



AFRL-RI-RS-TM-2020-001

## **EDGE OF THE ART IN VULNERABILITY RESEARCH**

---

TWO SIX LABS

*APRIL 2020*

TECHNICAL MEMORANDUM

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED***

STINFO COPY

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE**

## **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 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

**/ S /**

AMANDA OZANAM  
Work Unit Manager

**/ S /**

JAMES 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.

<b>REPORT DOCUMENTATION PAGE</b>				<b>Form Approved OMB No. 0704-0188</b>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. <b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b>					
<b>1. REPORT DATE (DD-MM-YYYY)</b> APRIL 2020		<b>2. REPORT TYPE</b> CONFERENCE PAPER (Post Print)		<b>3. DATES COVERED (From - To)</b> JAN 2019 – SEP 2019	
<b>4. TITLE AND SUBTITLE</b>  EDGE OF THE ART IN VULNERABILITY RESEARCH				<b>5a. CONTRACT NUMBER</b> FA8750-19-C-0009	
				<b>5b. GRANT NUMBER</b> N/A	
				<b>5c. PROGRAM ELEMENT NUMBER</b> 62303E	
<b>6. AUTHOR(S)</b>  Scott Tenaglia Perri Adams				<b>5d. PROJECT NUMBER</b> CHES	
				<b>5e. TASK NUMBER</b> S4	
				<b>5f. WORK UNIT NUMBER</b> 01	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b>  Two Six Labs 901 N Stuart Street, Suite 1000 Arlington, VA 22203				<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  N/A	
<b>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  <div style="display: flex; justify-content: space-between;"> <div>AFRL/Information Directorate Rome Research Site/RIGA 525 Brooks Road Rome NY 13441-4505</div> <div>DARPA/I20 675 North Randolph Street Arlington, VA 22203</div> </div>				<b>10. SPONSOR/MONITOR'S ACRONYM(S)</b>  AFRL/RI	
				<b>11. SPONSORING/MONITORING AGENCY REPORT NUMBER</b> AFRL-RI-RS-TM-2020-001	
<b>12. DISTRIBUTION AVAILABILITY STATEMENT</b> APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. DARPA DISTAR CASE # 32474 DATE CLEARED: APR 8, 2020					
<b>13. SUPPLEMENTARY NOTES</b> This report was compiled as part of the DARPA Computers and Humans Exploring Software Security (CHESS) program.					
<b>14. ABSTRACT</b>  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.					
<b>15. SUBJECT TERMS</b> Vulnerability Research, Reverse Engineering, Program Analysis, Cyber, Fuzzing, Software Security					
<b>16. SECURITY CLASSIFICATION OF:</b>			<b>17. LIMITATION OF ABSTRACT</b>  UU	<b>18. NUMBER OF PAGES</b>  82	<b>19a. NAME OF RESPONSIBLE PERSON</b> AMANDA P. OZANAM
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			<b>19b. TELEPHONE NUMBER (Include area code)</b> NA

## TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
<b>2</b>	<b>SCOPE OF THE DOCUMENT .....</b>	<b>2</b>
2.1	TOOLS CRITERIA.....	2
2.2	TECHNIQUE CRITERIA .....	3
<b>3</b>	<b>TOOLS AND TECHNIQUES.....</b>	<b>3</b>
3.1	TOOL AND TECHNIQUE CATEGORIES.....	3
3.2	FUZZING .....	4
3.2.1	<i>Section Organization</i> .....	4
3.2.2	<i>American Fuzzy Lop (AFL) Fuzzers</i> .....	5
3.2.2.1	AFL-Unicorn .....	5
3.2.2.2	AFLGo .....	7
3.2.2.3	AFLFast .....	8
3.2.2.4	AFLPlusPlus.....	10
3.2.2.5	WinAFL .....	11
3.2.3	<i>Non-AFL Fuzzers</i> .....	13
3.2.3.1	LibFuzzer.....	13
3.2.3.2	Eclipser .....	14
3.2.3.3	Angora .....	16
3.2.4	<i>Just In Time (JIT) Fuzzers</i> .....	18
3.2.4.1	Fuzzilli .....	18
3.2.4.2	CodeAlchemist.....	21
3.2.5	<i>Fuzzing Frameworks</i> .....	24
3.2.5.1	DeepState.....	24
3.3	DISASSEMBLY AND DECOMPIlation.....	27
3.3.1	<i>IDA Pro (Interactive DisAssembler) and Hex-Rays</i> .....	28
3.3.2	<i>Ghidra</i> .....	31
3.3.3	<i>Binary Ninja</i> .....	35
3.4	STATIC INSTRUMENTATION.....	39
3.4.1	<i>Multiverse</i> .....	39
3.4.2	<i>DDisasm</i> .....	42
3.4.3	<i>LIEF</i> .....	44
3.5	DYNAMIC ANALYSIS AND EXPLOITATION .....	47
3.5.1	<i>Lighthouse</i> .....	47
3.5.2	<i>Frida</i> .....	49
3.5.3	<i>rr</i> .....	51
3.5.4	<i>FUZE</i> .....	52
3.6	SYMBOLIC AND CONCOLIC EXECUTION .....	54
3.6.1	<i>angr</i> .....	55
3.6.1.1	Driller .....	55
3.6.2	<i>Manticore</i> .....	58
3.6.3	<i>QSYM</i> .....	60
3.6.4	<i>MemSight</i> .....	62
<b>4</b>	<b>OTHER TECHNIQUES .....</b>	<b>63</b>
4.1	JIT ATTACKS .....	63
4.2	ROP-BASED ATTACKS .....	65
4.3	HEAP ATTACKS.....	67
4.4	SIDE CHANNEL ATTACKS .....	70
<b>5</b>	<b>CONCLUSION .....</b>	<b>72</b>
<b>6</b>	<b>REFERENCES.....</b>	<b>73</b>

# 1 Introduction

The DARPA Computers and Humans Exploring Software Security (CHESS) program seeks to increase the speed and efficiency with which software vulnerabilities are discovered and remediated, 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 against which the TA1 and TA2 performers will be measured. Vulnerability research is a constantly evolving area of cyber security, which means that the baseline for measuring the success of the CHESS program is 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 the most emerging and trending techniques across all relevant vulnerability classes, tools, and methodologies. 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 the CHECKMATE team considers when planning for the next CHESS evaluation event.

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

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

## 2 Scope of the Document

The main purpose of this first *Edge of the Art (EotA)* report is to define and establish a set of baseline criteria for the “*edge*.” Each subsequent EotA report will document those things that have come into existence (or significantly matured) since the last report. This first report will also discuss several tools that are considered standard-bearers, because newness and novelty are often a result of deficiencies in the current state-of-the-art. In other words, this report presents emergent tools and techniques, as well as the origins of these tools, to make it a foundation for all subsequent reports.

The EotA reports will be 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 CHES program.

### 2.1 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 2014 or later). Over two-thirds of the tools were released in or after 2017, and most of those were released in or after 2018. Those released earlier, such as *Frida* and *rr*, are included because they have significantly matured since their initial release and represent major gains in runtime debugging, interaction, and interoperability.

**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. For example, the fuzzer *Eclipsr* does not necessarily outperform the state-of-the-art, but does represent a promising approach that could complement current state-of-the-art fuzzers. Another example is *FUZE*, a Linux kernel exploitation framework, that, while currently a prototype implementation, provides an interesting new look at exploitation frameworks.

**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. For example, *Lighthouse*, which displays coverage information, did not invent a new

analytical approach, but it enables the user to more intuitively compare and contrast coverage data. Thus, it is included in this report.

**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.

## 2.2 Technique Criteria

Most techniques are implemented by at least one tool, and are documented in that tool's description. The "Other Techniques" section attempts to cover techniques which are not implemented by any tool. They all emerged in 2014 or later and are grouped into categories like Just In Time (JIT) Exploitation Techniques, ROP Techniques, Heap Techniques, etcetera. Areas experiencing heavy research focus but not related to any one of the CHES vulnerability classes or target architectures, like embedded device security and hardware side channels, are considered out of scope and not included in this report.

# 3 Tools and Techniques

The CHECKMATE team has deep experience and expertise with a wide array of different tools and techniques to find, exploit, and remediate vulnerabilities. This draft report will not provide an exhaustive list of tools with detailed description of capabilities and usage. Instead, this section describes a representative subset of the tools and techniques that serves as a baseline for discussing and understanding the edge-of-the-art.

## 3.1 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.

This initial version of the Edge of Art report describes tools in the categories of *Fuzzing*, *Disassembly and Decompilation*, *Static Instrumentation*, *Dynamic Analysis and Exploitation*, and *Symbolic and Concolic Execution*. Subsequent versions of this report will cover advancements to existing tools, as well as new tools and techniques.

## 3.2 Fuzzing

The goal of fuzzing is to search the input space of a program for inputs that trigger a previously unknown vulnerability. Generally, fuzzing assumes no knowledge of where a vulnerability is located. Instead, it assumes that if a vulnerability exists, then it can be triggered with the correct combination of input values, which in turn generates an observable that the fuzzer uses to record its existence. While fuzzing could be considered just another form of dynamic analysis, the plethora of available fuzzers and fuzzing techniques warrants a category all its own.

Fuzzers and fuzzing techniques have evolved in the past three decades. Modern fuzzers are sophisticated automatic input generators and instrumentation engines which draw upon an active field of academic research. Every year scores of new fuzzers and fuzzing research emerge, promising better performance using novel techniques. Fuzzing consistently produces verified vulnerabilities (e.g., CVEs) and has proven to be an extremely effective method of finding vulnerabilities.

The difficulty in comparing fuzzers is a lack of well-defined criteria [1]. American Fuzzy Lop (AFL) is a very popular fuzzer that fuzzing research often uses for comparison. For example, the Eclipser fuzzer uses an interesting technique to generate relatively convincing metrics showing it outperforms AFL [2]. However other researchers found that each fuzzer outperformed the other on different test problems, thereby demonstrating a no one-size-fits-all approach to fuzzing [3].

The Eclipser example shows the difficulties in defining good metrics to measure and compare fuzzing tools and techniques. Fuzzers which do not significantly outperform a state-of-the-art fuzzer on certain test suites may still have utility by offering different coverage paths, different instrumentation options, or different results. This point is summarized succinctly by the security firm Trail of Bits, “In fuzzing, diversity is not just helpful, it is essential if you really want the best chance to find every last bug. No fuzzer will be best for all programs under test, or for all bugs in a given real-world program [3].”

### 3.2.1 Section Organization

The rest of section 3.2 is organized as follows. Section 3.2.2 introduces the state-of-the-art fuzzing tool AFL, along with the most recent improvements and novel approaches to enhance it. Section 3.2.3 describes other non-AFL based fuzzers, such as Eclipser [7] and Angora [8], which



attempt to achieve the benefits of concolic execution without the computational costs by solving path constraints without symbolic execution. Section 3.2.4 describes two Just In Time (JIT) fuzzers that implement novel methods to generate semantically and syntactically correct inputs for JavaScript engines. These techniques are independent of JavaScript and could be repurposed for other fuzzing targets. Section 3.2.5 describes DeepState [9], which is a fuzzing framework that combines fuzzers (AFL, LibFuzzer) and symbolic execution engines (angr) into a common interface [3]. Current research in this fuzzing topic area shows that combining different fuzzing techniques can improve better coverage of a target complex system. It can also make the instrumentation easier on new platforms. These edge-of-the-art enhancements are highlighted in that section.

## 3.2.2 American Fuzzy Lop (AFL) Fuzzers

AFL is an open-source fuzzer that was first released in 2014. Named for the American Fuzzy Lop, a breed of rabbit, it utilizes code coverage and genetic algorithms to generate test cases [11]. AFL has established itself as the enduring state-of-the-art fuzzer upon which many edge-of-the-art variants have been built. Although the latest release was in November 2017, many of the recent significant advances in fuzzing are based on projects that fork the AFL codebase. Discussed below are several of the latest tools that were built upon AFL.

### 3.2.2.1 AFL-Unicorn

<b>Reference Link</b>	<a href="https://github.com/Battelle/afl-unicorn">https://github.com/Battelle/afl-unicorn</a>
<b>Target Type</b>	Binary
<b>Host Operating System</b>	Linux <b>With Constraints:</b> Android; iOS; Windows
<b>Target Operating System</b>	x86 (32, 64)
<b>Host Architecture</b>	x86 (32,64)
<b>Target Architecture</b>	x86 (16, 32, 64); ARM (32, 64); MIPS (32, 64); M68K; SPARC
<b>Initial Release</b>	October 31, 2017
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Nathan Voss

#### Overview

AFL-Unicorn is designed to fuzz arbitrary binary code snippets with AFL using emulation. To mutate inputs, AFL requires an instrumented binary that can generate code coverage information for each input [11]. For applications with standard architectures and operating systems, and straightforward inputs (e.g., an x86 Linux binary that receives input from the command-line), instrumentation is added either during compilation or using AFL's QEMU mode. For less conventional software systems, such as embedded systems or software with unconventional inputs, AFL does not provide a solution. AFL-Unicorn was created in 2017 by Nathan Voss at Batelle to address this problem. By combining Unicorn, an emulation engine built on top of the QEMU emulator, with AFL's QEMU mode, AFL can fuzz arbitrary un-

instrumented code. A Unicorn-based harness is used to emulate the code, while information from the emulator is used to determine code coverage and ultimately mutate inputs [12].

## Design and Usage

AFL-Unicorn works by building a *Unicorn Mode* into AFL using a Unicorn harness to load and emulate the target code. The harness is passed to AFL as an argument, along with typical arguments like input and output paths. A harness must load the binary into memory and set up the initial register and memory state, then AFL-Unicorn will run the emulation and mutate the inputs based on the coverage and crash information returned. While running, the tool interacts with the user as AFL typically does, showing its distinctive command-line display and outputting the inputs that triggered crashes to a specified path [12], [13]. Figure 3-1 illustrates how AFL interacts with Unicorn to achieve the desired result [12].

