverage	0 / 1 blocks

Figure 4 Basic coverage report for the Binary Ninja UI available in the plugin.

Advanced use cases involve using coverage information as a bridge between static and dynamic analysis information. For example, determining the value of a particular register or memory location across all inputs generated by a fuzzer, by using bncov to filter out which seeds actually cover that block, and then running the target under a debugger to extract the relevant runtime information for each input. Another example is first using static analysis heuristics to find locations of potential vulnerabilities, and then using bncov to identify which inputs already cover that code or to determine which inputs are nearest to reaching the vulnerable code.

Other miscellaneous use cases exist for any coverage analysis-related task. Such ideas include:

- Using debug symbols to map block addresses to lines in source files and then using bncov to determine which inputs cover specific lines of code.
- Using the coverage information to minimize a set of input files to the smallest number of files that have equivalent coverage.
- Taking multiple fuzz harnesses targeting the same library and unifying the coverage information, similar to unit test coverage typically displaying the aggregate coverage across all unit tests.

The use of this tool is limited to targets where coverage information can be collected, which is challenging on uncommon platforms, or against targets that do not run under dynamic binary instrumentation for any reason. In most cases a debugger can be scripted to produce necessary coverage information, but that is left to a user to implement. Use of bncov is also limited to architectures that Binary Ninja can disassemble properly.

2.1.3 CodeQL

Reference Link	https://securitylab.github.com/tools/codeql
Target Type	Source code: C/C++, C#, Go, Java, JavaScript, Python
Host/Target Operating System	Linux, Windows, Mac
Host/Target Architecture	x86 (32, 64), ARM, Interpreted Languages
Initial Release	November 14, 2019 (current license)
License Type	Free for academic research, demonstrations, or for OSS on GitHub
Maintenance	Maintained by Semmle (owned by GitHub, owned by Microsoft)

Overview

CodeQL [2] is a semantic code analysis engine allowing users to query source code as though it were data. This allows for the creation of powerful and reusable static analyses that can be run across multiple code bases to quickly identify vulnerable patterns of interest.

Various analysis platforms by Semmle [3] have been available since at least 2006 [4], but CodeQL has recently been generating significant interest as a platform, especially since being acquired by GitHub in 2019 [5].

Design and Usage

The first step in using CodeQL is acquiring a database to query against. Pre-built databases can be acquired for many open-source projects from LGTM.com [6], which also offers the ability to run queries directly in the browser. For other software, the CodeQL command-line interface (CLI) [7] allows you to extract relational data about your software and build a database that can be queried later. For compiled languages, this also requires a build command. Here is a command-line invocation for building a database from a source tree that can be built with the "make" command:

```
% codeql database create codeql.db --language=cpp --command make
```

Once this is done, it is possible to run one or more queries against the database. Here is the main portion of a simple query provided in the CodeQL tutorial [8] that looks for redundant "if" statements.

```
import cpp

from IfStmt ifstmt, Block block
where ifstmt.getThen() = block and
  block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```

Figure 5 Query from the CodeQL tutorial. [8]

With appropriate packaging, metadata, and the CodeQL libraries, a query can be run from the command line as follows:

```
% codeql database analyze codeql.db package/redundant-if.ql --format=csv --output=redundant-if.out --search-path=/path/to/codeql-library
```

Results will be available in the output file, in this case in comma separated variable (CSV) format.

A GitHub-supported plugin for Visual Studio Code is also available [9], which makes it easy to choose a database that's been downloaded or built from the command line.

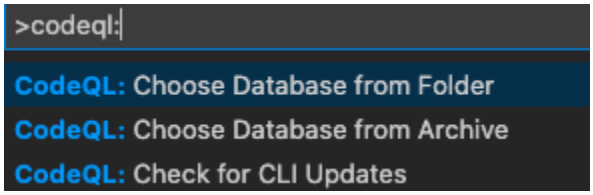


Figure 6 Screenshot from Visual Studio Code using the CodeQL plugin.

Users can also develop new queries in vscode and run groups of queries with a menu command. In the image below, we're executing some built-in queries available from the vscode-codeql-starter repository on GitHub [10].

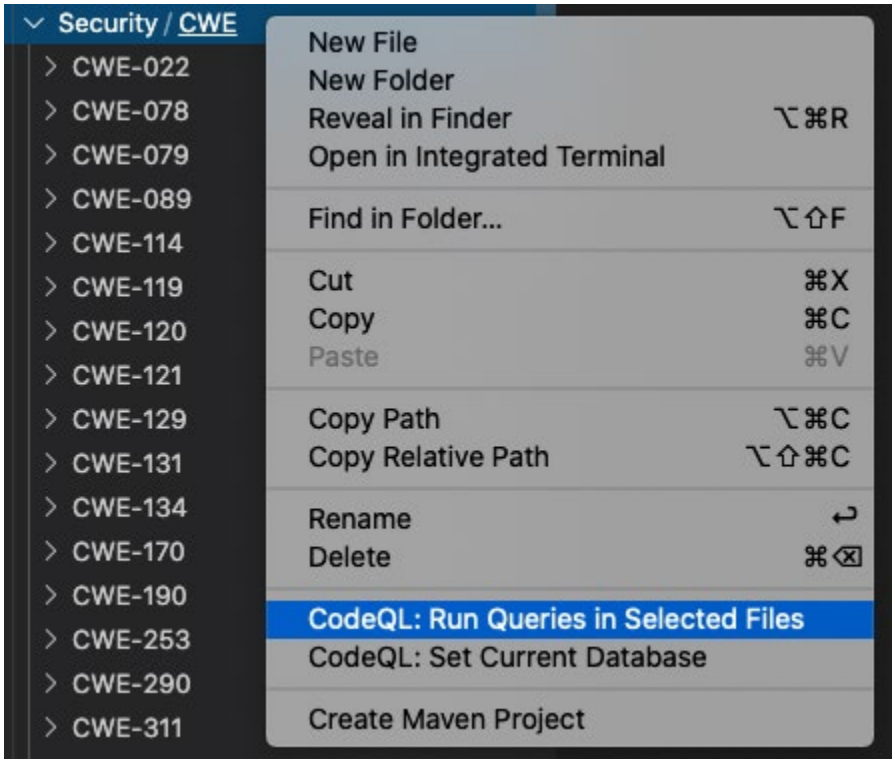


Figure 7 Screenshot from Visual Studio Code using the CodeQL plugin.

Use Cases and Limitations

CodeQL ships with a large library of tests for common error patterns as well as code snippets for building new, custom queries. For example, here are the built-in queries from the C++ "Critical" vulnerabilities section [11]:

DeadCodeCondition.ql DeadCodeFunction.ql DeadCodeGoto.ql DescriptorMayNotBeClosed.ql DescriptorNeverClosed.ql FileMayNotBeClosed.ql FileNeverClosed.ql GlobalUseBeforeInit.ql InconsistentNullnessTesting.ql InitialisationNotRun.ql LargeParameter.ql LateNegativeTest.ql MemoryMayNotBeFreed.ql MemoryNeverFreed.ql MissingNegativityTest.ql	MissingNullTest.ql NewArrayDeleteMismatch.ql NewDeleteArrayMismatch.ql NewFreeMismatch.ql NotInitialised.ql OverflowCalculated.ql OverflowDestination.ql OverflowStatic.ql ReturnStackAllocatedObject.ql ReturnValueIgnored.ql SizeCheck.ql SizeCheck2.ql Unused.ql UseAfterFree.ql

By running these existing queries, CodeQL can be used much like a traditional code quality or static security analysis tool to generate a report of potential issues. As with other static analysis tools this can include potentially many unintended or undesired findings. These are sometimes thought of as "false positives" in the sense that the finding may not be useful in terms of exploitability. [12]

The true power of CodeQL comes in the form of custom queries. As an example, here is a portion of a query for discovering uses of code segments that contain the Heartbleed vulnerability. [13]

```
from FunctionCall fc, Struct ssl3_record_st, Field data, Field length
where
  fc.getTarget().getName().matches("%memcpy%") and
  ssl3_record_st.hasName("ssl3_record_st") and
  data = ssl3_record_st.getAField() and
  data.hasName("data") and
  length = ssl3_record_st.getAField() and
  length.hasName("length") and
  pointsInto(fc.getArgument(1), data) and
  not comparedTo(fc.getArgument(2).(VariableAccess).getTarget(), length)
select fc, "This call to memcpy is insecure (Heartbleed vulnerability)."
```

Figure 8 CodeQL query for finding Heartbleed. (Included with CodeQL distribution.)

Other static analysis tools allow custom queries as well, but the model used by CodeQL allows for some particularly powerful and concise queries. This model is based on Datalog [4], a declarative logic language.

CodeQL is intended to be run over large code bases as part of continuous integration, providing rapid feedback to developers as they merge in code changes.

For vulnerability research, CodeQL shows potential in that a researcher can create a query that generalizes a particular vulnerability or code pattern of interest and search across large code bases. Or it can be used as an interactive part of the VR process to test hypotheses: "Is this particular variable ever used as part of an arithmetic operation?" "Does this function ever get called before this one?"

One drawback of this type of static analysis is that for some problems it can be challenging (or even impossible) to write a query that is both general enough to be useful and specific enough to avoid being overwhelmed by unintended findings. That said, a well-crafted query for a properly scoped problem can provide useful information with a manageable number of unintended findings.

2.1.4 Infer

Reference Link	**https://fbinfer.com/**
Target Type	Source code: C/C++/Objective C, Java
Host/Target Operating System	Linux, Windows, Mac, Android, iOS
Host/Target Architecture	x86 (32, 64), ARM, JVM
Initial Release	June 11, 2015 (open sourced)
License Type	MIT
Maintenance	Facebook, open source on GitHub [14]

Overview

Infer [15] is an open-source static program analyzer that uses an abstract interpretation technique called separation logic to reason about memory state and discover potential vulnerabilities in source code. Its analysis engine is also extensible to allow for other analyses.

Infer can be used to generate a list of potential bugs from a source base.

Design and Usage

Once Infer is installed, it is easy to use as long as the source code can be built. For example, if the program can be run with the "make" command, it can be as simple as:

```
% infer run -- make
```

Upon completion, Infer generates a report of potential issues. Here is example output from a project called Fuzzgoat [16], which includes intentional bugs for testing analysis tools such as fuzzers:

```
Found 2 issues

fuzzgoat.c:298: error: NULL_DEREFERENCE
  pointer `null_pointer` last assigned on line 297 could be null and is dereferenced at line 298, column 29.
  296.               if (value->u.string.length == 1) {
  297.                 char *null_pointer = NULL;
  298. >               printf ("%d", *null_pointer);
  299.               }
  300.   /****** END vulnerable code ************************************************/

fuzzgoat.c:1027: error: UNINITIALIZED_VALUE
  The value read from root was never initialized.
  1025.       }
  1026.
  1027. >     return root;
  1028.
  1029.   e_unknown_value:
```

Figure 9 Screenshot of actual output from Infer.

The first finding is one of the intentional bugs inserted by the authors of Fuzzgoat. The second finding is more of a code quality issue. The variable "root" is initialized inside a loop that is guaranteed to execute at least once, so contrary to the claim of the finding, it will be initialized when the above return instruction is reached. However, it can be argued that the finding is valid in the sense that "root" should be given a value upon declaration to support possible refactoring.

```
for (state.first_pass = 1; state.first_pass >= 0; -- state.first_pass)
{
   json_uchar uchar;
   unsigned char uc_b1, uc_b2, uc_b3, uc_b4;
   json_char * string = 0;
   unsigned int string_length = 0;

   top = root = 0;
```

Figure 10 Source code from Fuzzgoat.

Infer can also be configured to output reports in various formats as well as using a diff of various reports to show changes between different versions of a code base.

Use Cases and Limitations

Infer is intended to be run over large code bases as part of continuous integration, providing rapid feedback to developers as they merge in code changes. [17]

For vulnerability research, Infer shows potential to guide code auditors towards interesting code segments. This can be particularly useful if the findings can be integrated into a source browser so that they appear as annotations during browsing.

Infer generates a report of potential issues that can be used to repair defects or to help target further analysis. As with other static analysis tools this can include potentially many unintended or undesired findings. These are sometimes thought of as “false positives” in the sense that the finding may not be useful in terms of exploitability.

There are a number of open source and commercially available tools that provide similar services, though the underlying engine and type of analysis may differ. [18]

3 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 on page 49.

3.1 Dynamic Analysis Tools

3.1.1 Fermadyne

Reference Link	**https://github.com/firmadyne/firmadyne**
Target Type	• Binary • Embedded Linux Firmware
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	ARMEL (32 little endian); MIPSEL (32); MIPSEB (32)
Initial Release	Feb 2016
License Type	Open-Source (MIT)
Maintenance	Actively maintained on GitHub

Overview

FERMADYNE [19] is an end-to-end, automated solution for vulnerability research of embedded Linux firmware in bulk. It is composed of several modules that scrape, extract, emulate, and analyze embedded images. Each individual module is valuable as a tool in its own right, and they are applied in phases. The firmware images are collected from a variety of vendors and extracted. Each image is then emulated on QEMU [20] with a modified kernel to characterize network behavior. NVRAM (non-volatile random-access memory) peripherals are spoofed through APIs through use of LD_PRELOAD [21]. Finally, the running firmware images are probed from a TAP interface for a variety of SNMP and web server vulnerabilities.

Design and Implementation

FERMADYNE is composed of a variety of modules that automate embedded Linux firmware reverse engineering and exploitation. The modules form a complete toolchain that scrapes,

extracts, emulates, and analyzes embedded firmware in bulk. A high-level description of the project life cycle is in the figure below. Should any phase fail the project lifecycle cannot continue.

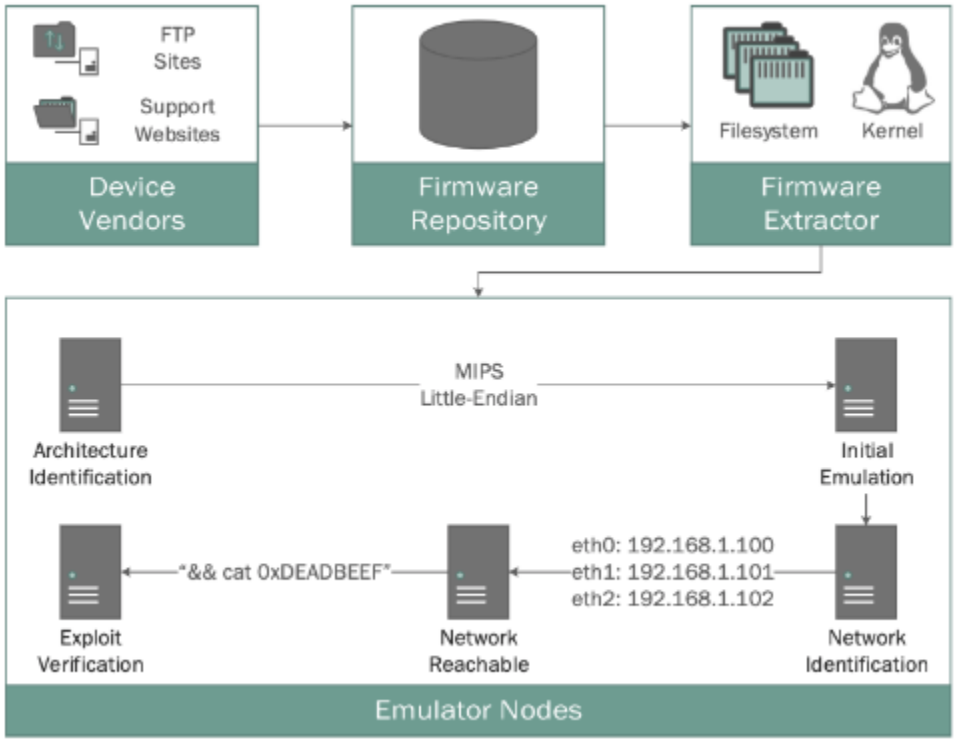


Figure 11 FIRMADYNE architectural diagram from "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware." [19]

Scrapers

The first phase is scraping: collecting the target firmware. The scrapers are written in python and leverage the scrapy website data extraction framework [22]. Firmware images are gathered by scrapy "spiders" that are manually tailored to a variety of vendor websites. These vendors include Qualcomm, Linksys, TP-link, Huawei, Netgear, and more.

Extractor

Once collected, the second phase extracts the collected firmware. The extractor is written in python to leverage binwalk [23] and peripherals like sasquatch [24] to recover the file system. The binaries in the file system are inspected to determine the target architecture. Crucially, the firmware image's kernel is not extracted and is instead substituted with a custom-built kernel. This custom kernel monitors networking system calls to perform a best guess network configuration during the emulation phase.

Emulator

Custom Kernel Instrumentation

The emulated environment is built on QEMU. As such, FERMADYNE can theoretically emulate any architecture supported by its parent project. However, for FERMADYNE to more accurately emulate the target environment it substitutes the original kernel with a custom build. This custom kernel contains instrumented networking system calls, special initialization of /dev and /proc, and custom lifecycle management of the init process. The readily available kernels with this instrumentation are for ARM 32 bit little endian and MIPS 32 bit big/little endian architectures. Other architectures are not immediately available and would require independent investment.

Supplying Dependencies: LibNVRAM

Many embedded Linux distributions use NVRAM for persistent storage. FERMADYNE uses a custom NVRAM library that is preloaded into the init process and implements a variety of routines. This guarantees all subsequent processes share the implemented NVRAM routines. Should the /sbin/init binary not exist, or libNVRAM not export necessary symbols, the project cannot continue.

Configuring Network Connectivity

FERMADYNE configures the image's network connectivity by executing the emulated firmware in a "learning" phase for 60 seconds. In this phase, "the emulator is configured with default hardware peripherals [and] up to four emulated network adapters." During the learning phases FERMADYNE tracks networking system calls through the custom kernel. Particularly, "the IP addresses assigned to network interfaces, as well as the presence of IEEE 802.1d bridges used to aggregate multiple network interfaces." Additional checks for VLANS and ethernet frames are also implemented. Should this network configuration fail, the project life cycle cannot continue. Once the network has been tailored and the /sbin/init process running, a TAP interface is created on the host and associated with the image's network adaptors to provide outside connectivity to the emulated environment.

Analyses

After the image is running and a network connection available, FIRMADYNE launches a variety of scripted exploits. The first analysis crawls web pages taken from /www or an equivalent directory, for pre-authentication bugs. Another analysis uses snmpwalk to find sensitive SNMP information. The final analysis uses known Metasploit exploits to find N-days.

Use Cases and Limitations

The FIRMADYNE project is intended for mass extraction, emulation, and analysis of embedded Linux firmware. This may require significant compute power, as each firmware image is potentially gigabytes in size and the scrapers are capable of collecting in excess of 23,000 images. Additionally, many images break design assumptions, which include: location of the init process, obscure networking, exotic storage, and unsupported architecture. Problems in any of these areas can halt the project before running any analyses. The current architectures supported are ARM 32 bit little endian, MIPSEB 32 bit, and MIPSEL 32 bit. Images with qualities described above require individual modifications that can be challenging to implement. Development must use musl libc and a musl toolchain. The overall efficacy of the project, as is, at each phase in its lifecycle with vendors separated by color, can be seen below.

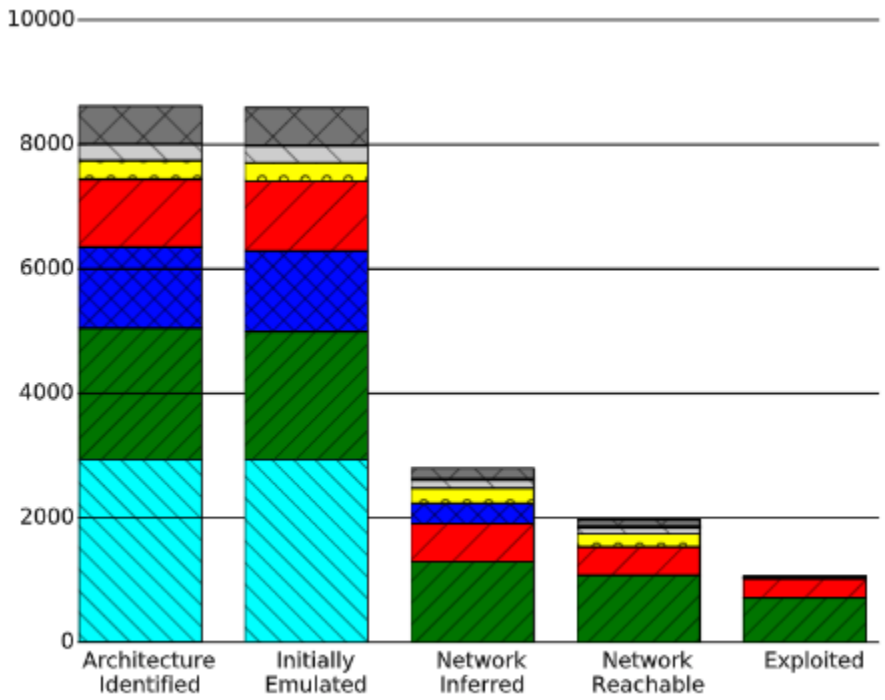


Figure 12 Breakdown of firmware images by emulation progress, colored by vendor. From "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware." [19]

Ultimately, the project is an effective method for testing the pervasiveness of an exploit against varieties of embedded Linux firmware. The components of FIRMADYNE are also useful. There are dozens of scrapers each tailored to a variety of websites that can be used independently. The binwalk extraction tool can also be used as a plug in to any extraction project. The analyses themselves can be used to walk webpages or launch exploits outside of the embedded firmware realm. Overall, the project can be used offensively or defensively to identify trends in the embedded Linux firmware research space.

3.1.2 Matryoshka

Reference Link	https://www.cs.ucdavis.edu/~hchen/paper/chen2019matryoshka.pdf
Target Type	Buildable Source Code
Host/Target Operating System	N/A
Host/Target Architecture	N/A
Initial Release	N/A
License Type	N/A
Maintenance	N/A

Overview

Matryoshka is a fuzzing tool for solving path constraints involving deeply nested, interdependent conditional statements such as the following:

```
1  void foo(unsigned x, unsigned y, unsigned z) {
2    if (x < 2) {
3      if (x + y < 3) {
4        if (z == 1111) {
5          if (y == 2222) { .... }
6          if (y > 1) { .... }
7        }
8      }
9    }
10 }
```

Figure 13 Nested Statement Example. From Matryoshka: Fuzzing Deeply Nested Branches. [25]

State of the art fuzzers do not offer nested branch solving capabilities due to the methods they use to solve constraints [25]. To reach the body on line 6 in the example statement above, many fuzzers will mutate only the variable *y* since it is in the conditional being examined. Mutating only *y* could render this statement unreachable since the updated value might then fail to satisfy the conditional at line 3. Statements such as this that are difficult to fuzz often occur in image and video decoders, network packet analyzers, and checksum tools. In test runs done with the Angora [26] fuzzer, a large percentage of the unsolved constraints were nested constraints, suggesting that improving nested constraint fuzzing could improve fuzzing performance and findings:

Program	Percentage of nested constraints in	
	all unsolved constraints	all constraints
djpeg	90.00 %	75.65 %
file	86.49 %	44.14 %
jhead	57.95 %	51.53 %
mutool	80.88 %	58.63 %
nm	84.32 %	68.16 %
objdump	90.54 %	73.95 %
readelf	84.12 %	70.50 %
readpng	94.02 %	89.50 %
size	87.86 %	71.46 %
tcpdump	96.15 %	78.98 %
tiff2ps	75.56 %	62.18 %
xmlint	78.18 %	72.37 %
xmlwf	96.18 %	68.16 %

Figure 14 Matryoshka Percentage Nested Constraints. From Matryoshka: Fuzzing Deeply Nested Branches. [25]

When analyzing a nested conditional statement, they found that it is "inadequate to mutate only the input bytes that flow into the conditional statement because doing so might render this statement unreachable". The authors came up with strategies to try to solve these constraints and created Matryoshka to implement these strategies. They ran Matryoshka against 13 open-source programs and found 41 unique new bugs and obtained 12 CVEs in 7 of the programs. When running against the LAVA-M [27] set of test binaries, only Matryoshka and the REDQUEEN [28] fuzzer were able to find all bugs added to the binaries.

Design

Matryoshka is implemented in Rust and C++. It borrows code from Angora for byte-level taint tracking and for mutating inputs using gradient descent.

The following high-level steps are taken by Matryoshka when it finds a new nested constraint during analysis:

- find unsolved nested constraint and related input
- identify control flow dependencies
- identify data (taint) flow dependencies
- solve for all constraints using optimization

Matryoshka tracks control flow and taint flow dependencies between conditional statements to produce only fuzzed data capable of reaching the conditional statement under test. The implementation of their flow analysis algorithms is split into intraprocedural control flow dependencies and interprocedural control flow dependencies. LLVM is used to analyze intraprocedural control flow dependencies.

To work around some limitations in LLVM, Matryoshka has a custom algorithm to track interprocedural control flow dependencies [25]. Data flow dependencies are tracked using two techniques: dynamic taint analysis to identify how each constraint is influenced by fuzzable inputs and union-find to group constraints with common corresponding inputs.

In their paper, the authors of Matryoshka attribute the fuzzer's impressive performance to its behavior of collecting nesting constraints of a target conditional statement that may cause the target statement to become unreachable. Their evaluation shows that taint flow only accounts for a small fraction of all conditional statements on the path to most target statements. Analyzing only this subset of conditional statements greatly simplifies the set of constraints that must be solved to reach the target statement.

Constraint solving is done using three strategies whose execution is split equally over time:

1. Prioritizing reachability (forward solving)
2. Prioritizing satisfiability (backward solving)
3. Join optimization

In the first strategy, variables already found in previous, dependent conditions are fixed and only new variables are fuzzed. In the second strategy, the target constraint is solved first and Matryoshka then analyzes the previous constraint and repeats this process of backwards solving, making sure that the line under test can still be reached after altering inputs. The third strategy looks at all individual constraints in the path, creates an objective function for these constraints, and then applies optimization methods to reduce the objective function to 0 in search of a solution. The third strategy is more intensive than the first two, but it can solve constraints that the first two strategies cannot.

The authors found that Matryoshka could naturally handle special structures or properties such as magic bytes or checksum values. The reachability and satisfiability strategies solved more constraints early on during an analysis while the joint optimization strategy grows slower but continues to grow when the other strategies have plateaued:

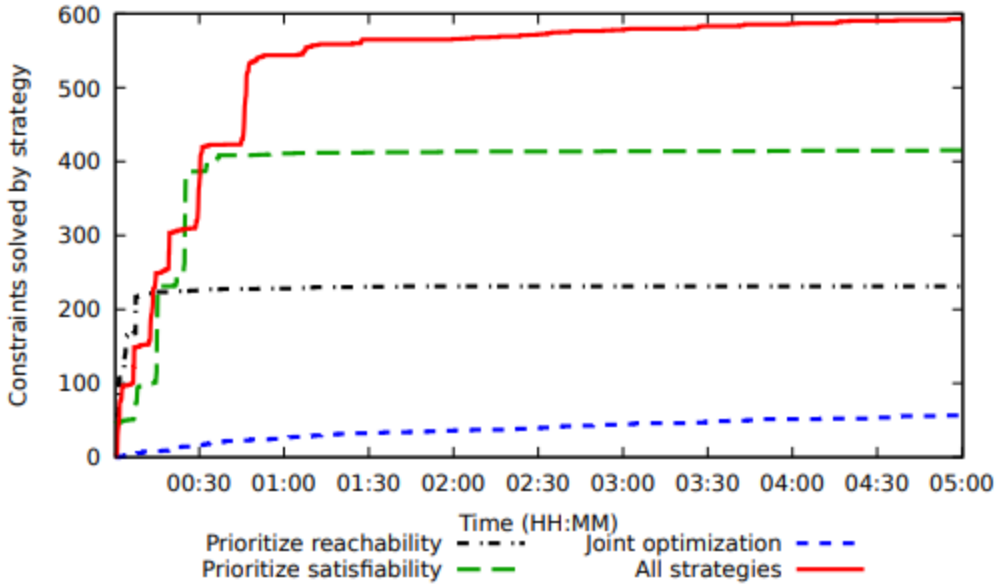


Figure 15 Constraints solved by strategy. From Matryoshka: Fuzzing Deeply Nested Branches. [25]

Use Cases and Limitations

Matryoshka's algorithm for fuzzing produced coverage improvements over other popular fuzzers when run against test binaries, as can be seen in the following coverage table:

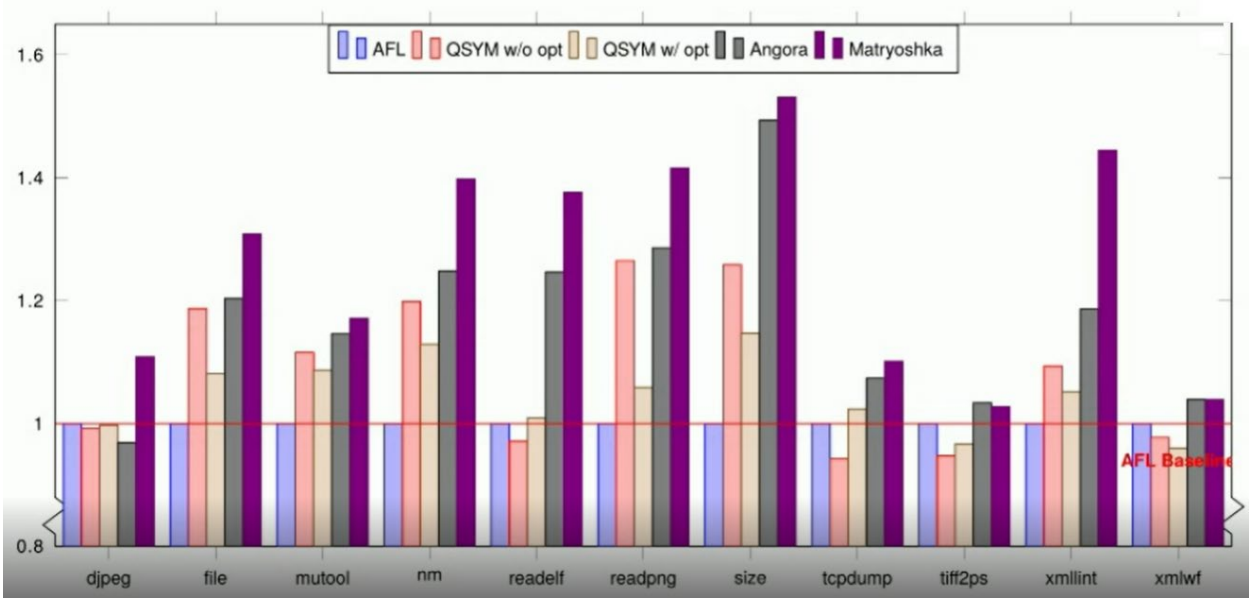


Figure 16 Graph based on data from Matryoshka: Fuzzing Deeply Nested Branches. [25] The y axis is a relative measure of program coverage.

On xmllint, line and branch coverage increased by 16.8% and 21.8%, respectively, over Angora . The authors argue that the coverage improvements are significant enough compared to the coverage of a concolic execution engine such as that used in the QSYM [29] fuzzer that we do not have to resort to resource intensive concolic execution to get good results. They go on to state that this avoids over-constraining issues that can occur when concolic execution engines are used on real-world programs.

Matryoshka only uses its strategies when analyzing a nested conditional statement. If the statement is not nested, then Matryoshka uses Angora or another fuzzer to solve the constraint. This gives Matryoshka the ability to match the speed of other fuzzers when solving non-nested conditional statements while doing better on nested statements.

Compared to symbolic execution, Matryoshka has the potential to solve hard constraints when the constraint is a monotonic function or when a local minimum is also a global minimum.

Matryoshka does have some limitations. It cannot track taint flows through external libraries. The developers manually modeled the taint flow in common external libraries, but their work was not comprehensive. Complex join constraints with many prior statements are still difficult to solve, as are constraints that are dependent on the *order* in which prior branches are taken.

Matryoshka's analysis is done using compile-time instrumentation, so the program under test must have available source code and a build environment.

In the video presentation of Matryoshka, the presenter states that they will release the code for Matryoshka once they finish reorganizing it. Unfortunately, we were not able to find any code, scripts, or programs online. We contacted one of the authors but did not hear back before publication of this report.

3.1.3 Qiling

Reference Link	**https://www.qiling.io/**
Target Type	● PE, MachO, ELF, COM, MBR ● Windows Driver (.sys), Linux Kernel Module (.ko) & MacOS Kernel (.kext) via Demigod
Host Operating System	Linux, MacOS, Windows, FreeBSD, DOS, UEFI and MBR
Target Operating System	Linux, MacOS, Windows, FreeBSD, DOS, UEFI and MBR
Host Architecture	X86, X86_64, Arm, Arm64, MIPS, 8086
Target Architecture	X86, X86_64, Arm, Arm64, MIPS, 8086
Initial Release	2019
License Type	Open-Source (GPLv2)
Maintenance	Actively maintained on GitHub

Overview

Qiling is a Python-based emulation framework that was originally designed to be a shellcode emulator [30]. The paradigmatic use case of Qiling is taking potentially malicious shellcode and examining its execution in a context that provides both insight into and control of the execution of the code, as well as an isolation layer that protects the host machine. To these ends, Qiling emulates both syscalls and operating system environments. These features allow Qiling to support cross-platform emulation. For example, Qiling is able to emulate Windows binaries on Linux hosts. CPU emulation allows Qiling to perform cross-architecture emulation as well (e.g., running 64-bit code on a 32-bit host). Qiling supports dynamic analysis via a hooking system, in which callbacks can be registered to types of events (e.g., memory writes, interrupts, instruction execution, etc.).

Design and Implementation