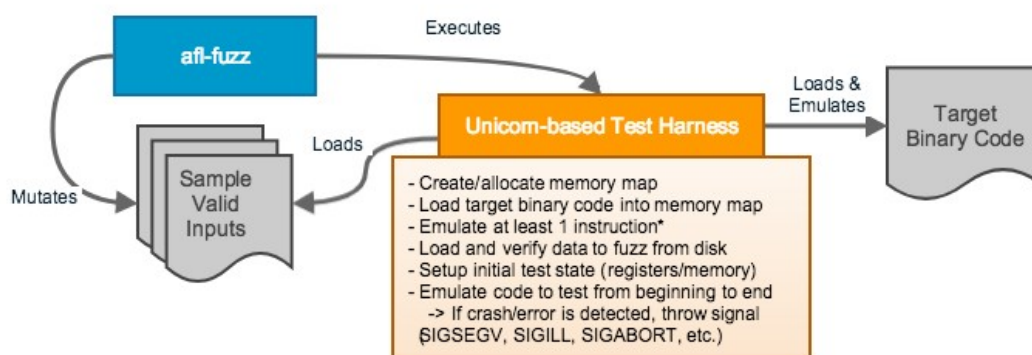


Figure 3-1 AFL-Unicorn architecture [12]

## Use Cases

The significance of Unicorn Mode is the expansion of AFL to fuzz portions of binary code for which source is not available, as well as providing a model for how to combine Unicorn with other fuzzers. Although the amount of reverse engineering needed to run AFL on nonstandard target code is reduced, writing an effective Unicorn harness is nontrivial and Unicorn mode has limitations, such as handling kernel calls and runtime memory allocation. Thus, the primary benefit of AFL-Unicorn is with embedded devices and firmware. On the other hand, AFL-Unicorn can be applied to applications on desktop operating systems when combined with additional AFL-Unicorn modules and significant reverse engineering of the targeted code. AFL-Unicorn has a set of scripts, referred to as *Unicorn Context Dumpers*, which save the state of the program at a specific breakpoint to disk. This state information is the context that is loaded by the harness in order to emulate the code. Context Dumpers exist for several debuggers including IDA Pro, GDB and LLDB. The *Unicorn Loader* module enables the user to manage emulation, including APIs to dump the current register contents and force crashes so AFL can detect them [12], [13].

## Limitations

Despite these promising and helpful additions, the AFL-Unicorn repository has not been updated since July 2018 and is not actively maintained at the time of this report. As an open-source tool, it provides a useful framework and a set of techniques that vulnerability researchers continue to develop. More broadly, it represents a shift in the field of automated program analysis towards fuzzing increasingly large and complex applications [12], [13].

### 3.2.2.2 AFLGo

<b>Reference Link</b>	<b><a href="https://github.com/aflgo/aflgo">https://github.com/aflgo/aflgo</a></b>
<b>Target Type</b>	Source (C/C++/Objective C)
<b>Host/Target Operating System</b>	Linux; BSD <b>With Constraints:</b> macOS, Solaris
<b>Host/Target Architecture</b>	x86 (32, 64); <b>With Modification:</b> ARM (32, 64); PPC (32, 64); MIPS (32, 64); etc.
<b>Initial Release</b>	October 30, 2017
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Last code commit August 2019

## Overview

AFLGo is an open-source implementation of a November 2017 paper by Marcel Bohme, Van-Thuan Pham, Manh-Dung Nguyen and Abhik Roychoudhury of the University of Singapore, called *Directed Greybox Fuzzing* [14]. The paper proposes a technique that centers around a directed fuzzer which “spends most of its time budget on reaching specific target locations without wasting resources stressing unrelated program components [14, p. 1].” Previously, directed fuzzers used symbolic execution, which had a significant performance cost. Directed greybox fuzzing attempts to improve that by creating a directed fuzzer that utilizes the distance between generated seeds and the target locations. The applications of a directed fuzzer envisioned by the creators of this technique are patch testing, crash reproduction, static analysis report verification, and information flow detection [14].

## Design and Usage

AFLGo is based on AFL’s LLVM mode, and uses Bohme et al.’s directed greybox fuzzing technique to create a graph modeling the distance between the input seed and the target. To use AFLGo, the user populates a plaintext file with the target locations, where each target takes the form of a path to a source code file and a line number. As such, AFLGo only works on applications for which there is source code. Next AFLGo generates a set of graphs and distance values. Then it builds the target application with its distance instrumentation. After this, the user runs AFLGo’s version of afl-fuzz on the resulting binary [6], [14].

## Use Cases and Limitations

The most obvious drawback of AFL-Go is that it requires source code. However, it does well as a patch testing tool. As such, it is integrated with the OSS-Fuzz project which seeks to fuzz open-source projects for vulnerabilities. AFLGo does not appear to be under active development, but is actively maintained, with its latest codebase commit in February 2019.

Regardless, the paper on which it was based presented a novel technique for instrumenting applications for directed fuzzing and may be built upon by future tools and techniques [6], [14].

### 3.2.2.3 AFLFast

<b>Reference Link</b>	<a href="https://github.com/mboehme/aflfast">https://github.com/mboehme/aflfast</a>
<b>Target Type</b>	Source (C/C++/Objective C); <b>QEMU Mode:</b> Binary
<b>Host/Target Operating System</b>	Linux; BSD <b>With Constraints:</b> macOS, Solaris
<b>Host Architecture</b>	<b>Primary:</b> x86 (32, 64); <b>With Modification:</b> ARM (32, 64); PPC (32, 64); MIPS (32, 64); etc.
<b>Target Architecture</b>	x86 (32, 64); <b>QEMU Mode:</b> QEMU Supported Architectures
<b>Initial Release</b>	October 28, 2016
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Incorporated into the AFLPlusPlus tool

## Overview

AFLFast was developed by several of the authors of AFLGo (Bohme, Pham, and Roychoudhry) and is based on their paper *Coverage-based Greybox Fuzzing as Markov Chain* [15]. Their goal is to improve coverage-based greybox fuzzing by developing an effective way to identify and use seeds that generate inputs that explore less frequently followed paths. Their technique uses a Markov chain model to develop several new approaches to generating inputs for low frequency paths. AFLFast is a version of AFL that is modified to implement these new approaches. In their paper the authors claim that AFLFast produced more unique crashes than AFL by an order of magnitude [15, p.1].

## Design and Usage

The authors use a Markov chain to model the probability that a seed input, which follows a path  $i$ , will generate a divergent path  $j$ . This model is used to optimize a scoring system or “power schedule [15, p. 2]” for seeds based on their ability to generate inputs that follow low frequency paths. AFL already gives performance scores to seeds based on coverage data and creation time. AFLFast incorporates these scores into their power schedules by assigning scores to seeds in a way that is inversely proportional to the path frequency. This means that a low scoring seed is one that is more likely to

generate inputs which follow a high frequency path, while a high scoring seed is one which follows a low frequency path. The score is then used to determine how many new inputs should be generated from the seed. The paper offers six power schedules, which all differ in how they assign scores to seeds [15].

To implement these power schedules, the authors modified AFL to change how seeds were chosen and ordered. AFL defines a function in *afl-fuzz.c* called *calculate\_score* which calculates the performance score of a seed. In AFLFast the authors modify this function so that users can select a power schedule with an AFL flag at runtime. These six options are as follows:

**EXPLOIT:** This is the original scoring system used by AFL, which is based on metrics such as coverage data and when the seed was created [15, p.7]. The flag to use this power schedule is *-p exploit* [16].

**EXPLORE:** This takes the original scoring system, as described in EXPLOIT, and divides it by a constant value. This keeps all scores low and decreases significant prioritization of certain seeds [15, p.7]. The flag to use this power schedule is *-p explore*.

**FAST:** This is the default schedule used by AFLFast. It is calculated using the original scoring system described in EXPLOIT but it also uses another metric, the path frequency of the seed's path. This generates a score in inverse proportion to that path frequency, which means seeds that follow high frequency paths will get lower scores [15, p.7]. The flag to use this power schedule is *-p fast* [16].

**LINEAR:** Like FAST but increases scores in a linear manner [15, p.7]. The flag to use this power schedule is *-p lin* [16].

**QUAD:** Like FAST but increases scores in a quadratic manner [15, p.8]. The flag to use this power schedule is *-p quad* [16].

**COE:** Like FAST, but it does not fuzz any seed with a path frequency above a certain threshold. This means that seeds that follow paths that are followed at a high enough frequency will not be fuzzed until that frequency is lower than the threshold [15, p.7]. The flag to use this power schedule is *-p coe* [16].

AFLFast also implements new search strategies to determine the order in which seeds are chosen. It prioritizes seeds which have been chosen less often and that follow low frequency paths [15, p.8].

## Use Cases and Limitations

One contribution of this paper is modeling seed mutations with Markov chains and using it to reason about scoring seeds. The paper also demonstrates that the AFLFast approach produced more unique crashes than AFL by an order of magnitude [15, p.1].

A recent paper, *Evaluating Fuzz Testing* [83], found that AFLFast did not outperform AFL in other testcases not used in the original AFLFast paper, suggesting that “evidence of AFLFast’s superiority... was weakened.” However after the AFLFast paper was published, some of the AFLFast improvements had been incorporated into AFL and *Evaluating Fuzz Testing* used a version of AFL that included these change. The paper states that because of this their “goal is not to reproduce AFLFast’s results [83, p.4].”

The code for AFLFast has been integrated into the actively maintained repository of AFLPlusPlus, which builds a number of additional patches into a fork of AFL [16].

### 3.2.2.4 AFLPlusPlus

<b>Reference Link</b>	<a href="https://github.com/vanhauser-thc/AFLplusplus">https://github.com/vanhauser-thc/AFLplusplus</a>
<b>Target Type</b>	Source (C/C++/Objective C); <b>QEMU Mode:</b> Binary
<b>Host/Target Operating System</b>	Linux; BSD <b>With Constraints:</b> macOS, Solaris
<b>Host Architecture</b>	<b>Primary:</b> x86 (32, 64); <b>With Modification:</b> ARM (32, 64); PPC (32, 64); MIPS (32, 64); etc.
<b>Target Architecture</b>	x86 (32, 64); <b>QEMU Mode:</b> QEMU Supported Architectures
<b>Initial Release</b>	June 4, 2019
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Marc Heuse and Heiko Eissfeldt

#### Overview

AFL’s popularity has resulted into a slew of fuzzers that fork the AFL codebase to add a small number of enhancements or features. These improvements and features are dispersed across these various AFL-based fuzzers, and many of the fuzzers are not actively maintained. AFL itself has not been actively maintained since 2017. These challenges were the impetus for AFLPlusPlus, an actively maintained version of AFL which integrates patches written by the community. For instance, it uses AFLFast’s capabilities, and maintains AFL and its dependencies, such as QEMU. As of the writing of this report, AFL does not work with QEMU v3.1, however AFLPlusPlus upgraded AFL in this regard [17]. This tool is actively maintained by Marc Heuse and Heiko Eissfeldt.

#### Design and Usage

AFLPlusPlus incorporates the changes made by AFLFast and adds support for QEMU v3.1. Additionally, a variety of other enhancements have been made to this version of AFL. These include LLVM modifications to only instrument relevant blocks, and QEMU modifications to enable caching. AFLPlusPlus incorporates the laf-intel tool, which increases code coverage in LLVM mode, and also adds support for PowerPC.

The basic usage of AFLPlusPlus is the same as that of AFL [17].

### Use Cases and Limitations

AFLPlusPlus can be used as a drop-in replacement for AFL. It is actively maintained, has up-to-date support for dependencies like QMEU, and also contains suggested improvements to AFL. However, a user may prefer to use the original version of AFL if they want to avoid certain optimizations or changes made in AFLPlusPlus.

#### 3.2.2.5 WinAFL

<b>Reference Link</b>	<a href="https://github.com/googleprojectzero/winafl">https://github.com/googleprojectzero/winafl</a>
<b>Target Type</b>	Source (C/C++/Objective C)
<b>Host/Target Operating System</b>	Windows
<b>Host/Target Architecture</b>	x86 (32, 64);
<b>Initial Release</b>	July 7, 2016
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Ivan Fratric

### Overview

AFL is designed for Unix-like systems and uses instrumentation mechanisms that do not work on Windows. To address this limitation, Google’s Project Zero designed WinAFL, a fuzzer built on AFL that implements several alternative instrumentation approaches designed for Windows. The three instrumentation modes are dynamic instrumentation with DynamoRIO, hardware tracing with Intel Processor Trace Tools, and static instrumentation with Syzgy. One of the key capabilities of WinAFL is persistent fuzzing mode, which takes a target function specified by the user and instruments it to run in a loop [5].

### Design and Usage

The design of WinAFL can best be understood by examining each of the three instrumentation modes.

**DynamoRIO Instrumentation Mode** – This mode uses DynamoRIO, which is a dynamic instrumentation platform to measure and extract target code coverage [18]. It takes a number of possible instrumentation options, including a target function (the function you wish to fuzz), a target module (the module containing the target function), a coverage module (the module for which coverage is recorded), the coverage type to be recorded, and the maximum number of iterations to loop the target function. These options are in addition to a subset of AFL options which are supported. This mode also enables ‘In App Persistence’ mode which, rather than looping the target function, assumes a loop is built into the program and restarts instrumentation when it reencounters the function [5], [18].

**Intel PT Mode** – This mode leverages Intel’s Processor Tracing (PT) capability, which enables the CPU to generate tracing instructions natively. As the GitHub README describes, “When a target is fuzzed with WinAFL in Intel PT mode, WinAFL opens the target in a debugger. The debugger implements the WinAFL persistence (looping over target function without the need to restart the process for every iteration), monitors for crashes, loaded modules etc. Before every iteration, the debugger enables Intel PT tracing for the target process and, after the iteration finishes, the trace is retrieved and analyzed, updating the AFL coverage map [19].”

**Static Instrumentation via syzygy** – Syzygy is a framework to decompose a 32-bit Portable Executable (PE). As the GitHub README describes, “*Decomposing* a binary is the term used to mean taking in input a PE32 binary and its Program Database file (PDB), analyze and decompose every function, every block of code / data in a safe way and present it to transformation ‘passes’. A transformation pass is a class that transforms the binary in some way. Once the pass has transformed the binary, it passes it back to the framework which is able to relink an output binary (with the transformations applied of course) [20].”

### **Use Cases and Limitations**

The explicit use case for WinAFL is fuzzing Windows applications, as AFL does not support these applications by default. However, constructing an effective target function, which is necessary to use WinAFL, can be difficult. The documentation for WinAFL’s DynamoRIO Instrumentation Mode states that “[i]n some applications it's quite challenging to find a target function that with a simple execution redirection won't break global states and will do both reading and processing of inputs [5], [18].” This may require relatively advanced Windows internals knowledge. These additional requirements complicate WinAFL usage and may lead to problems, such as a target function loading a DLL at every iteration, significantly slowing down execution. Therefore, fuzzing a specified function may require significant modification to the Windows binary, which can be difficult.