Qiling supports emulating both raw shellcode and binaries in a handful of file formats, including Portable Executable (Windows), Mach-O (macOS), and ELF (Linux). To support the latter, the Qiling framework includes a binary loader. The loader is responsible for configuring the emulation context for the targeted binary, as well as loading the binary into emulated memory.

To build this context, Qiling parses the required information from the target binary, using the binary's specification for its file format. While the loader handles mapping the target binary into emulated memory, the dynamic linker handles the resolution of dynamically loaded code. For example, Windows binaries frequently import functions from dynamically linked libraries (DLLs). In such cases, Qiling's dynamic linker will map the underlying DLL into emulated memory space and fix up the addresses of the imported functions to point to the now mapped DLL code.

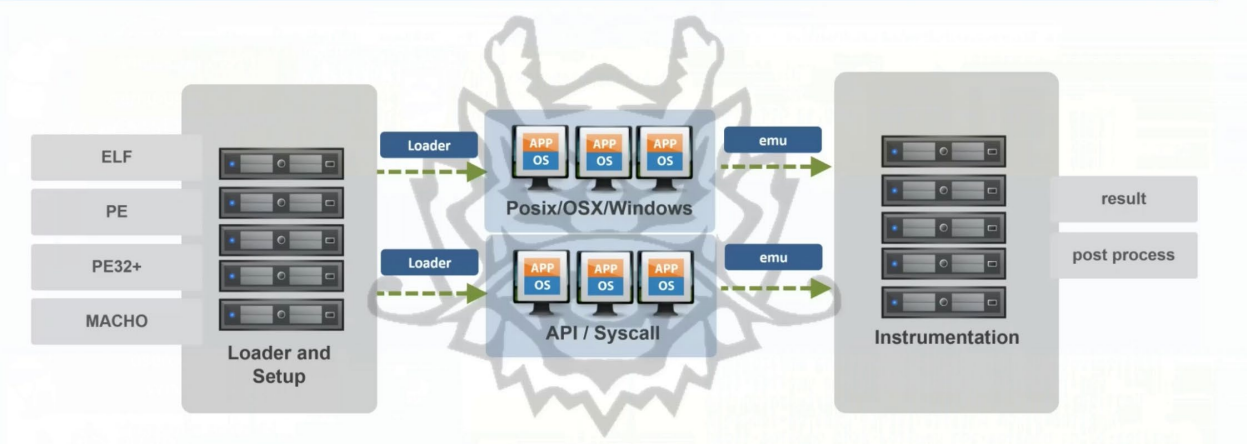


Figure 17 Overall design of the Qiling binary emulation framework. From "Qiling Framework: Learn how to build a fuzzer based on a 1day bug." [31]

Qiling relies on Unicorn [32] to perform CPU emulation. As Qiling single-steps through each instruction in the emulated binary, Unicorn evaluates the execution of each instruction and passes that information to Qiling. Qiling then updates its internal state tracking with that information. To better support additional architectures, Qiling has made their own modifications to the Unicorn engine.

Many binary emulation frameworks, such as user-mode QEMU [33], rely on syscall forwarding to handle the execution of syscalls. As the name implies, syscall forwarding consists of forwarding the execution of a syscall to the underlying host machine. This technique simplifies the emulation of syscalls but has the downside of restricting emulation to binaries whose execution is supported by the host machine (e.g., a Linux host can only emulate Linux binaries). Qiling takes the approach of syscall emulation. Instead of relying on the underlying host operating system to handle syscalls, Qiling emulates the execution of syscalls itself. By performing syscall emulation, Qiling can perform cross-platform emulation. Qiling's syscall emulation is split into two primary categories, OSX/Linux/*BSD and Windows. Because the similarities between the operating systems grouped together in the former category, much of the syscall emulation code is shared. In addition to syscall emulation, Qiling also performs kernel-mode emulation. This includes emulating kernel APIs (e.g., memory management), emulating driver interfaces (e.g., IOCTLs), and initializing critical kernel data structures. Qiling utilizes the Demigod framework [34] to perform kernel emulation. While originally a separate framework, Demigod has been merged into the Qiling framework as of Qiling version 1.2.

Qiling provides users the ability to perform dynamic program analysis via hook points exposed through an event system. When Qiling emulates the execution of an instruction, it checks whether the emulation of that instruction falls into a set of categories. Example categories include syscall execution, memory reads and writes, and general instruction execution. When such a categorization is made, Qiling checks whether any hooks have been registered to that category. If a hook exists, Qiling executes the callback function associated with each hook. For example, a Qiling-based program analysis tool that logs every interrupt triggered by a binary would consist of a Python script that registers a logging function to the interrupt event ("hook_intr" in Qiling). When an interrupt is emulated, Qiling will call the logging function. Qiling also supports dynamic memory patching, including patching the memory backing the target binary.

Qiling's emulation can be further controlled by connecting a debugger to the emulated binary. Connecting a debugger provides additional control over the emulation process itself, which Qiling does not provide natively. Examples include the ability to set breakpoints and single step through instructions. Qiling provides its own debugger, QDB [35]. However, Qiling also supports the gdb remote debugging protocol [36], which allows other debuggers (such as GDB and the IDA debugger) to debug emulated binaries as well.

Use Cases and Limitations

One potential use case for Qiling is as a foundation for building robust fuzzing tools. Qiling's event-based hooking capabilities allow for fine-grain detection of potentially interesting side effects. Qiling also provides restorable snapshots of an emulated state at any time, allowing

restoration of emulation to the point at which the snapshot was taken. In the context of fuzzing, the use of snapshots to reset program state to a certain point allows for rapid testing of new inputs by avoiding the overhead of initializing a program to the desired state. Qiling also provides the ability to run potentially malicious code in a sandboxed environment, protecting the host environment from exposure.

Regarding the limitations of Qiling, Qiling's support for emulating Windows APIs is incomplete. Many API calls are not implemented, meaning that there is a reasonable chance that Qiling will fail to fully emulate an arbitrary Windows program of non-trivial size. While workarounds have been suggested, including leveraging ReactOS or Wine to extend API support, these workarounds currently do not exist. Similarly, on the Unix side, many syscalls are emulated as null operations. For example, the emulation code for mprotect() simply returns zero. The decision to support some syscalls but not emulate their side effects was made to reduce development load. Qiling's dynamic linker allows it to resolve dependencies on external code. However, the dynamic linker must be provided the external shared libraries to map them into the emulated memory space and update references in the target binary to the newly mapped code. Thus, using Qiling to emulate code that has dependencies on dynamically loaded code requires the additional overhead of tracking down the required libraries and feeding them into Qiling, even if the emulated binary can be run natively on the host.

3.1.4 SymCC

Reference Link	**https://github.com/eurecom-s3/symcc**
Target Type	C/C++
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	X86, X86_64
Target Architecture	Architecture independent (via LLVM)
Initial Release	June 2020
License Type	Open-Source (GPLv3)
Maintenance	Actively maintained on GitHub

Overview

SymCC [37] is an LLVM-based C/C++ compiler framework. While SymCC can compile C and C++ programs to machine code via its integration with LLVM, its unique capability is the insertion of symbolic execution instrumentation into the programs it compiles. The result is a binary that can perform concolic execution (i.e., performing execution with both symbolic and concrete values) [38] while running at native speeds. By evaluating symbolic expressions natively, SymCC sees significant performance gains over other symbolic execution frameworks such as KLEE [39] and QSYM [29].

Design and Implementation

Like most compiler infrastructures, LLVM is composed of three primary components: a series of language-specific front-ends, middle-ends, and architecture-specific back-ends. At a high level, the front-ends are typically responsible for scanning and parsing source code, the middle-ends perform optimizations, and the back-ends produce machine code. While advertised as a compiler framework, SymCC is more an extension to the existing LLVM compiler infrastructure. Most of the work done by SymCC is performed by a custom LLVM IR pass (middle-end). As such, SymCC operates on LLVM's intermediate representation code (IR). Operating at the IR-level has its advantages. IR languages are typically constructed using fewer instructions than the machine code into which it is eventually translated. LLVM IR uses static single assignment, which makes symbolic execution much easier to implement. Because symbolic execution requires defining the semantics of all the instructions in a particular language, having a reduced instruction set lowers the amount of effort required to build a full set of semantics. Additionally, IR is source language agnostic. By leveraging LLVM IR, SymCC could theoretically be used to instrument any source language for which there exists an LLVM frontend (C/C++, Rust, Kotlin, Swift, etc.)

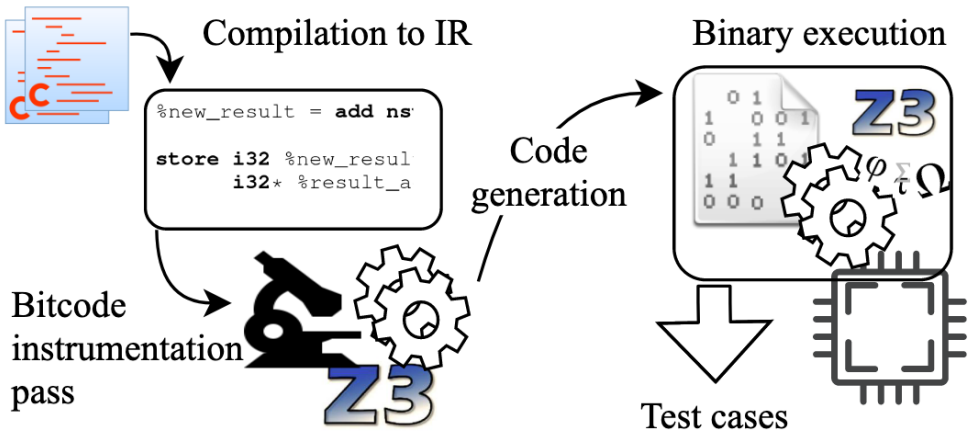


Figure 18 Overview of SymCC compiler process. From "Symbolic Execution with SymCC: Don't interpret, compile!" [37]

In the context of LLVM, SymCC operates as a module-level pass. As such, the SymCC pass iterates over the list of functions in the given module. For each function, SymCC constructs a set of symbolic instructions corresponding to the set of instructions in the original function. SymCC can reduce this set of symbolic instructions by removing any instructions that handle concrete values. Concrete values (structure offsets, hardcoded values, etc.) need not be included in the set of symbolic instructions, since those values never change between executions of the program to which they belong. Since SymCC utilizes both symbolic and concrete values, it falls within the category of concolic execution tools.