## 3.2.3 Non-AFL Fuzzers

### 3.2.3.1 LibFuzzer

Reference Link	<a href="https://llvm.org/docs/LibFuzzer.html">https://llvm.org/docs/LibFuzzer.html</a>
Target Type	Source
Host/Target Operating System	Linux; BSD; macOS; Windows
Host/Target Architecture	x86 (32, 64) <b>With Modification:</b> ARM (32, 64); MIPS (32, 64)
Initial Release	2015
License Type	Open-Source
Maintenance	Maintained by the LLVM Project

#### Overview

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine developed by LLVM and built into Clang. The key component of LibFuzzer is its integration with LLVM's Sanitizer Suite to enable coverage instrumentation. Additionally, as an in-process fuzzer, LibFuzzer does not start a new process for every test case. In use since at least 2015, LibFuzzer is actively maintained, and continues to be updated with new capabilities [21].

#### Design and Usage

LibFuzzer is now a built-in capability of Clang, and runs on a *fuzz target*, which is defined by LLVM as “a function that accepts an array of bytes and does something interesting with these bytes using the API under test.” Figure 3-2 shows an example fuzz target given in the LLVM documentation. [21].

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    DoSomethingInterestingWithMyAPI(Data, Size);
    return 0; // Non-zero return values are reserved for future use.
}
```

Figure 3-2 An example fuzzing target form the LLVM documentation [21]

Prior to compiling the target function, it is also necessary to compile all of its dependencies with Clang using the `-fsanitize=fuzzer` flag, as well as the Clang capabilities `AddressSanitizer` and `UndefinedBehaviorSanitizer`. Compiling with these capabilities is what enables LibFuzzer's coverage guidance. After compilation and corpus generation, the target binary is executed with an array of flags that control different aspects of how the fuzzer will behave [21].

#### Use Cases and Limitations

In their documentation, LLVM suggests use cases for LibFuzzer, “This Fuzzer might be a good choice for testing libraries that have relatively small inputs, each input takes < 10ms to run,

and the library code is not expected to crash on invalid inputs. Examples: regular expression matchers, text or binary format parsers, compression, network, crypto [21].”

The in-process nature of this fuzzer means that excessive memory consumption and infinite loops in the target library are hard to protect against. The most significant drawback, however, is that as a built-in capability of Clang, LibFuzzer must be run on source code compiled with Clang, which excludes prebuilt binaries (Trail of Bits has attempted, somewhat successfully, to apply LibFuzzer to a binary lifted to LLVM bytecode using their McSema tool [22]). However, this can also be considered a benefit, because LibFuzzer has an edge over fuzzers without built-in sanitizer support, like AFL. LibFuzzer also enables easy viewing of coverage with Clang Coverage. However, the benefits may be offset by LibFuzzer requiring more setup and harness development than AFL [21].

### 3.2.3.2 Eclipser

<b>Reference Link</b>	<a href="https://github.com/SoftSec-KAIST/Eclipser">https://github.com/SoftSec-KAIST/Eclipser</a>
<b>Target Type</b>	Binary
<b>Host/Target Operating System</b>	Linux
<b>Host/Target Architecture</b>	x86 (32, 64)
<b>Initial Release</b>	May 31, 2019
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by SoftSec Lab at KAIST

#### Overview

Modern day fuzzers are designed to analyze code coverage data from previous runs and produce new inputs that will explore new code paths. However, there is no guarantee that the new code paths will be significantly different from the previous path. Symbolic execution, by comparison, explores all possible paths but incurs considerable compute and memory costs. Hybrid fuzzing has emerged as a technique to leverage the benefits of both fuzzing and symbolic execution while minimizing their respective costs (e.g., Driller [23]). However, many hybrid fuzzers still suffer from the significant overhead costs of symbolic execution [2].

Eclipser hopes to provide an alternative approach that increases code coverage through means inspired by hybrid fuzzing, but without any of the costs of symbolic execution. This fuzzer is based on the ICSE 2019 paper *Grey-box Concolic Testing on Binary Code* by Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha [2]. Eclipser approximates path constraints rather than solving for them, which significantly increases code coverage in certain cases. They demonstrate that on LAVA-M benchmarks and real Debian packages Eclipser finds more bugs than AFLFast and laf-intel [2]. Other tests by Trail of Bits have shown slightly worse performance than AFL overall [3].

## Design and Usage

Instead of relying on SMT solvers (which are notoriously computationally expensive) to solve path constraints, Eclispr approximates them. Doing so results in constraints which are intended to be solved much faster [2].

Eclipser is written primarily in F# and run on .NET Core. An example of basic usage given in the README is below [7]:

```
dotnet build/Eclipser.dll fuzz \
    -p <target program path> -v <verbosity level> -t <timeout second>
\
```

## Use Cases and Limitations

From a technical perspective, a possible criticism of their approach is its lack of precision, which Eclipsr addresses in the paper:

“Naturally, the path constraints generated from Grey-Box concolic testing are imprecise, but, in practice, they are precise enough to quickly explore diverse execution paths. The primary design decision here is to trade off simplicity for precision [2, p. 2].”

As a practical matter, the metrics offered in the paper are promising, and their evaluation techniques are robust. However, other evidence shows that Eclipser does not outperform AFL in certain cases. Trail of Bits tested Eclipser against AFL and LibFuzzer with their tool DeepState and found that,

“While Eclipser is exciting, our preliminary tests indicate that it performs slightly worse than everyone’s favorite workhorse fuzzer, AFL, on both the file system and red-black-tree. In fact, even with the small set of testing problems we’ve explored in some depth using DeepState, we see instances where Eclipser performs best, instances where LibFuzzer performs best, and instances where AFL performs best. Some bugs in the red black tree required a specialized symbolic execution test harness to find (and Eclipser does not help, we found out). Moreover, even when one fuzzer performs best overall for an example, it may not be best at finding some particular bug for that example [3].”

Eclipser is still a helpful addition and can be used in tandem with AFL to increase path coverage and find more bugs.

### 3.2.3.3 Angora

Reference Link	<a href="https://github.com/AngoraFuzzer/Angora">https://github.com/AngoraFuzzer/Angora</a>
Target Type	Source (C/C++)
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	March 28, 2019
License Type	Open-Source
Maintenance	Maintained by Byte Dance AI Lab

#### Overview

Angora is another AFL-inspired fuzzer that aims to increase code coverage by solving path constraints without symbolic execution. It uses similar AFL code coverage metrics and instrumentation that includes byte-level taint analysis, function call context tracking, input length exploration, and data shape and type inference. The byte-level taint analysis and data width and type inference enable Angora to precisely manipulate subsections of an input. These metrics add constraints to memory that can be solved by searching for inputs using gradient descent instead of symbolic execution [8], [24]. Based on the authors' results, Angora "found eight times as many bugs as the second best-fuzzer while fuzzing the program who" [24].

#### Design and Usage

Fuzzing with Angora is a two-step process: instrumentation and execution. Similar to AFL, it first instruments branches to trace execution, but extends the instrumentation to include context sensitive transitions, such as calling a function with different arguments. It then associates all variables in a program with a taint label that describes what byte offset in an input flows into each variable. The taint labels are then stored into a tree data structure with nodes that represent each taint label, byte width, and inferred type. As variable relationships are discovered they are added to this tree. Angora's key insight is that this taint analysis only needs to be computed once, amortizing the cost of the analysis over the fuzzing duration [8], [24].

Once taint analysis and tracing instrumentation has completed, fuzzing can begin. However, effectively generating a seed requires repeatedly solving path constraints. Angora's key insight is that all conditional statements can be represented as either  $<$ ,  $<=$ , or  $==$  relationships. To solve for these constraints two traces are needed - a control and a mutation whose traces are inspected for differences. If the traces are the same, there has been no new coverage and the variable is randomly mutated. If the traces differ at earlier points in execution, a derivative is calculated representing the rate of change for the inspected variable. The more quickly the traces diverge, the steeper the gradient becomes, indicating the need for greater mutation. Eventually the variable is mutated to such a degree that it satisfies either  $<$ ,  $<=$  or  $==$  constraints and execution continues [8], [24].

To fuzz an application, one must first compile two files: a taint analysis file and an instrumented file. These files are then provided to Angora for execution. The recommended compilation and execution steps are shown below [8].

```

# Use the instrumenting compilers
CC=/path/to/angora/bin/angora-clang \
CXX=/path/to/angora/bin/angora-clang++ \
LD=/path/to/angora/bin/angora-clang \
PREFIX=/path/to/target/directory \
./configure --disable-shared

# Build with taint tracking support
USE_TRACK=1 make -j
make install

# Save the compiled target binary into a new directory
# and rename it with .taint postfix, such as uniq.taint

# Build with light instrumentation support
make clean
USE_FAST=1 make -j
make install

# Save the compiled binary into the directory previously
# created and rename it with .fast postfix, such as uniq.fast

```

```
./angora_fuzzer -i input -o output -t path/to/taint/program -- path/to/fast/program [argv]
```

## Use Cases and Limitations

One limitation of Angora is that it can only operate on source code. Another limitation is that its mutation engine lacks the capabilities of AFL. Therefore, it is suggested to combine them, leveraging AFL's mutation capabilities and speed with Angora's analyses. This can be done by running both in parallel as shown in the following [8].

```

~/afl/afl-fuzz -i seeds -o output -S afl_s -- ./afl/install/bin/file @@

# --sync_afl to allow sync seeds with AFL
# -A to disable AFL's random mutation in Angora.
~/angora/angora_fuzzer --sync_afl -A -i seeds -o output -t ./track/install/bin/file
-- ./fast/install/bin/file -m ./fast/install/share/misc/magic.mgc @@

```

### 3.2.4 Just In Time (JIT) Fuzzers

The following two fuzzers, Fuzzilli and CodeAlchemist are designed primarily for fuzzing JavaScript JIT compilers, which have emerged as a major target in attacks on browsers because a JIT compiler generates executable code. Fuzzing a core interpreter like a browser's JIT compiler presents an additional challenge: generation of semantically correct inputs. Mutating a large number of samples and testing for validity is not a sufficient solution, because the chance of a correct mutation is extremely low. This challenge can be addressed in multiple ways, two of which are implemented by Fuzzilli and Code Alchemist.

#### 3.2.4.1 Fuzzilli

<b>Reference Link</b>	<a href="https://github.com/googleprojectzero/fuzzilli">https://github.com/googleprojectzero/fuzzilli</a>
<b>Target Type</b>	Browser JIT Engine
<b>Targets</b>	JavaScript; Spidermonkey; v8
<b>Host/Target Operating System</b>	Linux; macOS
<b>Host/Target Architecture</b>	x86 (32, 64)
<b>Initial Release</b>	March 20, 2019
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Samuel Groß

#### Overview

Fuzzilli is an actively-maintained, open-source fuzzer, released in March 2019 by Samuel Groß of Google Project Zero, and intended primarily for fuzzing JavaScript JIT compilers. It is written in Swift and addresses the challenge of generating semantically correct JavaScript programs by creating a custom intermediate language called FuzzIL that is translated to executable JavaScript.

Fuzzilli has found a significant number of vulnerabilities in different browser's JIT compilers including JavaScriptCore (Webkit), Spidermonkey (Mozilla) and v8 (Chromium) [25]-[27].

#### Design and Usage

FuzzIL is “defined on which mutations to the control and data flow of a program can more directly be performed” [25] which is then translated to executable JavaScript. FuzzIL is a list of instructions, which themselves are operations that have input and output variables (as opposed to immediate values which are never used as inputs). Instructions can sometimes take parameters enclosed by single quotation marks. A (rough) example of FuzzIL and subsequently translated JavaScript is shown below.

FuzzIL [25]:

```
v0 <- LoadInt '0'
v1 <- LoadInt '10'
v2 <- LoadInt '1'
v3 <- Phi v0
BeginFor v0, '<', v1, '+', v2 -> v4
    v6 <- BinaryOperation v3, '+', v4
    Copy v3, v6
EndFor
v7 <- LoadString 'Result: '
v8 <- BinaryOperation v7, '+', v3
v9 <- LoadGlobal 'console'
v10 <- CallMethod v9, 'log', [v8]
```

which crudely translates into this JavaScript [25]:

```
const v0 = 0;
const v1 = 10;
const v2 = 1;
let v3 = v0;
for (let v4 = v0; v4 < v1; v4 = v4 + v2) {
    const v6 = v3 + v4;
    v3 = v6;
}
const v7 = "Result: ";
const v8 = v7 + v3;
const v9 = console;
const v10 = v9.log(v8);
```

FuzzIL code can be mutated via a number of mutation scripts. *CombineMutator* and *SpliceMutator* are the simpler ones: they combine multiple programs. *InputMutator* and *OperationMutator* changes instruction input values and operation parameters, respectively. *InsertionMutator* is more complex and uses a list of predefined code generators to create new code and place it randomly in the program [25]-[27].

Fuzzilli works from a *Corpus* which saves FuzzIL samples that produced interesting results, that is, results which explore new code paths. The *Corpus* will add or evict samples based on the *Evaluator*, which evaluates the execution that the sample produced. These FuzzIL samples are supplied to the *FuzzerCore*, which uses them to produce new samples using the above mutation techniques, among others. Then, it evaluates the samples for semantic correctness. The *Lifter* translates the mutation into JavaScript and the *ScriptRunner* executes it, aided by an *Environment* script which contains information about the execution environment [25]-[27].

Fuzzilli is run by first compiling the target JavaScript engine with instrumentation and then building the fuzzer with Swift. Finally, using Swift to run the fuzzer on the target [25].



The command line usage as defined by comments in Fuzzilli/Sources/FuzzilliCli/main.swift:

```
Usage:
\(args.programName) [options] --profile=<profile> /path/to/jsshell

Options:
--profile=name           : Select one of several preconfigured profiles.
                          Available profiles: \(profiles.keys).
--logLevel=level         : The log level to use. Valid values: "verbose", "info", "warning", "error", "fatal"
                          (default: "info").
--numIterations=n        : Run for the specified number of iterations (default: unlimited).
--timeout=n              : Timeout in ms after which to interrupt execution of programs (default: 250).
--minMutationsPerSample=n : Discard samples from the corpus after they have been mutated at least this
                          many times (default: 16).
--minCorpusSize=n        : Keep at least this many samples in the corpus regardless of the number of times
                          they have been mutated (default: 1024).
--maxCorpusSize=n        : Only allow the corpus to grow to this many samples. Otherwise the oldest samples
                          will be discarded (default: unlimited).
--consecutiveMutations=n : Perform this many consecutive mutations on each sample (default: 5).
--minimizationLimit=n    : When minimizing corpus samples, keep at least this many instructions in the
                          program. See Minimizer.swift for an overview of this feature (default: 0).
--storagePath=path       : Path at which to store runtime files (crashes, corpus, etc.) to.
--exportState            : If enabled, the internal state of the fuzzer will be written to disk every
                          6 hours. Requires --storagePath.
--importState=path       : Import a previously exported fuzzer state and resuming fuzzing from it.
--networkMaster=host:port : Run as master and accept connections from workers over the network. Note: it is
                          *highly* recommended to run network fuzzers in an isolated network!
--networkWorker=host:port : Run as worker and connect to the specified master instance.
```

The user can choose two modes of execution. *Forkserver* mode forks a new child for every new sample (similar to AFL), and *Read-Eval-Print-Reset-Loop* (REPR) mode receives the script over an IPC and resets the internal state after it executes the script. The latter mode is faster [25].

## Use Cases and Limitations

This fuzzer was very specifically designed for JavaScript engines in browsers. The techniques developed for these fuzzers could be applied to other instances in which semantically correct inputs are necessary and hard to randomly generate, such as WASM execution. Browsers have become a major target for exploitation, and fuzzing techniques such as these are increasingly necessary [25]-[27].

Fuzzilli's approach to generating JavaScript code has the benefit of being able to theoretically explore all possible patterns given enough computing power. This is not the case with a hardcoded generator, like CodeAlchemist.

However, a major limitation of the Fuzzilli approach is that it uses more computing power to generate inputs, as compared to Code Alchemist [25], [28].

The most notable benefits of Fuzzilli are its user friendliness and flexibility. Its rich set of options enable a user to control how Fuzzilli will operate based on their specific needs. These are areas in which CodeAlchemist, by comparison, falls short.



## Future Work

Fuzzilli is actively maintained, supported by Google Project Zero, and continues to find vulnerabilities in modern browsers. Groß, its author, has indicated that going forward he hopes to implement a “compiler” to translate JavaScript to FuzzIL and extend FuzzIL’s language capabilities, among other improvements [27].

### 3.2.4.2 CodeAlchemist

<b>Reference Link</b>	<a href="https://github.com/SoftSec-KAIST/CodeAlchemist">https://github.com/SoftSec-KAIST/CodeAlchemist</a>
<b>Target Type</b>	Browser JIT Engine
<b>Targets</b>	JavaScript; Spidermonkey; v8; ChakraCore
<b>Host/Target Operating System</b>	Linux
<b>Host/Target Architecture</b>	x86 (32, 64)
<b>Initial Commit</b>	March 29, 2018
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by SoftSec Lab at KAIST

## Overview

While Fuzzilli uses a detailed Intermediate Language (IL), CodeAlchemist leverages Abstract Syntax Trees (ASTs) to create an algorithmic approach, called “semantics-aware assembly”, to generate semantically and syntactically correct JavaScript. Working from a set of JavaScript seeds, it breaks them into fragments, termed ‘code bricks’, which represent valid ASTs. Then a set of constraints is applied to each code brick in order to combine code bricks, mixing and matching them to create new samples [28].

Although some runtime errors still occur in the samples produced, this method significantly minimizes this. As a novel approach which will generate a different set of inputs it provides a complementary alternative to Fuzzilli [28].

To run CodeAlchemist, the user first prepares the seeds and configuration and then uses the .NET Core to execute the fuzzer [29].

## Design and Usage

CodeAlchemist first parses a JavaScript seed into a set of valid ASTs. It then defines pre- and post- conditions to split these into code bricks. These are then instrumented and combined with one another based previous executions to generate new inputs [28].

Figure 3-3, Figure 3-4, Figure 3-5 are from the CodeAlchemist paper.

Figure 3-3 shows its architecture.

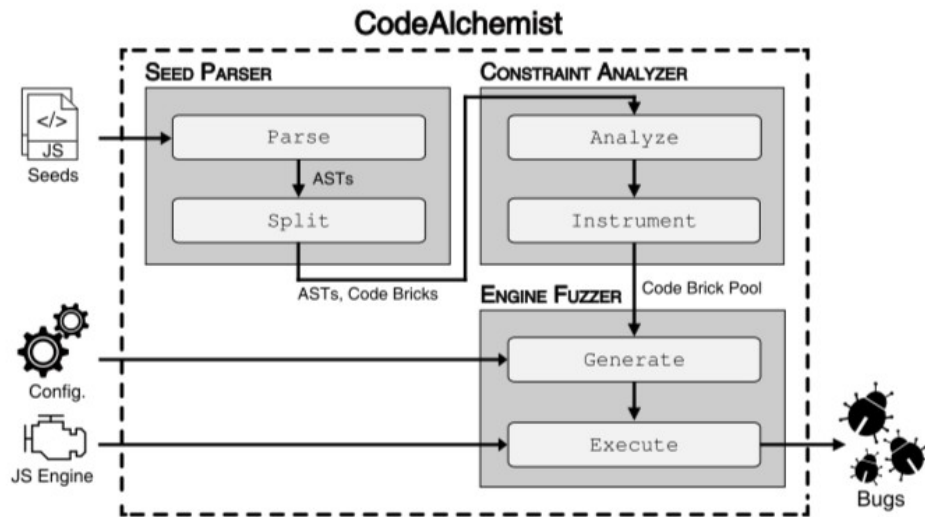
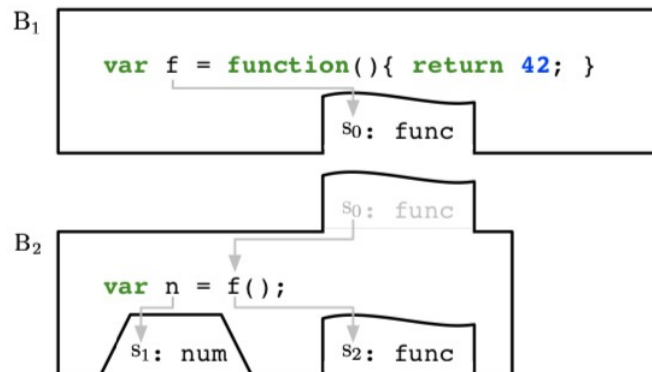


Figure 3-3: Architecture of CodeAlchemist [28, Fig. 5]

Figure 3-4 shows how code bricks are defined.

```
1  var f = function () { return 42; };
2  var n = f();
```

(a) A sample JS seed with two statements. We assume that we create one code brick for each statement, respectively.



(b) Two code bricks obtained from the seed. The teeth and holes on each code brick represent the assembly constraints.

Figure 3-4: Definition of code bricks [28, Fig. 4]

Figure 3-5 gives an example of an input and the seed from which it was generated.

```
1  var n = 42; // Var1
2  var arr = new Array(0x100); // Var2
3  for (let i = 0; i < n; i++) // For3-0, For3-1
4  { // Block4
5      arr[i] = n; // Expr5
6      arr[n] = i; // Expr6
7  }
```

(a) An example JS code snippet used as a seed.

```
1  var s0 = new Array(0x100); // Var2
2  var s1 = 42; // Var1
3  for (let s2 = 0; s2 < s1; s2++) { // For3-1
4      for (let s3 = 0; s3 < s2; s3++) { // For3-0
5          s0[s3] = s2;
6          s0[s2] = s3;
7      }
8  }
```

(b) A generated code snippet from the seed.

*Figure 3-5: Examples of input seeds [28, Fig. 6]*

Code Alchemist is written in primarily F# and is run on .NET Core. The user creates a configuration file with the format described in Figure 3-6:

A configuration file is json file with following fields. And you can find example configuration files for 4 JS engines.

- `engine` : Type of JS engine ("Charka", "JSC", "MOZ", "V8").
- `engine_path` : ABSPATH to engine
- `timeout` : Timeout for executing a JS code.
- `argv` : Additional arguments for executing a JS engine.
- `env` : Additional environment variables for executing a JS engine.
- `seed_path` : ABSPATH to seed.
- `preproc_dir` : ABSPATH for saving preprocessing results.
- `tmp_dir` : ABSPATH for temporarily saving generated JS code.
- `bug_dir` : ABSPATH for saving JS code which triggered some crash.
- `filters` : List of symbols to exclude from the code brick pool.
- `jobs` : The number of jobs (cores) to use for fuzzing.

*Figure 3-6: CodeAlchemist/conf/README.md [29]*

The user first preprocesses JavaScript input seeds by calling

```
dotnet bin/Main.dll rewrite <absolute path to configuration file>
```

Then the user fuzzes with

```
dotnet bin/Main.dll fuzz <absolute path to configuration file>
```

## Use Cases and Limitations

This fuzzer was very specifically designed for JavaScript engines in browsers. The techniques developed for these fuzzers could be applied to other instances in which semantically correct inputs are necessary and hard to randomly generate, such as WASM execution. Browsers have become a major target for exploitation, and fuzzing techniques such as these are increasingly necessary [25]-[27].

Fuzzilli's approach to generating JavaScript code has the benefit of being able to theoretically explore all possible patterns given enough computing power. This is not the case with a hardcoded generator, like CodeAlchemist.

However, a major limitation of the Fuzzilli approach is that it uses more computing power to generate inputs, as compared to Code Alchemist [25], [28].

The most notable benefits of Fuzzilli are its user friendliness and flexibility. Its rich set of options enable a user to control how Fuzzilli will operate based on their specific needs. These are areas in which CodeAlchemist, by comparison, falls short.

## 3.2.5 Fuzzing Frameworks

### 3.2.5.1 DeepState

<b>Reference Link</b>	<a href="https://github.com/trailofbits/deepstate">https://github.com/trailofbits/deepstate</a>
<b>Target Type</b>	Source (C/C++)
<b>Host/Target Operating System</b>	Linux
<b>Host/Target Architecture</b>	x86 (32, 64)
<b>Initial Release</b>	2018
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Trail of Bits

## Overview

DeepState is a new tool from Trail of Bits designed to be a symbolic execution and fuzzing framework for C and C++. It provides developers with a common framework to use tools integrated into DeepState, which at the time of this report were Manticore, angr, LibFuzzer, AFL and Eclipser. The project intends to make it easier to not just create a common interface for fuzzing but to “make a series of push-button front-ends to promising tools that require more work to apply than AFL or LibFuzzer.” This was one of the motivations behind adding Eclipser and angr [3].

## Design and Usage

It is built as a Python library which can be used to write test harnesses [9]. An example of one such harness is given in the README and shown in Figure 3-7.

```
#include <deepstate/DeepState.hpp>

using namespace deepstate;

/* Simple, buggy, run-length encoding that creates "human readable"
 * encodings by adding 'A'-1 to the count, and splitting at 26.
 * e.g., encode("aaabbbbbc") = "aCbEcA" since C=3 and E=5 */

char* encode(const char* input) {
    unsigned int len = strlen(input);
    char* encoded = (char*)malloc((len*2)+1);
    int pos = 0;
    if (len > 0) {
        unsigned char last = input[0];
        int count = 1;
        for (int i = 1; i < len; i++) {
            if (((unsigned char)input[i] == last) && (count < 26))
                count++;
            else {
                encoded[pos++] = last;
                encoded[pos++] = 64 + count;
                last = (unsigned char)input[i];
                count = 1;
            }
        }
        encoded[pos++] = last;
        encoded[pos++] = 65; // Should be 64 + count
    }
    encoded[pos] = '\0';
    return encoded;
}

char* decode(const char* output) {
    unsigned int len = strlen(output);
    char* decoded = (char*)malloc((len/2)*26);
    int pos = 0;
    for (int i = 0; i < len; i += 2) {
        for (int j = 0; j < (output[i+1] - 64); j++) {
            decoded[pos++] = output[i];
        }
    }
    decoded[pos] = '\0';
    return decoded;
}

// Can be (much) higher (e.g., > 1024) if we're using fuzzing, not symbolic execution
#define MAX_STR_LEN 6

TEST(Runlength, BoringUnitTest) {
    ASSERT_EQ(strcmp(encode(""), ""), 0);
    ASSERT_EQ(strcmp(encode("a"), "aA"), 0);
    ASSERT_EQ(strcmp(encode("aaabbbbbc"), "aCbEcA"), 0);
}