```
define i32 @is_double(i32, i32) {
  ; symbolic computation
  %3 = call i8* @_sym_get_parameter_expression(i8 0)
  %4 = call i8* @_sym_get_parameter_expression(i8 1)
  %5 = call i8* @_sym_build_integer(i64 1)
  %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)
  %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)
  %8 = call i8* @_sym_build_bool_to_bits(i8* %7)

  ; concrete computation (as before)
  %9 = shl nsw i32 %1, 1
  %10 = icmp eq i32 %9, %0
  %11 = zext i1 %10 to i32

  call void @_sym_set_return_expression(i8* %8)
  ret i32 %11
}
```

Figure 19 Sample instrumentation, which demonstrates the addition of symbolic calls and the preservation of the IR generated from the original source code. From "Symbolic Execution with SymCC: Don't interpret, compile!" [37]

It should be noted that the original IR code (derived from the program's original source code) remains unchanged. This means that the compiled program still executes the same set of instructions it would if it were compiled without SymCC. The only difference is that the SymCC compiled binary executes additional, symbolic instructions alongside the instructions generated from the original source code. Because the symbolic instructions are added directly to the IR code, they are subject to the same set of optimizations the original program receives, resulting in additional performance gains. Eventually, the instrumented and optimized IR code is translated into architecture-specific machine code by the compiler's back-end. The result is a binary that evaluates symbolic expressions natively, without the need for (and overhead from) an interpreter.

Most programming languages provide their own run-time libraries for interacting with the underlying operating system. To account for this code, SymCC can instrument these run-time libraries as well. This instrumentation allows for symbolic values (e.g., user input via functions like std::cin in C++) to be traced through the run-time library and into the target program.

The symbolic instructions emitted by SymCC take the form of function calls. These calls are implemented in an external library, which is linked into the runtime environment of the program. This external library is responsible for generating symbolic expressions and converting them into a format that can be ingested by an SMT solver. SymCC comes with its own backend library, which is a thin wrapper over Z3 [40]. However, because the execution of symbolic instructions exists independently of the backend, it is possible to build your own backend library without modifying the instrumentation part of SymCC. As part of the SymCC evaluation, its authors wrote such a backend that wraps QSYM.

Uses Cases and Limitations

One use case for SymCC is part of a fuzzing framework. In this context, SymCC can be used to expand fuzzing code coverage. Where a traditional fuzzer like AFL [41] would attempt to find new code execution paths by randomly generating inputs, SymCC uses its symbolic representation of branching conditions to solve for exact inputs that direct execution flow to new branches. In fact, this was one of the methodologies used to evaluate SymCC in its original paper. Another possible use case for SymCC is a foundation for language-agnostic program analysis. Since SymCC operates on LLVM bitcode, it can theoretically be applied to programs written in other languages for which there exists an LLVM front-end. This functionality allows the creation of program analysis code that can be reused across multiple programming languages compiled to different machine architectures.

Because it operates on LLVM-IR, the paradigmatic use case for SymCC requires access to source code. This is problematic in situations in which source code is not available. While frameworks that can "lift" machine code to LLVM IR exist, [42] [43] the use of these tools was not explored in the SymCC paper. Additionally, the LLVM IR generated by these frameworks is typically more verbose than the LLVM IR generated from the original source code, which may cause issues when generating the equivalent symbolic instructions. Since SymCC programs run natively (i.e., not in an interpreted environment), any code used by the compiled program that itself was not compiled with SymCC is effectively treated as a blackbox. Examples of such code include syscalls and code run in dynamically loaded libraries. While SymCC can still compile binaries that invoke external code, the values returned by external code are treated concretely, for which symbolic reasoning does not occur. It should also be noted that SMT solving, the methodology for generating values that satisfy a set of constraints generated from symbolic expressions, is NP-complete. This means that even if SymCC can generate a set of constraints understood by an SMT solver, it is not guaranteed the SMT solver can determine if it is possible to satisfy those constraints in a reasonable timeframe. SymCC hedges the impact of this issue by setting a 30-minute timeout on all test executions.

3.1.5 Zelos

Reference Link	**https://github.com/zeropointdynamics/zelos**
Target Type	Binary
Host Operating System	Linux, Windows
Target Operating System	Linux
Host Architecture	Any supporting Python
Target Architecture	X86, X86_64, ARM, MIPS
Initial Release	June 2020
License Type	Open-Source (GPLv3)
Maintenance	Actively maintained on GitHub

Overview

Zelos (Zeropoint Emulated Lightweight Operating System) is an open-source binary execution emulation framework [44]. Zelos allows users to control the emulation of the execution of a binary in much the same way that a debugger controls the execution of a binary in a native environment (e.g., setting breakpoints, single stepping, etc.). Additionally, Zelos exposes interfaces to the context in which the binary is emulated, specifically memory and register state. Zelos' two major contributions to achieving this functionality are syscall emulation (for a limited set of architectures), and a scriptable interface for performing the dynamic analysis of emulated binaries.

Design and Implementation

Like most binary emulators, Zelos emulates a program by single stepping through each instruction in a particular code path. The emulation of each instruction is performed by the CPU emulator. In the case of Zelos, CPU emulation is handled by Unicorn, which is a separate project [32]. Unicorn itself is a stripped-down version of QEMU [33]. As Zelos steps through each instruction, its internal state tracking is updated with the results of emulating the execution of that instruction.

When Zelos reaches a syscall instruction, it relies on custom syscall emulations to handle updating its internal state tracking accordingly. This process stands in contrast to syscall forwarding (i.e., forwarding syscalls to the host operating system to perform work on behalf of the emulation framework), which is used in other emulation frameworks such as QEMU. Syscall emulation is necessary to isolate the emulated program from the host operating system. As far as the program is concerned, syscalls are a black-box, which is why Zelos must perform its own emulation of them.

As Zelos runs a program, the emulation of each instruction is analyzed by an internal eventing engine. If the emulated instruction matches heuristics for any registered event types, Zelos will generate a new event of the corresponding type. Example event types include but are not limited to: single-step instruction execution, memory access (reads and writes), and syscall execution. These events are accessible to both scripts and plugins. Being accessible to the scripting and plugin interfaces allows for the creation of hooks, allowing code that only executes when certain events occur.

Zelos' tracing capabilities are built on top of the event engine and take the form of feeds. By reducing the number of subscriptions, feeds make it possible to control Zelos' runtime overhead, which can lead to performance gains.

Zelos exposes both its state tracking and emulation control functionality (i.e., the code that "drives" the emulation of a program) to its scripting interface. In the case of the former, scripts can directly modify the execution state generated as a program is emulated. This state includes both the virtual memory and register context. These basic primitives can then be used to build stronger primitives. For example, a user can leverage the ability to dynamically modify register values to redirect control-flow by writing to the instruction pointer register. Zelos' emulation

control functionality is exposed to the scripting interface as well. Examples of this functionality include setting breakpoints and executing single steps. Exposing this functionality allows users to write scripts that work with values that are not available at compile-time but are generated as the program is run.

```
def patch_mem():
    z = Zelos("password_check.bin", inst=True)
    # The address cmp instr observed above
    target_address = 0x0040107C
    # run to the address of cmp and break
    z.set_breakpoint(target_address, True)
    z.start()