TEST(Runlength, EncodeDecode) {
    char* original = DeepState_CStrUpToLen(MAX_STR_LEN, "abcdef0123456789");
    char* encoded = encode(original);
    ASSERT_LE(strlen(encoded), strlen(original)*2) << "Encoding is > length*2!";
    char* roundtrip = decode(encoded);
    ASSERT_EQ(strncmp(roundtrip, original, MAX_STR_LEN), 0) <<
        "ORIGINAL: '" << original << "', ENCODED: '" << encoded <<
        "'", ROUNDTRIP: '" << roundtrip << "'";
}
```

Figure 3-7: An example test harness [9]

## **Use Cases and Limitations**

The goal of DeepState is to make it easier to fuzz C and C++ programs with just one framework but many fuzzers, and to simplify less user-friendly tools such as Eclipser. Its main appeal is that it enables the user to write only one harness and then reuse it with multiple fuzzers. DeepState also includes capabilities that aid fuzzing and can be expanded to include more.

### 3.3 Disassembly and Decompilation

Disassemblers and decompilers are two primary tools used in reverse engineering. A disassembler translates a program's machine code into assembly language instructions. A decompiler converts a program's machine code into 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. The difference is that a decompiler generates something closer to the original source code, but is not guaranteed to create a semantically faithful representation of the underlying machine code (due to the number of transformations applied). A disassembler, on the other hand, is performing a direct conversion from a binary representation of the program to the lowest level human readable form. While errors can occur in this translation, it is much less likely.

For over a decade, IDA Pro, with its decompiler Hex-Rays, was the de facto tool for reverse engineering. In 2016, Vector 35 released Binary Ninja, a reverse engineering platform with a user interface and program analysis capabilities that made it an attractive alternative to IDA Pro. However, Binary Ninja could not fully compete with IDA Pro because it lacked a decompiler at the time of its release. Binary Ninja's decompiler is under active development and is expected to be released in the near future.

In March of 2019, the United States National Security Agency (NSA) released Ghidra, a disassembler and decompiler with comparable performance to IDA Pro. Originally developed as an internal NSA tool, Ghidra is now free open-source software, which contrasts with IDA Pro's significant license fees as proprietary software. Having a well-tested open-source disassembler/decompiler (Ghidra has been actively developed and used by the NSA for over a decade) was seen as a benefit to a community which had for years relied on a single closed-source, albeit well performing, product.

The entrance of these new competitors is already having noticeable effects. The program analysis capabilities at the core of the Binary Ninja product has created a strong differentiator from IDA Pro, even without a decompiler. Their license fees are also significantly less than IDA Pro, and they have publicly released their Low- and Medium-Level Intermediate Languages. Since then, IDA Pro finally made available their Intermediate Representation, Microcode. After Ghidra was released with a built-in 'Undo' capability, IDA Pro added an 'Undo' capability as well (something IDA Pro has notoriously lacked). However, IDA Pro has been the state-of-the-art, not just due to lack of competition, but because of its high quality and accuracy, which came from years of expert development and careful tuning.

The following sections introduce and compare each of these tools. While none of them clearly represent the edge-of-the-art, each one continues to evolve and introduce new advances that help define the edge-of-the-art in decompilation and binary analysis. These sections will examine how each of these tools has contributed to this area in recent years, and how the increased competition in this space will affect future innovations.

### 3.3.1 IDA Pro (Interactive DisAssembler) and Hex-Rays

<b>Reference Link</b>	<a href="https://www.Hex-Rays.com/">https://www.Hex-Rays.com/</a>
<b>Target Type</b>	Binary
<b>Host Operating System</b>	Linux; macOS; Windows
<b>Target Operating System</b>	<b>Windows:</b> PE (Portable Executable); MS DOS/MS DOS Driver/MS DOS Com; Windows Crash Dump; etc. <b>macOS/iOS:</b> Mach-O; etc. <b>Linux:</b> ELF; etc. <b>Android:</b> DEX Format; etc. <b>Other:</b> JAR Format; COFF (Common Object File Format); Raw Binary; etc.
<b>Host Architecture</b>	x86 (32, 64)
<b>Target Architecture</b>	<b>IDA Pro Disassembler:</b> x86 (16, 32, 64); ARM (32, 64); PPC (32; 64); MIPS (32, 64); SPARC (32, 64); PIC (12, 16, 17, 18, 24); Java bytecode; DEX bytecode; etc. <b>Hex-Rays Decompiler:</b> x86 (32, 64); ARM (32, 64); PPC (32; 64); MIPS (32, 64); etc.
<b>Initial Release</b>	<b>Disassembler Release (Commercial):</b> 1996 <b>Decompiler Release:</b> 2007
<b>License Type</b>	Proprietary
<b>Maintenance</b>	Maintained by Hex-Rays SA

#### Overview

IDA Pro (Interactive DisAssembler) and Hex-Rays are a disassembly framework and decompiler, respectively. IDA Pro boasts a large number of applicable processor targets and offers a variety of additional capabilities including a debugger, a custom scripting language (IDC), and an interface to execute python scripts called IDA Pro Python (IDAPython).

Since the early 2000s IDA Pro has been regarded as the ‘state-of-the-art’ in disassemblers. With the release of their decompiler in 2005 [30], they established themselves as a leader in automated decompilation for more than a decade. While not considered the edge-of-the-art, the fact that IDA Pro is actively adding new capabilities and refining their existing ones justifies a discussion of the tool in this report. This discussion will focus on its new capabilities and future direction, which is the edge-of-the-art as it pertains to IDA Pro. These new additions include the introduction of their Intermediate Representation (IR) language, value-range analysis engine, and support for ARM v8.3 [32].

#### New Features

**Microcode** – In recent years, competing disassemblers like Binary Ninja and Ghidra have been cutting into IDA Pro’s market share. As such, IDA Pro has evolved, most notably by publicizing their IR, Microcode, to compete with Binary Ninja’s multiple ILs. Microcode has been around for most of IDA Pro’s history, but it was never public and was never API accessible. When Microcode was released it came with a C++ API but no support for IDA Pro

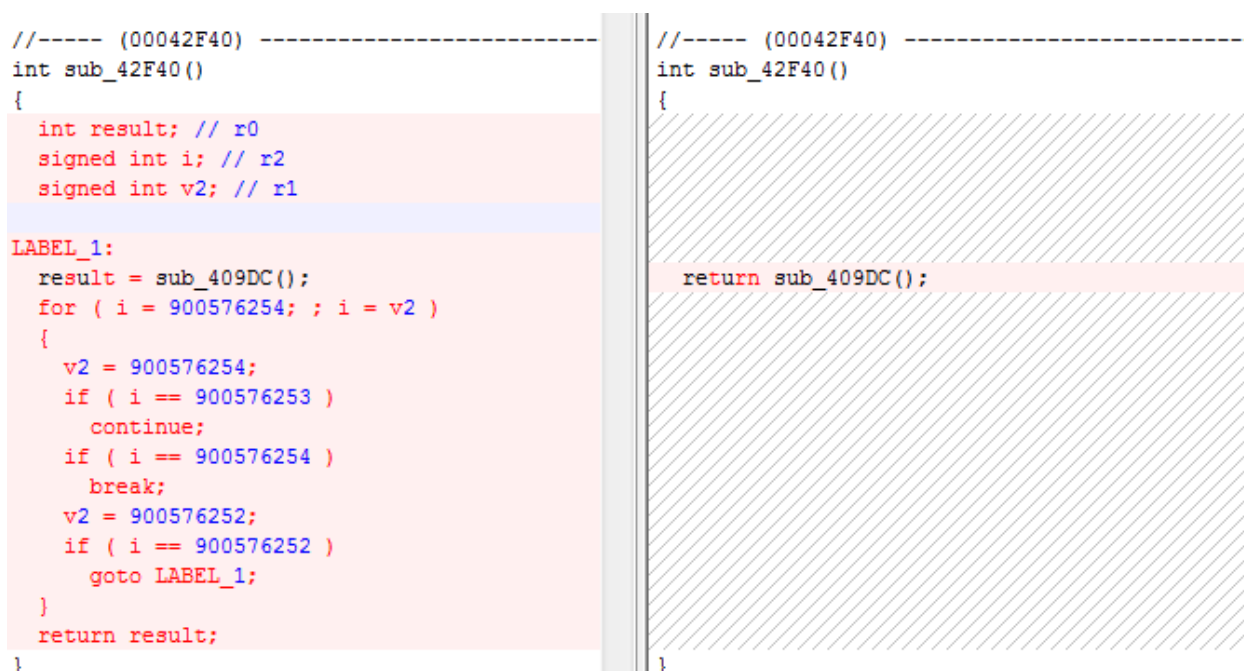


Python. It has various “maturity phases” that it optimizes the code, and the API can be used to select a desired level [31].

**iOS Support** – IDA Pro has also responded to new security capabilities and industry priorities by adding support for Pointer Authentication Code (PAC), as Apple leveraged ARM’s PAC capability in its latest version of iOS. They also added numerous other capabilities to support iOS 12, such as debugger capability to handle new OSX and iOS capabilities, like stack unwinding [32].

**Value-Range Analysis Support** – IDA Pro continues to improve their decompiler. One of the optimizations is integration of value-range analysis, which Hex-Rays discusses in their release announcement of IDA Pro 7.2:

“Now the decompiler has a powerful value-range analysis engine. More than that, it can be used from the Decompiler SDK. The value-range analysis improves the decompilation quality and will also be used to improve the analysis performed by IDA. On the left side is the decompiler output of v7.1, on the right side the decompiler output of v7 [31].”



```
//----- (00042F40) -----  
int sub_42F40()  
{  
    int result; // r0  
    signed int i; // r2  
    signed int v2; // r1  
  
    LABEL_1:  
    result = sub_409DC();  
    for ( i = 900576254; ; i = v2 )  
    {  
        v2 = 900576254;  
        if ( i == 900576253 )  
            continue;  
        if ( i == 900576254 )  
            break;  
        v2 = 900576252;  
        if ( i == 900576252 )  
            goto LABEL_1;  
    }  
    return result;  
}  
  