    # Execution is now STOPPED at address 0x0040107C

    # Write 0x0 to address [rbp - 0x38]
    z.memory.write_int(z.regs.rbp - 0x38, 0x0)
    # resume execution
    z.start()

if __name__ == "__main__":
    patch_mem()
```

Figure 20 An example of a Zelos script (taken from the Zelos documentation [45]). This script demonstrates the use of API calls that leverage emulation control (setting a breakpoint) and state tracking modification (memory write).

The Zelos framework can be further extended via plugins. Plugins operate in a similar fashion to scripts but exist within the context of the Zelos framework. This locality allows for plugins to have access to internal Zelos functionality, which is not available via the scripting interface. Since they are incorporated into the Zelos framework itself, plugins can be run at the command line.

Zelos also supports dynamic binary patching. Whereas the scripting and plugin capabilities described above can be used to modify tracked state and emulation control, dynamic binary patching provides the ability to virtually patch the emulated binary. Under the hood, Zelos uses the Keystone assembler [46] to assemble a Python string of assembly instructions into an encoded string of opcodes. Zelos' ability to write to memory allows patching the emulated binary with the newly created string of bytes.

Upon completion of emulation, Zelos can export the information it gathered during emulation. This design allows for the creation of external plugins that convert Zelos data into other formats. For example, an IDA plugin [47] can ingest data exported from Zelos and use it to overlay information, such as what execution path was taken during emulation, onto the IDA GUI interface.

Use Cases and Limitations

One use case for Zelos is as an execution environment for fuzzing. Zelos emulates the binary rather than running it on bare metal. Because it controls the binary's execution context, Zelos provides a direct window into a substantial percentage of the side effects that occur during

emulation. This insight can be leveraged to monitor for interesting side effects such as out-of-bounds memory accesses, even those that don't result in a fault. Zelos' ability to export trace data can also be useful for supplying static analysis tools with data specific to a particular execution case.

For example, Zelos can be used to triage a crash case given a set of inputs. In this case, Zelos can emulate the execution of the program with the crashing input, then dump the resultant trace data. This dump can then be ingested into a static analysis tool (like IDA), providing additional data specific to that execution case (e.g., code flow, memory and register state, etc.).

Regarding limitations, Zelos depends on Unicorn to perform CPU emulation. While this dependency simplifies the management of the Zelos codebase, it also introduces some of Unicorn's shortcomings into Zelos. Most notably, the QEMU code on top of which Unicorn is built is very outdated [48]. As of the writing of this report, Unicorn was built on top of QEMU 2.2.1, while the current stable release of QEMU is version 5.1.0. Additionally, Zelos' current syscall emulation is limited to Linux syscalls on the following architectures: x86_64 (32 and 64-bit processes), ARM, and MIPS. There tend to be more dynamic vulnerability research (VR) tools for Linux than for Windows and macOS, with Zelos being another tool in that class. In a similar vein, the plugin ecosystem for Zelos is fairly small. Thus, adopting Zelos into an existing workflow would likely require additional development resources to build the required plugins for integration.

4 Techniques and Workflows

This is a new section for this edition of the EotA report. While previous reports have focused more on techniques implemented by tools, this section discusses approaches that analysts can take to improve their results. In other words, the focus is less on tools and more on how to use them. This can include methodologies, organization, collaboration, human-machine teaming, tool evaluation, and workflows.

The archetype of the lone genius hacker staring at code for days or weeks on end has a strong hold on the collective imagination. Ultimately, however, that model cannot scale to the level needed by society. This section focuses on a few new pieces of research that explore different approaches.

4.1 Fuzz Testing Evaluations

Overview

Motivation

Much academic research has gone into advancing the state of the art of fuzzers. However, the work of Klees et al [49] demonstrates that the current evaluation criteria of these advances is inadequate. Current evaluations rely largely on code coverage metrics and heuristics which do not accurately quantify ground truth bugs. Additionally, the performance of a fuzzer is largely susceptible to the context in which it runs - number of seeds, execution cycles, number of runs, etc. The research investigates the claims of modern fuzzing publications, "25 [of the] 32 papers [evaluated] published since 2016," identifies weaknesses in their evaluations, and suggests criteria for future evaluations. [49]

Findings

Current Evaluation Criteria is Insufficient

Evaluating fuzzers generally follows the procedure below:

1. A new fuzzing technique is developed and implemented
2. A collection of fuzzers are selected for comparison.
3. A compelling sample of target programs is selected for testing
4. A performance metric is measured that compares the fuzzers (e.g., crashing input, code coverage, bugs discovered)
5. A meaningful set of configurations are chosen - (e.g., number of runs, timeouts, seeds)
6. Metrics derived in (4) are compared

This procedure is logical and is used in this research as well. However, this research shows that the performance metrics and configurations vary widely in academic testing:

- "14 out of 32 papers … used code coverage to assess fuzzing effectiveness."

- “17 out of 32 fuzzing papers make no mention of the number of trials performed [and are presumed to be one].”
- “Only 6 papers injected artificial bugs [for ground truth]”
- “Only 2 papers used a non-empty seed corpus”

The variety of metrics result in conclusions that are either misleading, incomparable, irreproducible, or incorrect. This is further exacerbated by the random nature of fuzzers which mandates robust standards for testing. Fuzz tests against nm demonstrate that random variance can cause algorithmic performance to overlap. The figures below show the potential for AFL to outperform AFLFast under certain circumstances: execution time, and number of seeds provided. The “solid line is median; dashed lines are confidence intervals, and max/mins.”

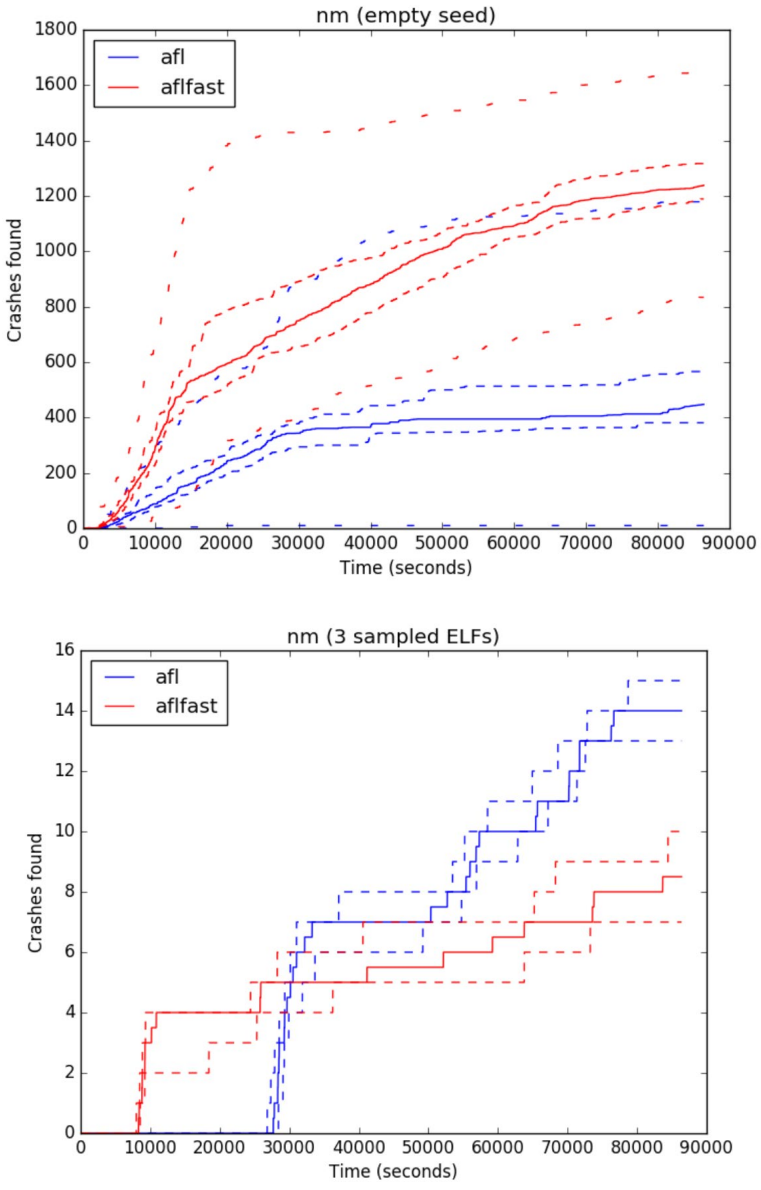


Figure 21 Crashes found over time. From “Evaluating Fuzz Testing.” [49]

The research also showed that once fuzzers found crashes, the methods by which a crash was mapped to a bug was unreliable. Automatic "crash deduplication relied on heuristics [that incorrectly counted] the number of unique crashes." This is seen below using the program cxxfilt, comparing AFL and AFLfast. "The height of the bar is the count of crashing inputs discovered during that run. Each bar is divided by color, clustering inputs with other inputs that share the same root cause, and the number of unique bugs is indicated above each bar."

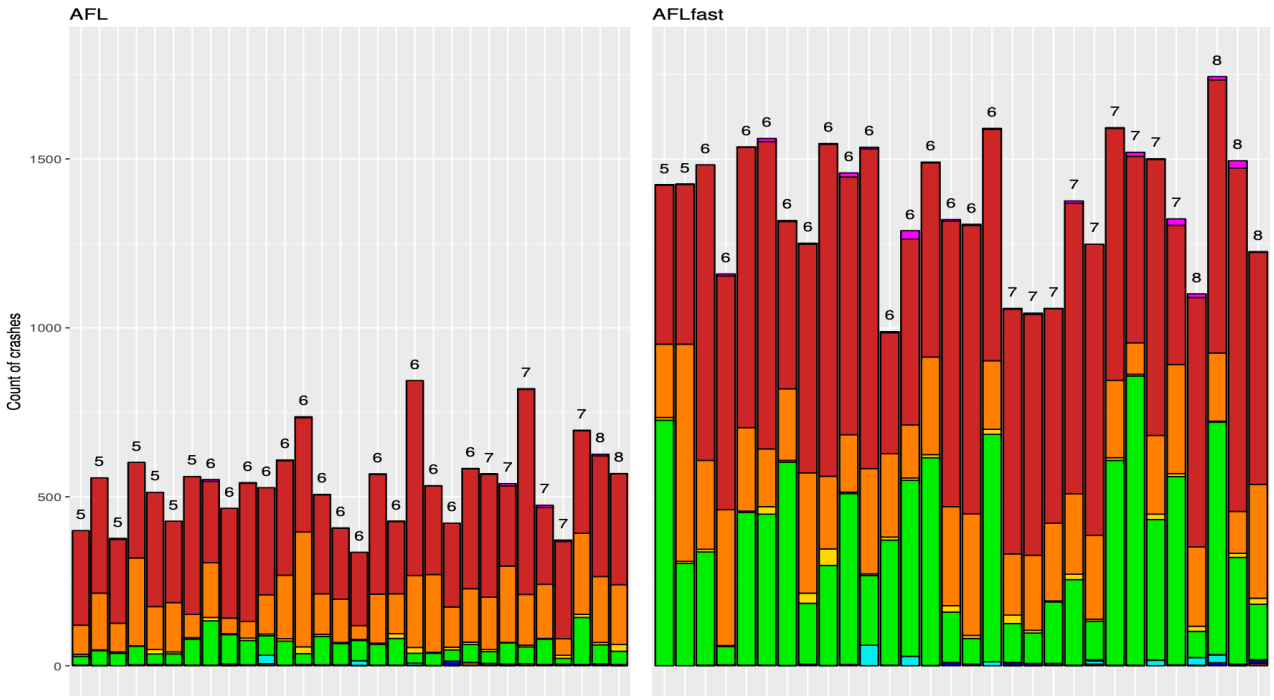


Figure 22 Importantly, though AFLfast produced thousands more crashes, the overall ground truth - number of bugs found - remains remarkably similar. From "Evaluating Fuzz Testing." [49]

It was found that overall variance resulted from:

1. "Reliance on heuristics for identifying unique crashes"
2. "Substantial performance variations based on the seeds used"
3. "Performance variations over the course of a [single] run"
4. "Performance variations under the same configuration [between runs]

The Experiment Setup

The research does not attempt to test the claims of all the fuzzing technology cited. It only serves to demonstrate the potential for drawing incorrect conclusions using the current, variable metrics found in other research. The fuzzers analyzed in the research appeared to only test AFL and AFLFast.

The Hardware

“The experiments were conducted on three machines. Machines I and II [were] equipped with twelve 2.9GHz Intel Xenon CPUs (each with 2 logical cores) and 48GB RAM running Ubuntu 16.04. Machine III has twenty-four 2.4GHz CPUs and 110GB RAM running Red Hat Enterprise Linux Server 7.4”

Software Corpus

The research “targeted three binutils programs: nm, objdump, and cxxfilt, and two image processing programs: gif2png and FFmpeg”. These were chosen as they were used in prior fuzzing evaluations. These targets do not represent “a complete benchmark suite” and it is acknowledged that “deriving a good benchmark suite is an open problem.”

Conclusion

The current criteria used to judge fuzzer performance is not sufficient to adequately describe their variance.

Suggested Evaluation Criteria

The research suggested the following best practices to standardize results for true comparisons:

1. “Multiple trials with statistical tests to distinguish distributions.”
2. “A range of benchmark target programs with known bugs (e.g., LAVA-M, CGC, or old programs with bug fixes).”
3. “A measurement of performance in terms of known bugs, rather than heuristics based on AFL coverage profiles or stack hashes - block or edge coverage can be used as a secondary measure.”
4. “A consideration of various (well documented) seed choices including the empty seed
5. “Timeouts of at least 24 hours, or else justification for less, with performance plotted over time”

Above all, the researchers think that “ground truth, according to bugs discovered, should always be the primary” metric by which fuzzers should be judged.

4.2 Vulnerability Research Workflows

Overview

Recent research into vulnerability discovery workflows attempts to both understand current practices as well as explore the possibilities of improving the vulnerability research (VR) process.

Much of the Edge of the Art report is dedicated to individual tools that provide some capability, often in the context of how an individual user might apply it to solve a problem. However, providing a single researcher more tools does not amount to a significant increase in productivity. Hiring additional expert researchers is expensive and ultimately cannot scale to match the growing demand for sophisticated analyses.

To match this demand, there are two key areas that must be addressed:

1. Allowing non-experts to contribute meaningfully to VR.
2. Effectively integrating machine autonomy into the VR process.

The key to unlocking these areas may be in finding better ways to organize people and autonomous systems into effective workflows.

In modern VR, autonomy includes many of the technologies outlined in the Edge of the Art report, such as state space exploration technologies like fuzzing and symbolic execution. DARPA's Cyber Grand Challenge [50] demonstrated the potential of fully autonomous "cyber reasoning systems." However, as recognized by DARPA's follow-on CHESS program and its goal of human-machine teaming, full automation falls far short of what is needed. Human insight is still critical for effective VR.

Experiments [51] have shown that even complete novices can improve upon the performance of cyber reasoning systems simply by interacting with (in this case) command-line interfaces. Through interaction, a human can understand the purpose of the program, what it should and shouldn't do, and even what common mistakes the developers may have made. Such analysis is completely orthogonal to that done by state space exploration programs, which are essentially exploring program branches through brute force with no concept of the intended semantics of the program. By transparently combining this simple human intuition with automation, researchers significantly improved results.

Other research [52] [53] has attempted to identify and formalize the process used by both VR practitioners ("hackers") and software testers, identifying a common set of steps used by both to effectively carry out their related work:

1. Information Gathering
2. Program Understanding
3. Attack Surface

4. Exploration
5. Vulnerability Recognition
6. Reporting

Another study [54] augments this process with an additional "step 0" called Targeting. Through a human study, the work showed that a diverse VR team could more effectively leverage both the experience of its members as well as maximize the use of automation by carrying out multiple analyses in a "breadth-first" targeting strategy, rather than devoting the team to one or a few targets. In this way, teams allow less experienced analysts to make a first attempt at a target by finding the quickest path to automation, moving on to another target, and only revisiting the original once that automation had a chance to complete an initial exploration.

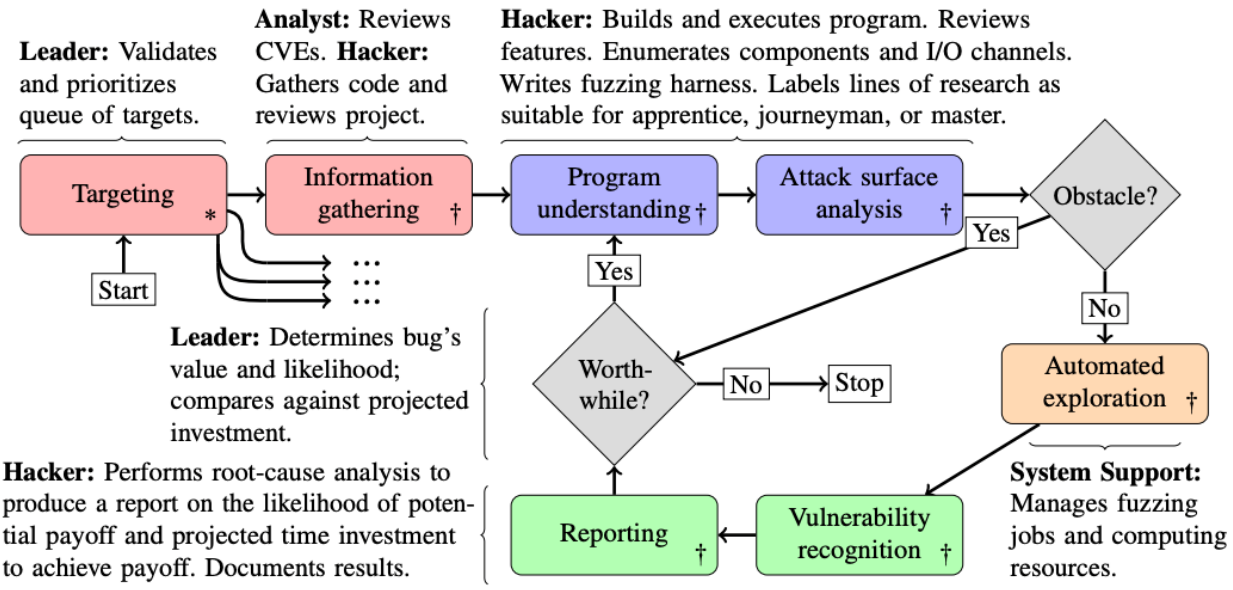


Figure 23: A vulnerability process as described in "The Industrial Age of Hacking" [54]

More research is needed to identify and improve upon current workflows, especially given the diverse set of targets and requirements in VR, some combinations of which may need specific solutions.

5 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.

5.1 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:

1. **Social Media** - Participating in social media platforms, including online forums and chat applications, to identify key influencers, build relationships, and identify new research directions.
2. **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.
3. **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.
4. **Academic Literature** - Surveying academic literature frequently to ensure complete coverage of novel algorithms and approaches driven by academic research.

5.2 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.

5.3 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.

5.4 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.

5.5 Static Analysis Technical Overview

5.5.1 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) [55, p. 2]. Many popular disassemblers including IDA Pro [56] and Binary Ninja [57] 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 [58], 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 [59] which has features complementary to its core disassembler and is designed to be used via an API. The disassembly framework Miasm [60], 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 [56], Ghidra [61] and Binary Ninja [57].

5.5.1.1 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 Dissassembling* [62], 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 [62, 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 [62, p. 1]. Since 2015, this technique has been improved, notably by the creators of angr who built a reassembling tool called Ramblr [63].

More recently, the tool DDisasm [64] was introduced. (DDisasm was discussed in the first version of this report.)

5.5.1.2 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 [55], 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 [65], 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.

5.5.1.3 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 [66], the IR used in the Clang compiler [67]. 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 [68]. Unlike machine code however, LLVM is designed to be human readable [68].

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 [70], 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 [71].

Ghidra uses SSA form in its decompilation. However, unlike LLVM, P-Code is not in SSA form by default [71]. 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 [57].

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.

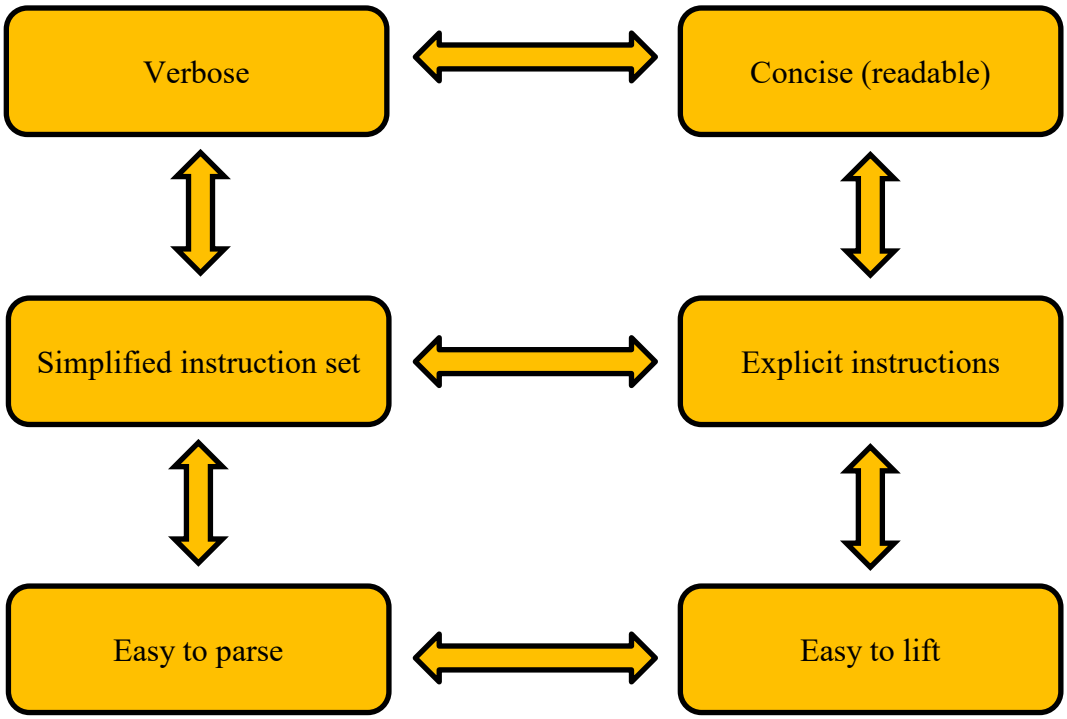


Figure 24: Tradeoffs of IRs, Pt. 1 [72, p. 29]
The double arrows imply that emphasizing one makes the other more difficult.

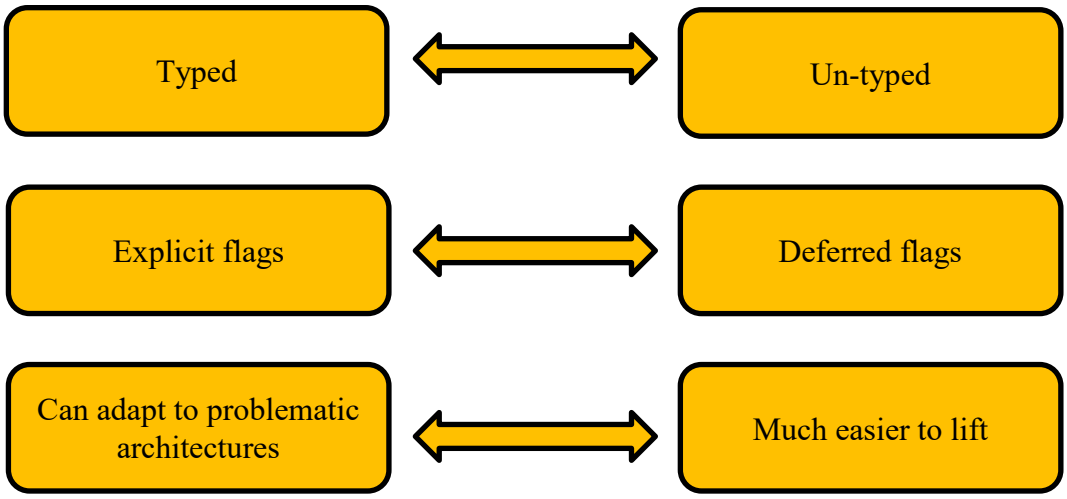


Figure 25: Tradeoffs of IRs, Pt. 2 [72, p. 30]
The double arrows imply that emphasizing one makes the other more difficult.

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 utilize it for their own analyses. IDA Pro only recently documented their API [73], 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 [74]. 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) [75] is a framework designed for program analysis which is built around the BAP Intermediate Language (BIL), which has a formally defined grammar [76]. Miasm has an expression-based IR that facilitates tracking memory and registry values [77]. Miasm also has a JIT engine for emulation and has built in support for symbolic execution [78].

Certain IRs are tailored for specific use cases. For example, Fuzzilli, a fuzzer which targets Javascript JIT engines, uses a custom IR called FuzzIL [79]. Seeds are constructed and mutated in FuzzIL then translated into Javascript before being fed into the engine [79]. 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 [80]. WinAFL, a version of the AFL fuzzer for the Windows operating system, uses DynamoRIO, a dynamic binary instrumentation engine with its own IR [81].

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 [82], 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.

5.5.2 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* [83]. 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 [84]. 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 [85]. 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.

5.5.3 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 [86], CodeSonar [87], and CodeQL [88]. 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) [80, p. 5].

5.5.3.1 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) [80, 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 [55, p. 5]. 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 [55, p. 5].

- **Value Set Analysis (VSA):** VSA is a form of static analysis which attempts to track values and references throughout a binary [80, p. 5]. 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 [80, p. 5].

- **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 50 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. [89].

5.6 Dynamic Analysis Technical Overview

5.6.1 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 [90]) 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 [91]. The tool rr [92], which was discussed in the first version of this report, enables recordable replayable debugging on Linux.

5.6.2 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 [93] and DynamoRIO [94], which underlie many of the tools discussed in these reports. Both can be used to drive the Windows fuzzer WinAFL [95], and the dynamic binary analysis tool Triton is built around Intel Pin [96]. 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 [93]. 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 [94]. 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 [97] operates).

5.6.3 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.

5.6.4 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 [98], Dr. Memory (a part of the DynamoRIO framework) and LLVM's Sanitizer Suite which includes Address Sanitizer (ASAN) [66].

5.6.5 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.

5.6.6 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 [99, p. 1]". 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 [99, p. 3]. In the paper *(State of) The Art of War: Offensive Techniques in Binary Analysis* [80], 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. [80, 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 [100].

The other common definition of DSE and concolic execution refers solely to symbolic execution which is "driven by a specific concrete execution [101, 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 [102], [101, pp. 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 [101, 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 [103] (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 [100] (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 [101]. 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.

5.6.6.1 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.

6 Bibliography

[1] Nalen98, "AngryGhidra Github Project," [Online]. Available: https://github.com/Nalen98/AngryGhidra. [Accessed 5 January 2021].

[2] GitHub, "CodeQL for research," [Online]. Available: https://securitylab.github.com/tools/codeql. [Accessed 5 January 2021].

[3] Semmle, "Semmle," [Online]. Available: https://semmle.com/. [Accessed 5 January 2021].

[4] Wikipedia, "Semmle," [Online]. Available: https://en.wikipedia.org/wiki/Semmle. [Accessed 5 January 2021].

[5] N. Friedman, "Welcoming Semmle to Github," GitHub, 18 September 2019. [Online]. Available: https://github.blog/2019-09-18-github-welcomes-semmle/. [Accessed 5 January 2021].

[6] Semmle, "lgtm," [Online]. Available: https://lgtm.com/. [Accessed 5 January 2021].

[7] GitHub, "CodeQL CLI," [Online]. Available: https://help.semmle.com/codeql/codeql-cli.html. [Accessed 5 January 2021].

[8] GitHub, "Basic query for C and C++ code," [Online]. Available: https://help.semmle.com/QL/learn-ql/cpp/basic-query-cpp.html. [Accessed 5 January 2021].

[9] Microsoft, "CodeQL for Visual Studio Code," 14 November 2019. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-codeql. [Accessed 5 January 2021].

[10] GitHub, "vscode-codeql-starter," [Online]. Available: https://github.com/github/vscode-codeql-starter. [Accessed 5 January 2021].

[11] GitHub, "CodeQL Critical," [Online]. Available: https://github.com/github/codeql/tree/main/cpp/ql/src/Critical. [Accessed 5 January 2021].

[12] GitHub, "CodeQL query help," [Online]. Available: https://help.semmle.com/wiki/pages/viewpage.action?pageId=1608834. [Accessed 5 January 2021].

[13] GitHub, "Query module OpenSslHeartbleed," [Online]. Available: https://help.semmle.com/qldoc/cpp/Security/CWE/CWE-327/OpenSslHeartbleed.ql/module.OpenSslHeartbleed.html. [Accessed 5 January 2021].

[14] Facebook, "Infer Github Project," [Online]. Available: https://github.com/facebook/infer. [Accessed 5 January 2021].

[15] Facebook, "A tool to detect bugs in Java and C/C++/Objective-C code before it ships," [Online]. Available: https://fbinfer.com/. [Accessed 5 January 2021].

[16] fuzzstati0n, "Fuzzgoat Github Page," [Online]. Available: https://github.com/fuzzstati0n/fuzzgoat. [Accessed 5 January 2021].

[17] P. O'Hearn, D. Distefano and C. Calcagno, "Open-sourcing Facebook Infer: Identify bugs before you ship," 11 June 2015. [Online]. Available: https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/. [Accessed 5 January 2021].

[18] Wikipedia, "List of tools for static code analysis," [Online]. Available: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis. [Accessed 10 January 2021].

[19] D. D. Chen, M. Woo, D. Brumley and M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," *NDSS,* vol. 16, pp. 1-16, 2016.

[20] QEMU Organization, "QEMU: the FAST! processor emulator," [Online]. Available: https://www.qemu.org/. [Accessed 5 January 2021].

[21] P. Goldsborough, "The LD_PRELOAD trick," [Online]. Available: http://www.goldsborough.me/c/low-level/kernel/2016/08/29/16-48-53-the_-ld_preload-_trick/. [Accessed 5 January 2021].

[22] "Scrapy," [Online]. Available: https://scrapy.org/. [Accessed 18 January 2021].