//----- (00042F40) -----  
int sub_42F40()  
{  
    return sub_409DC();  
}
```

Figure 3-8: An example of IDA Pro value range analysis [31].

## Use Cases and Limitations

IDA Pro is a mature tool that has both benefits and drawbacks. It was one of the only decompilers on the market until 2019, and as the industry standard for more than a decade, a plethora of plugins, techniques, educational materials, etc. have been developed for it. It has also been well tested and comes with paid support. However, an IDA Pro license with the Hex-Rays

decompiler is expensive (a Binary Ninja license is at least an order of magnitude less expensive, and Ghidra is free).

**Compared to Ghidra** – Ghidra offers a free alternative to IDA Pro, with similar disassembly and decompilation capabilities. Its decompilation approach is based on the same academic work as Hex-Rays [36] and decompilation can often return similar results in both tools. The Hex-Rays decompiler does perform significantly better and more reliably than Ghidra's; the latter struggles with switch statements and no-return functions, among other issues. However, Ghidra can decompile anything it can disassemble, and thus supports many more architectures than IDA Pro. This makes it more attractive for decompiling certain nonstandard architectures, like those used in firmware and embedded systems. Additional architectures can also be added with custom processor modules. Ghidra also has a much more usable IR than IDA Pro, and a better scripting capability to interact with the IR [33], [34], [37].

The ability to see and interact with Ghidra's source code enables better, more integrated plugins. However, IDA Pro as a proprietary product, offers more reliable maintenance and customer support. IDA Pro also has more sophisticated support for iOS and macOS, as well as other capabilities that Ghidra lacks [33], [34], [37].

**Compared to Binary Ninja** – Binary Ninja is not open-source software like Ghidra, but a license is significantly less expensive than an IDA Pro license. Regarding decompilation, Binary Ninja cannot compete with Hex-Rays, as Binary Ninja currently has no publicly available decompiler.

However, Binary Ninja was created to be a sophisticated, interactive program analysis tool. Binary Ninja's IR is a set of multiple, composable intermediate languages (ILs) with built-in Single Static Assignment (SSA), and the other capabilities described in the previous section. IDA Pro's Microcode is difficult to use and not designed to be interacted with by users. Binary Ninja's Python also has more capability than IDA Pro's IDAPython. Lastly, its GUI is more modern and easier to use [35].

### 3.3.2 Ghidra

<b>Reference Link</b>	<a href="https://ghidra-sre.org">https://ghidra-sre.org</a>
<b>Target Type</b>	Binary
<b>Host Operating System</b>	Linux; macOS; Windows
<b>Target Operating System</b>	<b>Windows:</b> PE (Portable Executable); etc. <b>macOS/iOS:</b> Mach-O; etc. <b>Linux:</b> ELF; etc. <b>Android:</b> DEX Format; etc. <b>Other:</b> COFF (Common Object File Format); Raw Binaries; etc.
<b>Host Architecture</b>	x86 (32, 64)
<b>Target Architecture</b>	<b>Disassembler and Decompiler:</b> x86 (16, 32, 64); ARM (32, 64); PPC (32, 64); MIPS (32, 64); SPARC (32, 64); PIC (12, 16, 17, 18, 24); Java bytecode; DEX bytecode; etc.
<b>Initial Release</b>	March 2019
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by the National Security Agency (NSA)

#### Overview

For years IDA Pro, and its decompiler, Hex-Rays, was the most widely used tool for automated decompilation. Although other decompilers have been developed, Hex-Rays was the only tool of its kind available to the public. That changed when the National Security Agency (NSA) fully open-sourced their own decompilation tool in March 2019. As the newest tool available to the public, Ghidra could be considered the current edge-of-the-art in decompilation, but it has been in development and use since at least the mid 2000s. In addition to being an open-source decompiler, Ghidra brings with it a fully integrated Intermediate Representation (IR), P-Code, which users can interact with via multiple APIs, native scripting in Java, as well as the ability to use Python via Jython, and the built in ability to have a shared Project on a Ghidra Server [34], [36].

#### Design and Usage

Ghidra is written in Java and run with OpenJDK. It's decompiler works by first lifting binary code to P-Code, and then decompiling from the P-Code. This means that anything Ghidra can disassemble accurately to P-Code can also be decompiled.

To use Ghidra, one first creates a project and then imports a file to be analyzed. Its default UI, CodeBrowser, is similar to that of IDA Pro. An example is shown in Figure 3-9 for a sample x86 ELF binary. It shows windows for "Program Trees", "Symbol Tree", "Data Type Manager", a disassembly "Listing" window, a "Decompile" window, and a scripting console. Ghidra will also generate a Function Graph for the loaded binary, as shown in Figure 3-10 [34], [36].

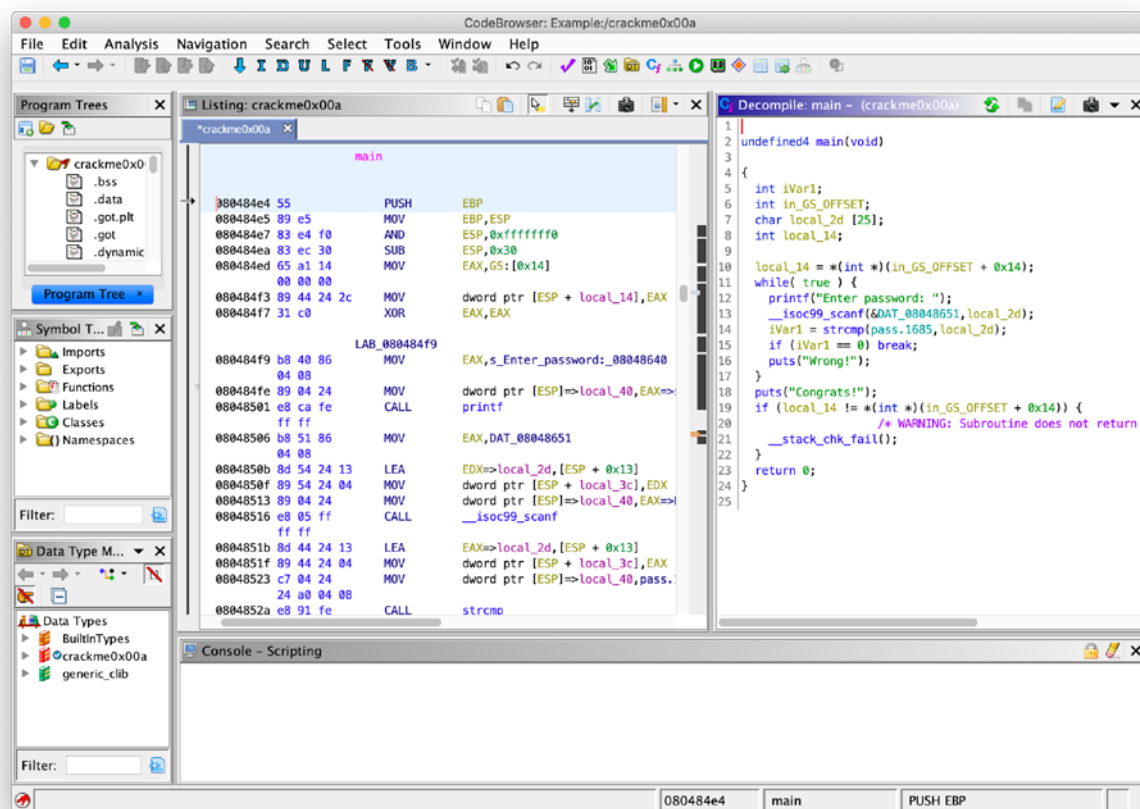


Figure 3-9 Ghidra CodeBrowser

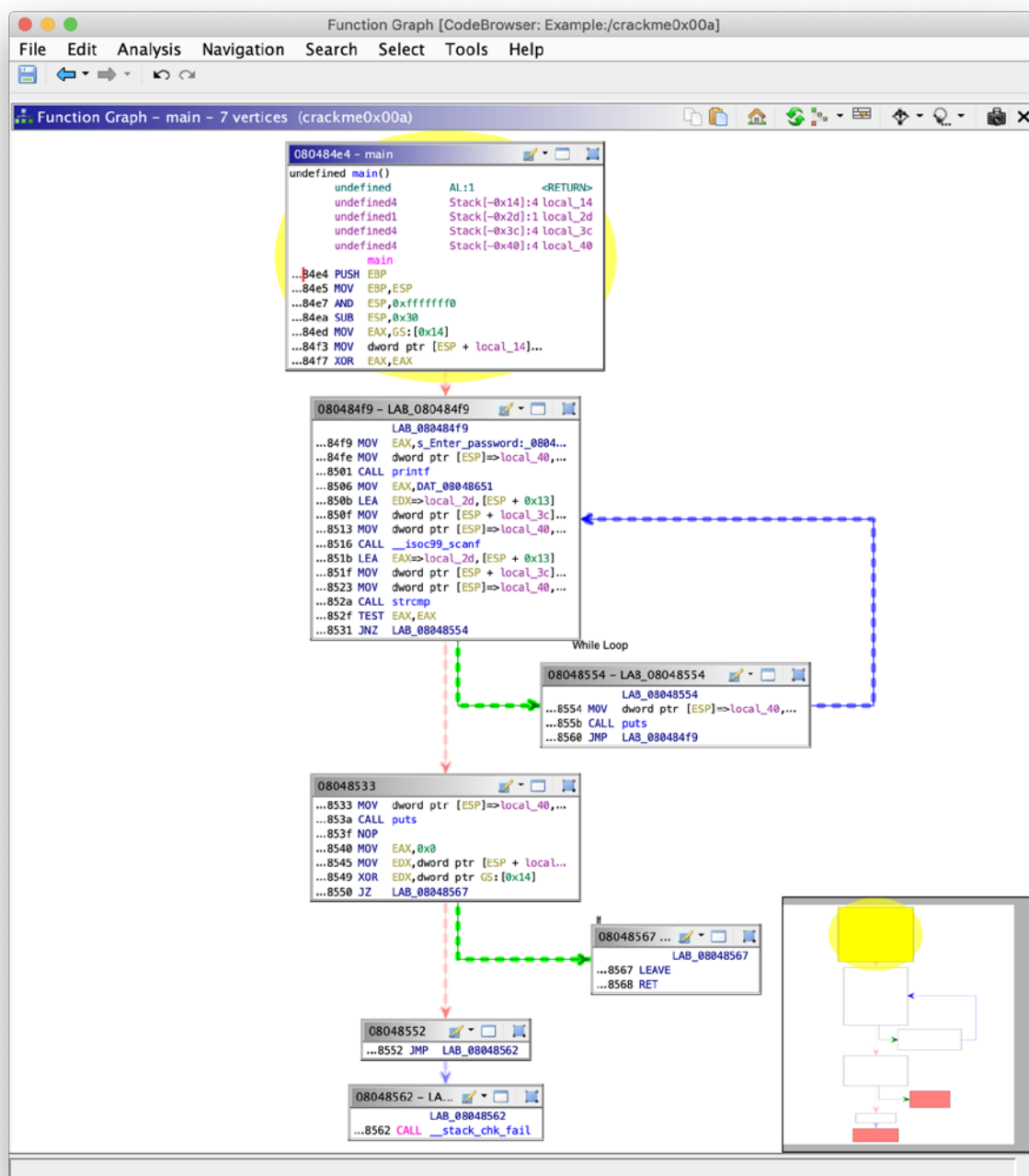


Figure 3-10: Ghidra Function Graph

**Scripting** – Scripting in Ghidra is primarily in Java but can also be done in Python via Jython. It also comes with over 200 preinstalled scripts that not only provide additional capability but are helpful examples for developers. Ghidra has two main APIs, the Ghidra Program API which is heavily object oriented and its simplified counterpart, the FlatProgramAPI. Ghidra also provides integration with Eclipse, which enables script development in a full IDE. Lastly, Ghidra offers a Python interpreter window, which also uses Jython [34], [36].

**P-Code** – One of the most interesting capabilities of Ghidra is its P-Code IR. P-Code operates on *varnodes*, which represent a register or a memory location that are comprised of the address space (RAM), the offset in that space and the size [36]. Instructions are abstracted to P-Code's operations, which is a small instruction set [37]. P-Code is not in SSA form until decompilation [34]. Ghidra's processor specification language, SLEIGH, will lift a variety of architectures to P-Code.

**Shared Projects** – An issue that has long plagued IDA Pro users is the lack of support for collaboration. Many attempts to create an IDA Pro plugin for collaboration were at best partially successful due to the lack of inherent support for the capability. Ghidra enables multiple users to work on the same project at once by creating a *Shared Project* with a *Ghidra Server* [34].

## Use Cases and Limitations

**Compared to IDA Pro** – Ghidra offer a free alternative to IDA Pro, with similar disassembly and decompilation capability. Its decompilation approach is based on the same academic work as Hex-Rays [36] and decompilation can often return similar results in both tools. The Hex-Rays decompiler does perform significantly better and more reliably than Ghidra's decompiler; the latter struggles with switch statements and no-return functions, among other issues. However, Ghidra can decompile anything it can disassemble, and thus supports many more architectures than the ones supported by IDA Pro. This makes it more attractive for decompiling nonstandard architectures, like those used in firmware and embedded systems. Additional architectures can also be added with custom processor modules. Ghidra also has a much more usable IR than IDA Pro, and a better scripting capability to interact with the IR [33], [34], [37].

The ability to see and interact with Ghidra's source code enables better, more integrated plugins. However, IDA Pro as a proprietary product, offers more reliable maintenance and customer support. IDA Pro also has more sophisticated support for iOS and macOS, as well as other capabilities that Ghidra lacks [33], [34], [37].

**Compared to Binary Ninja** – Ghidra offers an integrated decompiler, which Binary Ninja currently lacks. The ability to see and interact with Ghidra's source code enables better, more integrated plugins [35], [36]. However, as a disassembler, Binary Ninja offers significant benefits that Ghidra does not. Ghidra's P-Code is not designed to be human readable and is not inherently SSA. It also lacks the multiple levels of ILs that Binary Ninja offers.

Binary Ninja was built on top of years of program analysis work that Ghidra, due to its age, was not. This results in Binary Ninja being far more suited for automated program analysis, because that capability is scriptable. Yet, as a decompiler with a scriptable IR, Ghidra does offer program analysis capabilities that Binary Nina does not [35].

### 3.3.3 Binary Ninja

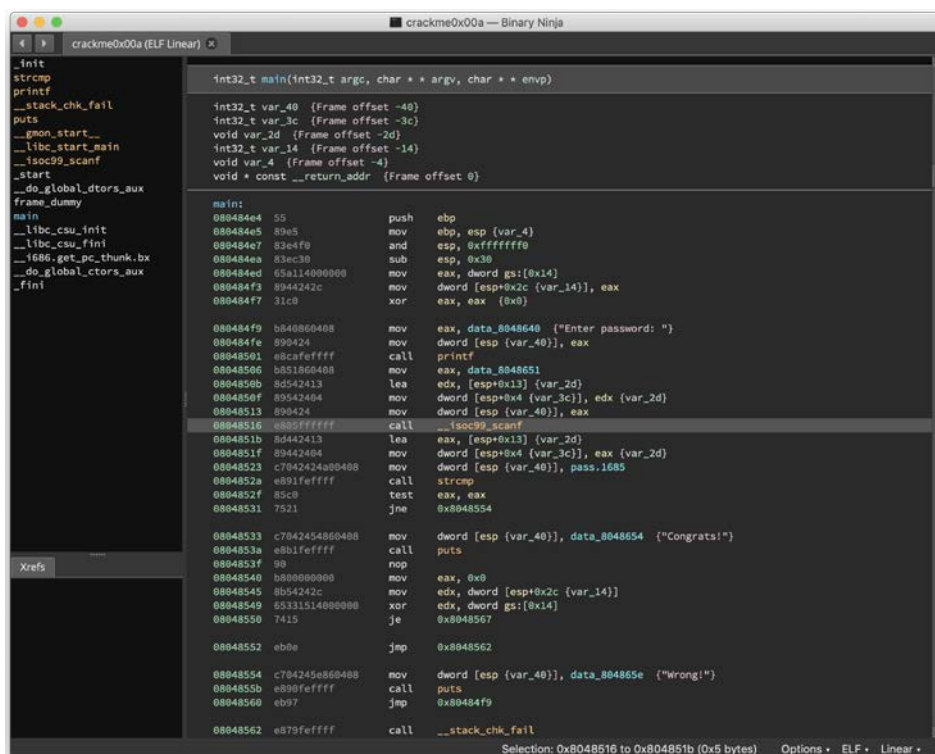
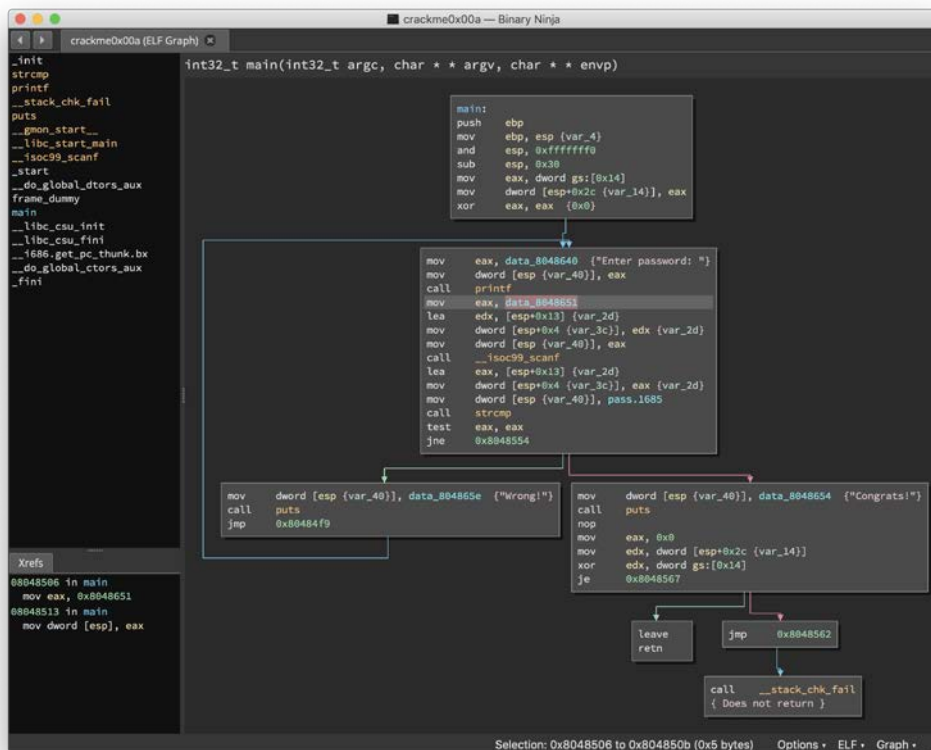
Reference Link	https://binary.ninja																																																		
Target Type	Binary																																																		
Host Operating System	Linux; macOS; Windows																																																		
Target Operating System	<b>Windows:</b> PE (Portable Executable); etc. <b>macOS/iOS:</b> Mach-O; etc. <b>Linux:</b> ELF; etc. <b>Other:</b> COFF (Common Object File Format); Raw Binary; etc.																																																		
Host Architecture	x86 (32, 64)																																																		