[23] "Binwalk Github Page," [Online]. Available: https://github.com/ReFirmLabs/binwalk. [Accessed 18 January 2021].

[24] "Sasquatch Github Page," [Online]. Available: https://github.com/devttys0/sasquatch. [Accessed 18 January 2021].

[25] P. Chen, J. Liu and H. Chen, "Matryoshka: Fuzzing Deeply Nested Branches," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security,* pp. 499-513, 2019.

[26] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," *IEEE Symposium on Security and Privacy,* pp. 711-725, 2018.

[27] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich and R. Whelan, "LAVA: Large-scale Automated Vulnerability Addition," *IEEE Symposium on Security and Privacy,* pp. 110-121, 2016.

[28] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," *NDSS,* vol. 19, pp. 1-15, 2019.

[29] I. Yun, S. Lee, M. Xu, J. Yeongjin and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," *27th USENIX Security Symposium,* pp. 745-761, 2018.

[30] L. K. Jern, "Qiling Advanced Binary Emulation Framework," [Online]. Available: https://www.qiling.io. [Accessed 6 January 2021].

[31] K. Lau, A. Q. Nguyen, H. Chen, T. Ding, B. Sun and T. Yu, "Qiling Framework: Learn how to build a fuzzer based on a 1day bug," in *Hack-in-the-Box*, 2020.

[32] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA,* 2015.

[33] F. Bellard, "QEMU, a fast and portable dynamic translator," *USENIX Annual Technical Conference, FREENIX Track,* vol. 41, p. 46, 2005.

[34] N. A. Quynh, N. H. Quang and D. M. Tuan, "Demigod: The Art of Emulating Kernel Rootkits," *BlackHat USA,* 2020.

[35] L. K. Jern, "Qdb: Qiling Debugger," [Online]. Available: https://github.com/ucgJhe/Qdb. [Accessed 6 January 2021].

[36] B. Gatliff, "Embedding with gnu: the gdb remote serial protocol," *Embedded Systems Programming,* pp. 108-113, 1999.

[37] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!," *29th USENIX Security Symposium,* pp. 181-198, 2020.

[38] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes,* vol. 30, no. 5, pp. 263-272, 2005.

[39] C. Cadar, D. Dunbar and D. R. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI,* vol. 8, pp. 209-224, 2008.

[40] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," *International conference on Tools and Algorithms for the Construction and Analysis of Systems,* pp. 337-340, 2008.

[41] M. Zalewski, "american fuzzy lop," Google, [Online]. Available: https://github.com/google/AFL. [Accessed 6 January 2021].

[42] Microsoft, "Machine Code to LLVM," [Online]. Available: https://github.com/microsoft/llvm-mctoll. [Accessed 6 January 2021].

[43] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm," in *ReCon*, Montreal, Canada, 2014.

[44] zeropointdynamics, "Zeropoint Emulated Lightweight Operating System," [Online]. Available: https://github.com/zeropointdynamics/zelos. [Accessed 6 January 2021].

[45] Zeropoint Dynamics, "Scripting with Zelos," [Online]. Available: https://zelos.readthedocs.io/en/latest/tutorials/02_scripting.html?highlight=patch_mem#method-1-writing-memory. [Accessed 13 January 2021].

[46] A. Q. Nguyen, "KEYSTONE: Next Generation Assembler Framework," in *BlackHat USA*, Las Vegas, NV, 2016.

[47] Zeropoint Dynamics, "Zelos: Exporting An Overlay & IDA Pro Plugin," [Online]. Available: https://zelos.readthedocs.io/en/latest/tutorials/06_snapshot_overlay.html. [Accessed 6 January 2021].

[48] "unicorn qemu version," [Online]. Available: https://github.com/unicorn-engine/unicorn/blob/master/qemu/VERSION. [Accessed 6 January 2021].

[49] G. Klees, A. Ruef, B. Cooper, S. Wei and M. Hicks, "Evaluating Fuzz Testing," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security,* pp. 2123-2138, 2018.

[50] DARPA, "Cyber Grand Challenge (CGC)," [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge. [Accessed 6 January 2021].

[51] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel and G. Vigna, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," *ACM SIGSAC Conference on Computer and Communications Security,* pp. 347-362, 2017.

[52] D. Votipka, R. Stevens, E. Redmiles, J. Hu and M. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," *IEEE Symposium on Security and Privacy (SP),* pp. 374-391, 2018.

[53] D. Votipka, S. Rabin, K. Micinski, J. S. Foster and M. L. Mazurek, "An Observational Investigation of Reverse Engineers' Processes," *29th USENIX Security Symposium,* pp. 1875-1892, 2020.

[54] T. Nosco, J. Ziegler, Z. Clark, D. Marrero, T. Finkler, A. Barbarello and W. M. Petullo, "The Industrial Age of Hacking," *29th USENIX Security Symposium,* pp. 1129-1146, 2020.

[55] S. Dinesh, N. Burow, D. Xu and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, 2020.

[56] Hex-Rays, "Hex-Rays," 30 09 2019. [Online]. Available: Hex-Rays.com. [Accessed 21 1 2020].

[57] Vector 35, "Binary Ninja," 2019. [Online]. Available: binary.ninja. [Accessed 21 1 2020].

[58] Free Software Foundation, Inc., "GNU Binutils," 12 10 2019. [Online]. Available: https://www.gnu.org/software/binutils/. [Accessed 21 1 2020].

[59] N. A. Quynh, "Capstone Engine," [Online]. Available: http://www.capstone-engine.org/. [Accessed 21 1 2020].

[60] CEA IT Security, "Miasm," [Online]. Available: miasm.re. [Accessed 21 1 2020].

[61] National Security Agency, "Ghidra," [Online]. Available: ghidra-sre.org. [Accessed 21 1 2020].

[62] S. Wang, P. Wang and D. Wu, "Reassembleable Disassembly," in *USENIX Security Symposium*, Washington, D.C, 2015.

[63] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2017.

[64] A. Flores-Montoya and E. Schulte, "Datalog Disassembly," *eprint arXiv:1906.03969,* 2019.

[65] Quarkslab, "LIEF," [Online]. Available: https://lief.quarkslab.com/. [Accessed 21 1 2020].

[66] LLVM Foundation, "The LLVM Compiler Infrastructure," [Online]. Available: https://llvm.org/. [Accessed 21 01 2020].

[67] LLVM Foundation, "Clang: a C language family frontend for LLVM," [Online]. Available: https://clang.llvm.org/. [Accessed 21 1 2020].

[68] L. Foundation, "LLVM Language Reference Manual," [Online]. Available: https://llvm.org/docs/LangRef.html. [Accessed 21 1 2020].

[69] LLVM Foundation, "“CLANG” CFE INTERNALS MANUAL," [Online]. Available: https://clang.llvm.org/docs/InternalsManual.html. [Accessed 21 1 2020].

[70] National Security Agency, "Ghidra Software Reverse Engineering Framework," [Online]. Available: https://github.com/NationalSecurityAgency/ghidra. [Accessed 21 1 2020].

[71] A. Bulazel, "WORKING WITH GHIDRA'S P-CODE TO IDENTIFY VULNERABLE FUNCTION CALLS," Riverloop Security, 11 05 2019. [Online]. Available: https://www.riverloopsecurity.com/blog/2019/05/pcode/. [Accessed 21 1 2020].

[72] "P. LaFosse and J. Weins, “Modern Binary Analysis with ILs,” presented at the BlueHat Seattle 2019, BlueHat Seattle 2019, 25-Oct-2019.".

[73] R. Rolles, "Hex-Rays Microcode API vs. Obfuscating Compiler," 19 09 2018. [Online]. Available: https://www.hexblog.com/?p=1248. [Accessed 21 1 2020].

[74] Vector 35, "Binary Ninja Intermediate Language Series, Part 1: Low Level IL," [Online]. Available: https://docs.binary.ninja/dev/bnil-llil.html. [Accessed 21 1 2020].

[75] "CMU Cylab, “Carnegie Mellon University Binary Analysis Platform (CMU BAP).” [Online]. Available: https://github.com/BinaryAnalysisPlatform/bap.," [Online]. [Accessed 21 1 2020].

[76] "CMU Cylab, “A formal specification for BIL: BIL Instruction Language,” 02-Oct-2015. [Online]. Available:

https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf.," [Online]. [Accessed 10 1 2020].

[77] *Fabrice Desclaux, "Miasm : Framework de reverse engineering," Symposium sur la sécurité des technologies de l'information et des communications (SSTIC) 2012, p. 25, 2012..*

[78] *CEA IT Security, "Miasm." [Online]. Available: https://github.com/cea-sec/miasm. [Accessed: 10-Jan-2020]..*

[79] S. Groß, "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines," Karlsruhe Institute of Technology (KIT), 2018..

[80] *Y. Shoshitaishvili et al., "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, doi: 10.1109/SP.2016.17..*

[81] *Google Project Zero, "WinAFL Dynamorio Instrumentation mode," 21-Jun-2019. [Online]. Available: https://github.com/googleprojectzero/winafl. [Accessed: 21-Jun-2019]..*

[82] *GrammaTech, "GTIRB." [Online]. Available: https://github.com/GrammaTech/gtirb/blob/master/README.md. [Accessed: 10-Jan-2020]..*

[83] *C. Cifuentes, "Reverse Compilation Techniques," Queensland University of Technology, 1994..*

[84] *I. Guilfanov, "Keynote: The story of IDA Pro," presented at CODE BLUE 2014, 2014..*

[85] *L. H. Newman, "The NSA Makes Ghidra, a Powerful Cybersecurity Tool, Open Source," Wired, 05-2019. [Online]. Available: https://www.wired.com/story/nsa-ghidra-open-source-tool/. [Accessed 2019 01 21]..*

[86] Synopsys, "COVERITY SCAN STATIC ANALYSIS," [Online]. Available: https://scan.coverity.com/. [Accessed 21 1 2020].

[87] GrammaTech, "CodeSonar," [Online]. Available: https://www.grammatech.com/products/codesonar. [Accessed 21 1 2020].

[88] Semmle, "Semmle - Code Analysis Platform for Securing Software," [Online]. Available: https://semmle.com/. [Accessed 21 1 2020].

[89] *Gotovchits, Ivan, "[ANN} BAP 2.0 Release," OCaml, Nov-2019. [Online]. Available: https://discuss.ocaml.org/t/ann-bap-2-0-release/4719..*

[90] Free Software Foundation, Inc., "GDB: The GNU Project Debugger," [Online]. Available: https://www.gnu.org/software/gdb/. [Accessed 21 1 2020].

[91] Microsoft, "Debugging Using Windbg Preview," [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview. [Accessed 2020 13 05].

[92] Mozilla, "rr Project," [Online]. Available: https://rr-project.org/. [Accessed 13 5 2020].

[93] Intel, "Pin - A Dynamic Binary Instrumentation Tool," Intel , [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. [Accessed 21 1 2020].

[94] D. Bruening, "DynamoRIO," [Online]. Available: https://www.dynamorio.org/. [Accessed 21 1 2020].

[95] Google Project Zero, "WinAFL," 21-Jun-2019. [Online]. Available: https://github.com/googleprojectzero/winafl. [Accessed: 21-Jun-2019].

[96] Quarkslab, "Triton - A DBA Framework," [Online]. Available: triton.quarkslab.com. [Accessed 21 1 2020].

[97] O. A. V. Ravnås, "Frida," [Online]. Available: frida.re. [Accessed 21 01 2020].

[98] T. V. Developers, "Valgrind," [Online]. Available: https://valgrind.org/. [Accessed 21 1 2020].

[99] V. Sharma, M. Whalen, S. McCamant and V. Willem, "Veritesting Challenges in Symbolic Execution of Java," *ACM SIGSOFT Software Engineering Notes,* vol. 42, pp. 1-5, 2018.

[100] Seclab at University of California, Santa Barbara and SEFCOM at Arizona State University , "angr," [Online]. Available: http://angr.io/. [Accessed 21 1 2020].

[101] 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, 2018.

[102] *CEA IT Security, "Playing with Dynamic symbolic execution," Miasm, 05-Oct-2017. [Online]. Available: https://miasm.re/blog/2017/10/05/playing_with_dynamic_symbolic_execution.html. [Accessed: 10-Jan-2020]..*

[103] sslab-gatech, QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. gts3.org (SSLab@Gatech), 2019. .

[104] GitHub, "Running CodeQL code scanning in your CI system," [Online]. Available: https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code/running-codeql-code-scanning-in-your-ci-system. [Accessed 5 January 2021].

[105] "LLVM Language Reference Manual," [Online]. Available: https://llvm.org/docs/LangRef.html. [Accessed 6 January 2021].

[106] Qiling Framework, "Qiling Advanced Binary Emulation framework," [Online]. Available: https://github.com/qilingframework/qiling. [Accessed 21 01 2020].

[107] H. Xu, Z. Zhao, Y. Zhou and M. R. Lyu, "On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs," *On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs ,* December 2017.