Target Architecture	<table><tr><th>Architecture</th><th>Disassembly</th><th>Lifting</th><th>Assembling</th><th>Compiling</th></tr><tr><td>x86 32-bit</td><td>Y</td><td>Partial</td><td>Y</td><td>Y</td></tr><tr><td>x86 64-bit</td><td>Y</td><td>Partial</td><td>Y</td><td>Y</td></tr><tr><td>ARMv7</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr><tr><td>Thumb2</td><td>Y</td><td>Y</td><td>Y</td><td>N</td></tr><tr><td>ARMv8</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr><tr><td>PowerPC</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr><tr><td>MIPS</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr><tr><td>6502</td><td>Y</td><td>Y</td><td>-</td><td>-</td></tr><tr><td>Many others!</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table> <p>Supported Architectures [35]</p>	Architecture	Disassembly	Lifting	Assembling	Compiling	x86 32-bit	Y	Partial	Y	Y	x86 64-bit	Y	Partial	Y	Y	ARMv7	Y	Y	Y	Y	Thumb2	Y	Y	Y	N	ARMv8	Y	Y	Y	Y	PowerPC	Y	Y	Y	Y	MIPS	Y	Y	Y	Y	6502	Y	Y	-	-	Many others!	-	-	-	-
Architecture	Disassembly	Lifting	Assembling	Compiling																																															
x86 32-bit	Y	Partial	Y	Y																																															
x86 64-bit	Y	Partial	Y	Y																																															
ARMv7	Y	Y	Y	Y																																															
Thumb2	Y	Y	Y	N																																															
ARMv8	Y	Y	Y	Y																																															
PowerPC	Y	Y	Y	Y																																															
MIPS	Y	Y	Y	Y																																															
6502	Y	Y	-	-																																															
Many others!	-	-	-	-																																															
Initial Release	2016																																																		
License Type	Proprietary																																																		
Maintenance	Maintained by Vector 35																																																		

#### Overview

Binary Ninja emerged in 2016 as one of the more formidable competitors to IDA Pro in the disassembler market. Although lacking a decompiler, Binary Ninja carved a niche as a modern reverse engineering platform built for modern program analysis techniques [35].

#### Design and Usage

Binary Ninja is built for contemporary reverse engineering and ease of use, while also incorporating the capability for complex program analysis techniques. IDA Pro and Ghidra were both written over a decade ago and this is reflected in the look of their interfaces. By contrast Binary Ninja is modern and easy to use. After choosing a binary, a user is presented its default Graph View, an example of which is shown in Figure 3-11. Similar to other disassemblers, Binary Ninja also offers Hex and Linear Views, the latter shown in Figure 3-12 [35].





**Intermediate Languages** – One of Binary Ninja’s most useful capabilities are its intermediate languages (ILs). Binary Ninja’s IR consists of an entire family of ILs: low, medium and soon a high-level IL (shown in Figure 3-13). The user can easily switch between disassembly and low-level IL (LLIL) in graph view with a hotkey. In contrast with IDA Pro’s Microcode, and to a lesser extent Ghidra’s P-Code, Binary Ninja’s LLIL is meant to be human readable. Binary Ninja’s ILs are also more sophisticated, using Single Static Assignment, deferred flag calculation, the ability to transform assembly instructions to generic ones that enable for equations with multiple operations, and showing comparisons instead of flag setting. Much of this happens in Binary Ninja’s Medium Level IL (MLIL) [35], [38], [39].

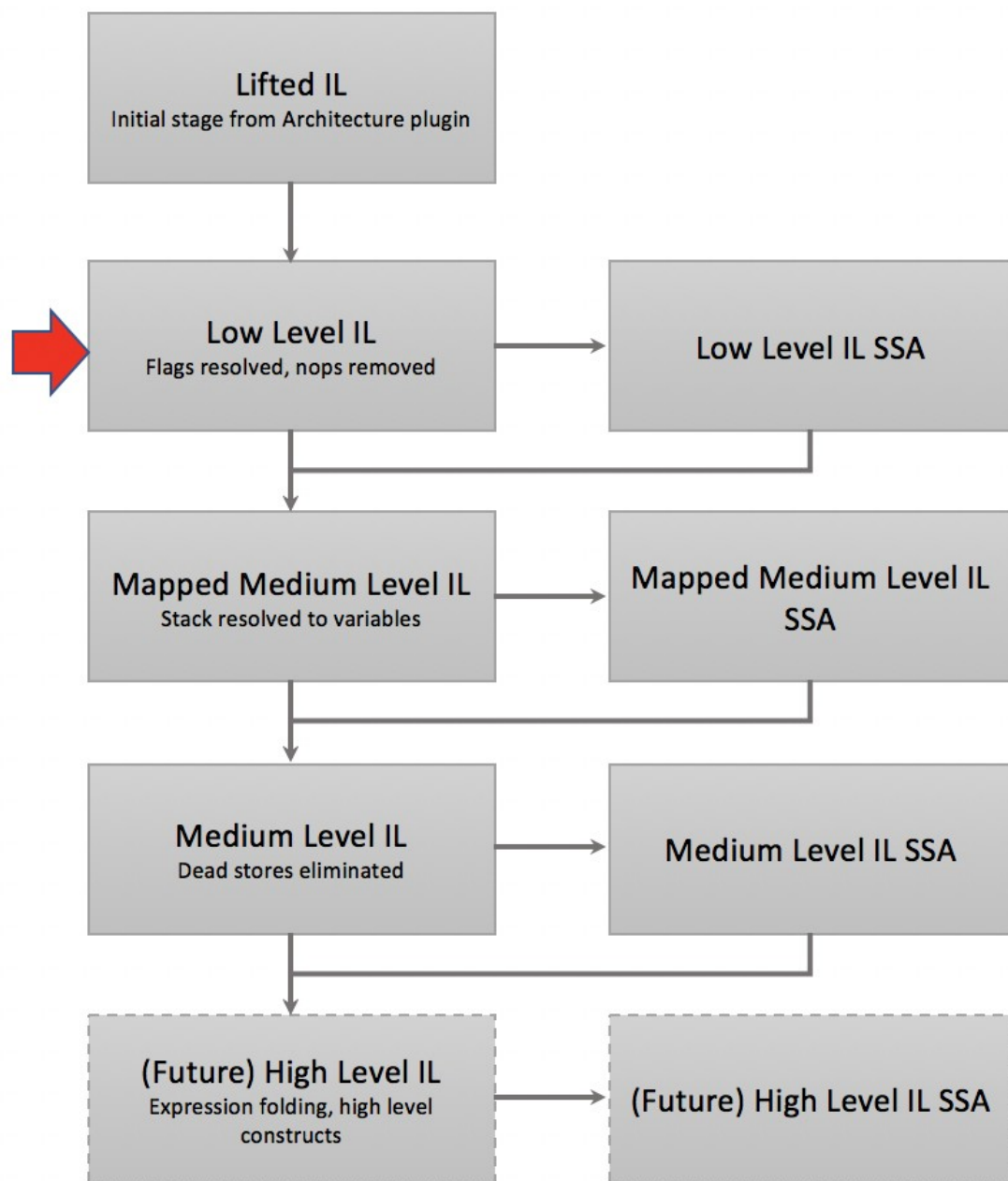


Figure 3-13: Binary Ninja’s various intermediate languages [38]

**Python Scripting** – Binary Ninja’s Python scripting tool and API were integrated from the start, enabling the user access to an array of capabilities, either through writing scripts or using the IPython command-line. Like Ghidra, Binary Ninja’s Python scripting tool interfaces with its ILs. However, in Binary Ninja’s case, it offers far more capability, especially considering that it can interact with both the LLIL and the MLIL [35], [38], [39].

## **Use Cases and Limitations**

Binary Ninja is designed for modern day program analysis and could be considered the current “cutting edge” in program analysis disassembly frameworks. The biggest limitation that Binary Ninja has is its lack of a decompiler. Users must use Binary Ninja in conjunction with IDA Pro or Ghidra if they also need automatic decompilation.

Binary Ninja does outstrip both IDA Pro and Ghidra in the area of program analysis. It has a far more sophisticated set of ILs and better integration with its scripting tool. Its scripting tool is also often considered superior.

**Comparison to IDA Pro** – Binary Ninja is not open-source software like Ghidra, but a license costs several orders of magnitude less than an IDA Pro license. In the area of decompilation Binary Ninja cannot compete with Hex-Rays, as Binary Ninja does not offer a publicly available decompiler.

However, Binary Ninja was created as a sophisticated program analysis tool. IDA Pro’s Microcode is difficult to use and not designed to be interacted with in the way as Binary Ninja’s multiple ILs. That Binary Ninja has multiple ILs is also a major benefit. The ILs themselves also make Binary Ninja an attractive alternative to IDA Pro, with built-in SSA, and the other capabilities described in the previous section. Binary Ninja’s Python interface can offer more capabilities than IDA Pro’s IDAPython. Lastly, its GUI is more modern and easier to use [35].

**Comparison to Ghidra** – Ghidra offers an integrated decompiler, which Binary Ninja currently lacks. The ability to see and interact with Ghidra’s source code enables better, more integrated plugins [35], [36].

However, as a disassembler, Binary Ninja offers significant benefits that Ghidra does not. Ghidra’s P-Code is not designed to be human readable and is not inherently SSA. It also lacks the multiple levels of ILs that Binary Ninja offers.

Binary Ninja was built on top of years of program analysis work that Ghidra, due to its age, was not. This means Binary Ninja is much better suited for automated program analysis, because that capability is scriptable. Yet, as a decompiler with a scriptable IR, Ghidra does offer program analysis capabilities that Binary Ninja does not [35].

## 3.4 Static Instrumentation

Instrumentation facilitates debugging, tracing, and profiling, which in turn enables higher level analysis like fuzzing, symbolic execution, etc. This section discusses several tools that have made significant gains in their specific niche and have more recently come into wider use.

### 3.4.1 Multiverse

Reference Link	<a href="https://github.com/utds3lab/multiverse">https://github.com/utds3lab/multiverse</a>
Target Type	Binary
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	February 2018
License Type	Open-Source
Maintenance	Last commit February 2018

#### Overview

Multiverse is a static binary rewriter based on the paper *Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics* by Erick Bauman, Zhiqiang Lin and Kevin W. Hamlen of the University of Texas at Dallas. It operates either as a standalone executable with limited capability, or a more robust Python library [40], [41].

Many tools exist to dynamically instrument binaries in a variety of ways, but few do so statically. Static rewriters are one class of static instrumentation tools that serve a variety of code transformation purposes, including adding (or, in a malicious case, removing) security capabilities like stack protectors and Control Flow Integrity (CFI). Current static rewriters often require binaries with debugging symbols or relocation entries that are disassembled correctly, among other constraints. Multiverse, however, is able to perform the same capability with far fewer limitations on its binary targets [40], [41].

Multiverse does so by disassembling the binary into a superset disassembly which contains “all legal instructions” [41, p. 1]. It also uses an instruction rewriter to relocate instructions and modify control flow [40], [41].

The resulting tool can rewrite binaries with or without instrumentation. When run from the command line as a Python file, it can only rewrite binaries without instrumentation. When used via its Python library, it can rewrite binaries with instrumentation [40].

#### Design

The authors of Superset Disassembly offer an overview of the two techniques which comprise their tool. They describe their superset disassembly technique as one which “does not make any assumptions on where a legal instruction should start and instead disassembles and

reassembles each offset, achieving complete recovery of original instructions.” They then describe their static instruction reassembling technique as one which, “translates all indirect control flow transfer instructions (including those in the library) and redirects their target addresses to correct ones, achieving the soundness of original program execution.” Figure 3-14 below provides an overview of Multiverse that the authors included in their paper [41].

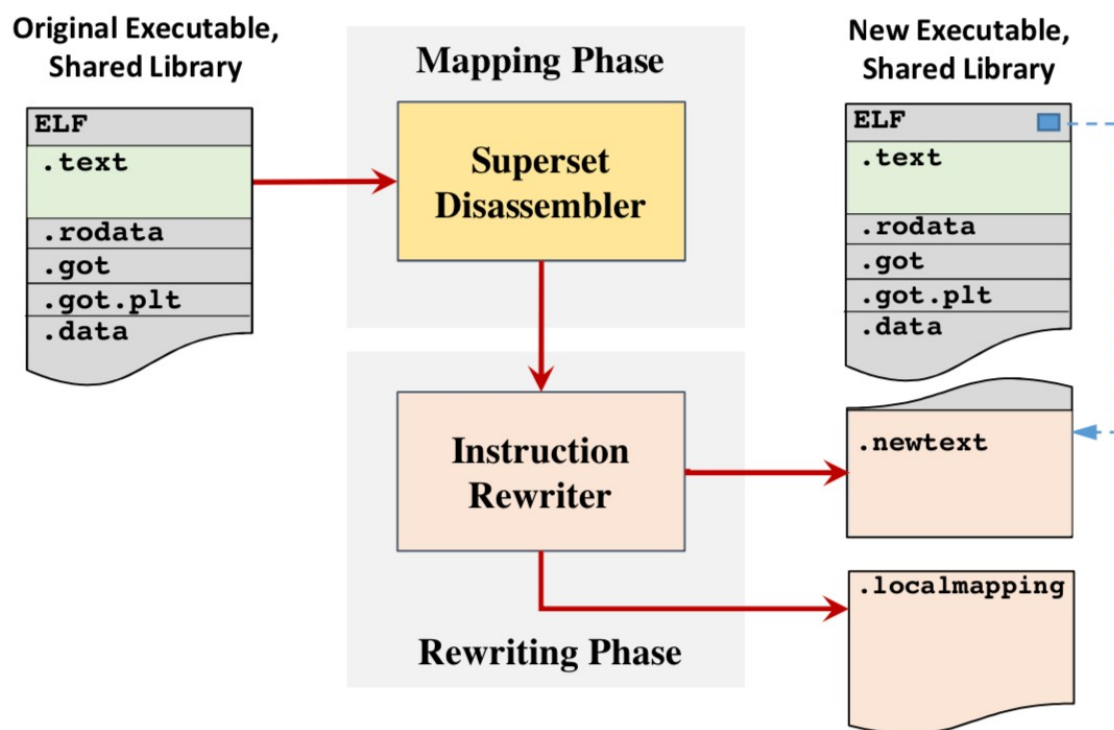


Figure 3-14: An overview of multiverse [41, Fig. 2]

This is all done without heuristics, unlike the reassembling tool DDisasm. However, this does function like a reassemble, albeit disassembling into a specific superset of assembly designed for the instruction rewriter. [41] Multiverse is written in Python and built on top of the Capstone disassembler [40].

## Usage

Multiverse can be run either directly from the command line as a Python file or used as a Python library. The former only rewrites binaries without instrumentation. Multiverse offers this option to “make sure that everything is installed correctly or to debug changes to the rewriter [40].” It works by executing a command in the format:

```
./multiverse.py [options] <filename>
```

Several options are available including `--so` (rewrites a shared object), `--execonly` (rewrites a main binary using the original, unmodified libraries), `--nopic` (rewrites binary without support for position independent code) and `--arch` (specify the binary's architecture).

To fully utilize the instrumentation capabilities of Multiverse, the user must employ the tool's Python library. The user creates a Rewriter object whose constructor takes three boolean arguments corresponding to the first three options listed for the command line usage defined above, in that same order [40]. Below in Figure 3-15 is an example of a simple Python file which uses Multiverse to add a NOP after every instruction in an x64 binary.

```
1  #!/usr/bin/python
2
3  import sys
4  from elftools.elf.elffile import ELFFile
5  from multiverse import Rewriter
6  from x64_assembler import _asm
7
8  def count_instruction(inst):
9      template = ''
10     nop
11     ''
12     inc = template
13     return _asm( inc )
14
15  if __name__ == '__main__':
16     if len(sys.argv) == 2:
17         f = open(sys.argv[1])
18         e = ELFFile(f)
19         entry_point = e.header.e_entry
20         f.close()
21         #write_so = False, exec_only = True, no_pic = True
22         rewriter = Rewriter(False, True, False)
23         rewriter.set_before_inst_callback(count_instruction)
24         rewriter.rewrite(sys.argv[1], 'x86-64')
25     else:
26         print "Error: must pass executable filename.\nCorrect usage: %s <filename>"%sys.argv[0]
```

*Figure 3-15: An example python file which uses Multiverse to add a NOP after every instruction in an x64 binary (multiverse/addnop.py) [40]*

## Use Cases and Limitations

As discussed in the overview, Multiverse and its techniques have a plethora of potential uses, many of which previously required a dynamic instrumentation tool. For example, the paper demonstrates using Multiverse to insert a shadow stack (a separate memory region that preserves the state of the stack when a function is called to check its integrity after the function returns).

Another potential use is instrumenting prebuilt binaries with AFL’s instrumentation, enabling the user to run AFL directly on the binary without QEMU mode or some other emulation.

However, the Multiverse implementation is a prototype based on their paper and is not currently being maintained. It only supports x86 ELF binaries and shared library object files. There are several ways it can be optimized, and much of the potential capability has yet to be implemented.

### 3.4.2 DDisasm

Reference Link	<a href="https://github.com/GrammaTech/ddisasm">https://github.com/GrammaTech/ddisasm</a>
Target Type	Binary
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	April 16, 2018
License Type	Open-Source
Maintenance	Maintained by GrammaTech

#### Overview

Disassembly is one of the first steps reverse engineers take when analyzing a binary. Surprisingly, there are few tools concerned with automatically reassembling disassembled code. In a 2015 paper, *Reassembleable Dissassembling*, the authors Shuai Wnag, Pei Wang and Dinghao Wu of Pennsylvania State University claimed 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 resulted 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. [42, p. 1]” The paper presented a tool that could disassemble a binary over a set of rules which made the resulting disassembly relocateable, which they claim is the “key” to reassembling [42].

Since 2015, this technique has been improved, notably by the creators of angr who built a reassembling tool called Ramblr [43]. More recently, the tool DDisasm was introduced in a June 2019 paper, *Datalog Dissassembly*, by Antonio Flores-Montoya and Eric Shulte of GrammaTech, Inc. [44]. Their technique disassembles binaries with accurate symbolic information which, as they write in their abstract, enables “cross-referenc[ing] for analysis and... [the] adjustment of code and data pointers to accommodate rewriting. Our technique capabilities multiple static analyses and heuristics in a combined Datalog implementation [44, p. 1].” They claim that this technique significantly out-performs Ramblr, which they term “the current state-of-the-art [44, p. 1].”

Alongside their paper they provide an open-source implementation, written in Datalog, which disassembles binaries in a way that enables them to be reassembled by common compilers, specifically gcc and Clang [45].

## Design

Using analysis of common compiler and assembler methods and static program analysis, the authors were able to reason about instruction boundaries, symbolization information and function boundary identification, along with other aspects of binary code. Instruction boundary identification addresses the issue that binaries lack clear beginnings and ends to instruction, and this can be difficult to discern, especially for architectures with variable length instructions.

Symbolization information, on the other hand, concerns discerning between literals and references. The authors provide an example that, “[i]f we modify a binary, for example by moving a block of code, all of the references that point to that block, and to all of the subsequently shifted blocks, have to be updated. On the other hand, literals, even if they coincide with the address of a block, have to remain unchanged [44, p. 1].”

They implement their tool in Datalog and provide it as an open-source repository alongside the paper.

## Usage

The tool is invoked from the command line as follows:

```
ddisasm my_binary --asm my_output_file.s
```

The tool can take a number of flags including `--asm` which specifies the output file, `--debug` which prints debug information along with information, `--keep-` functions which prints skipped functions like `_start`, and `--sect arg (=,plt,got,fini,init,plt,text,)` which enables the user to specify sections to decode. The tool works on stripped binaries as well [45].

Reassembling the code simply requires running `gcc` on the disassembly file:

```
gcc my_output_file.s
```

## Use Cases

The authors of DDisasm argue that their tool is the better than angr’s Ramblr, the current state-of-the-art. One of the examples they provide for this is evaluating the tools on binaries from the Cyber Grand Challenge (CGC).

“We disassemble the binaries with DDisasm and Ramblr, we reassemble the resulting assembly code `gcc` and we run the original tests on the new binaries. We also perform the experiment with stripped versions of the binaries. In that case, we strip the binaries before running the disassemblers.

“DDisasm is able to produce reassembleable assembly code for all the binaries and only 3 in the CGC benchmark fail some of the tests. Note that the number of binaries that fail some tests is smaller [than] the number of broken binaries according to our previous experiment...

“This is because the test suites of the binaries are not exhaustive. The results also show that DDisasm does not depend on the information present in symbol tables and can perform equally well with stripped binaries.

“On the other hand, there are many binaries that fail to reassemble with Ramblr and the results of the tests are worse than those of the symbolization information. We have found and reported several bugs to the Ramblr authors which they have promptly fixed but there might be others that cause additional failures. Ramblr fails to produce reassembleable assembly for the stripped versions of most programs in Coreutils and the real-world benchmarks. Many of the failures are because Ramblr does not find the main function or generates assembly with undefined labels. We believe that these are not fundamental issues and should be easy to fix in most cases [44, p. 12].”

## Limitations

From the tool evaluation example quoted above, it is clear that the reassembly is not always accurate, and can at times cause the new binary to have inaccurate information or fail tests. Their tool resulted in broken binaries 1.66% of the time on the real-world programs they chose to evaluate [44, p. 12]. By comparison, Ramblr resulted in likely broken binaries 35.5% of the time on the same tests [44, p. 12]. This does not account for the binaries that may have been broken but were not identified through their tests.

### 3.4.3 LIEF

<b>Reference Link</b>	<b><a href="https://lief.quarkslab.com">https://lief.quarkslab.com</a></b>
<b>Target Type</b>	Binary
<b>Host Operating System</b>	Linux; macOS; Windows
<b>Target Operating System</b>	Linux; macOS; Windows; Android
<b>Host/Target Architecture</b>	x86 (32, 64)
<b>Initial Release</b>	April 4, 2019
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Quarkslab

## Overview

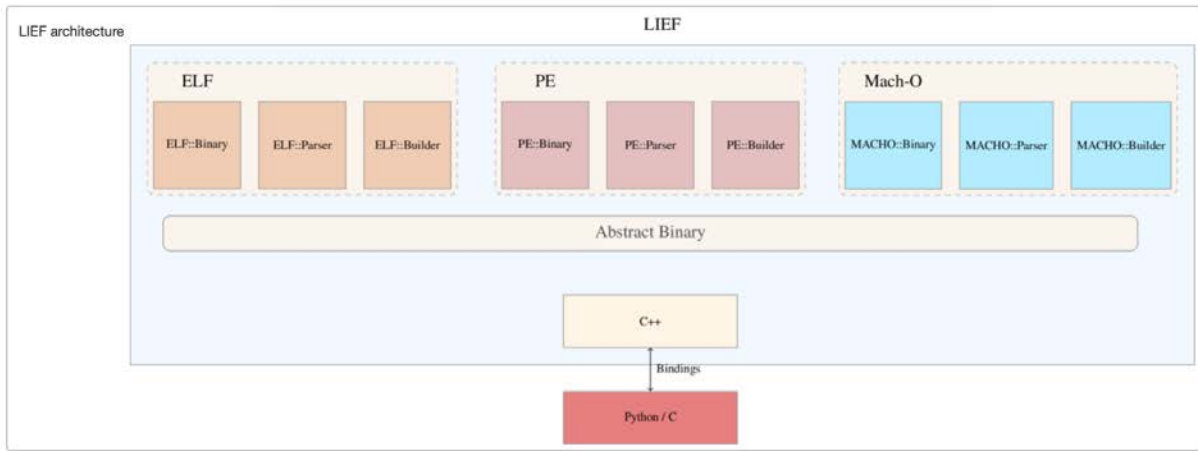
LIEF, or “Library to Instrument Executable Formats,” is an open-source tool used to parse, modify, and abstract executables. It offers a Python, C++, or C API with which to parse, edit and analyze portions of executables. By abstracting common characteristics of executables, the user can compare across different types [46].

One of the most attractive capabilities of LIEF is its robust documentation and ease of use. It offers a clean, simple way for users to work with executable formats, which is in demand in reverse engineering and program analysis communities. It has a sophisticated, interactive alternative to bare bones tools like objdump. It runs on all three major desktop operating systems (not just Linux), and can handle their executable formats as well [46].



## Design and Usage

Figure 3-16 shows LIEF's architecture.



*Figure 3-16: Architecture of LIEF [47]*

The documentation offers the following example of how LIEF would parse an ELF executable using the Python library. The `ELF::Parser` class contains the function `parse()` which takes a file path to an executable and parses it. It does so by splitting the ELF into its various parts so segments will each become `ELF::Segment` objects and sections become `ELF::Section` objects. These `ELF::Segment` objects then become part of an `ELF::Binary` object, which is returned by `parse()` [48].

Example usage below:

```
binary = lief.parse("path/to/executable")
```

This 'binary' object contains 'binary.segments' and 'binary.sections' and the user can retrieve the objects they contain and modify their contents. The documentation offers this example of doing so to the `.text` section:

```
text = binary.get_section(".text")
```

```
text.content = bytes([0x33] * text.size)
```

This object is modifiable: its type, size, content, etc. can be changed. It can then be rebuilt with the `ELF::Builder` class by calling `binary.build()`. A diagram of this process applied to the executable `/bin/ls` is provided by the documentation and shown in Figure 3-17 [48].

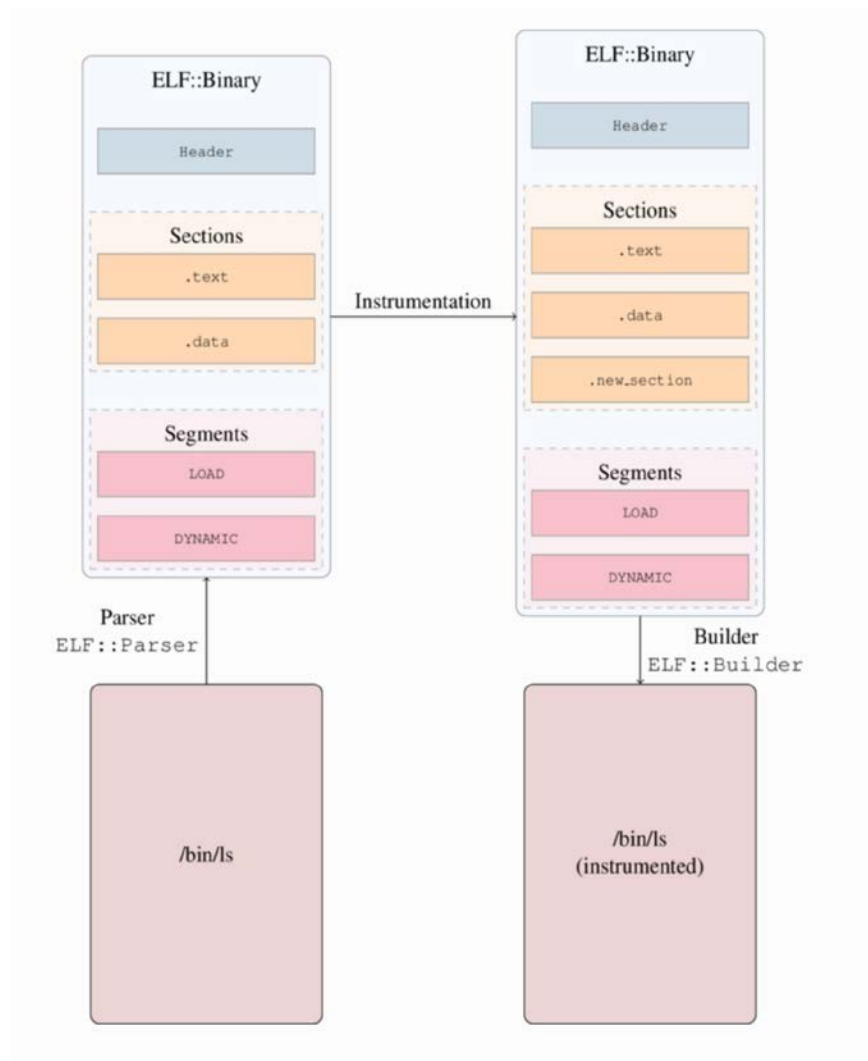


Figure 3-17: Applying LIEF to `/bin/ls` [48]

LIEF enables the same to be done with Windows PEs, but goes further by enabling the user to create a PE from scratch. Additional capabilities include hooking into ELF and PEs, manipulating the PLT and GOT, and converting an ELF into a library. LIEF also has an entire set of capabilities designed for Android formats. It offers Python, C++, and C APIs and integration with Frida [46].

## Use Cases and Limitations

LIEF offers a number of use cases in reverse engineering and program analysis, anywhere from simply analyzing a binary to instrumenting it. It can be used to dynamically reverse engineer a binary, hook a library and manipulate symbolic data. While LIEF is a well maintained, well documented, and organized tool, it also does not have a significant breadth or depth of capabilities. None of the operations described here are novel. It cannot statically rewrite binaries like DDisasm or instrument them like Multiverse. LIEF, however, makes the operations it can perform easy and painless, which is its appeal.

## 3.5 Dynamic Analysis and Exploitation

Dynamic Instrumentation enables the analysis of binaries during execution. Frida and rr significantly provide a way to control and modify execution. Tools like Lighthouse complement Frida, and state-of-the-art tools like DynamoRIO, by enabling the user to better interpret their results. Exploitation frameworks like FUZE automate certain dynamic tasks necessary to exploitation.

### 3.5.1 Lighthouse

Reference Link	<a href="https://github.com/gaasedelen/lighthouse">https://github.com/gaasedelen/lighthouse</a>
Target Type	<b>Plugin:</b> IDA Pro; Binary Ninja <b>Coverage Data:</b> DynamoRIO; Intel Pin; Frida
Host Operating System	<b>IDA Pro Plugin:</b> Windows; Linux; macOS <b>Binary Ninja Plugin:</b> Windows; Linux
Host Architecture	x86 (32, 64)
Initial Release	March 8, 2017
License Type	Open-Source
Maintenance	Maintained by Ret2 Systems

#### Overview

Lighthouse is a code coverage plugin, primarily for IDA Pro but with experimental support in Binary Ninja. It colorfully displays coverage data generated by DyanamoRIO, Intel Pin, or Frida in regular IDA Pro views, and presents users with a widget for viewing and interacting with the coverage data. Furthermore, it helps examine relationships between multiple coverage sets as coverage compositions. Lighthouse includes a syntax for creating compositions and a UI for evaluating these compositions in real-time, called *Hot Shell* [10], [50], [51].

#### Design and Usage

Coverage data is loaded as a .log file generated by one of three tools, DynamoRIO, IntelPin, and experimentally Frida. Lighthouse gives the user *Coverage Painting* and *Coverage Overview* views, examples of which are shown in Figure 3-18 and Figure 3-19, respectively. Code Painting enables the user to see “the active coverage data [painted] across the three major IDA Pro views as applicable, whereas Coverage Overview is a “dockable widget that provides a function level view of the active coverage data for the database [10].”

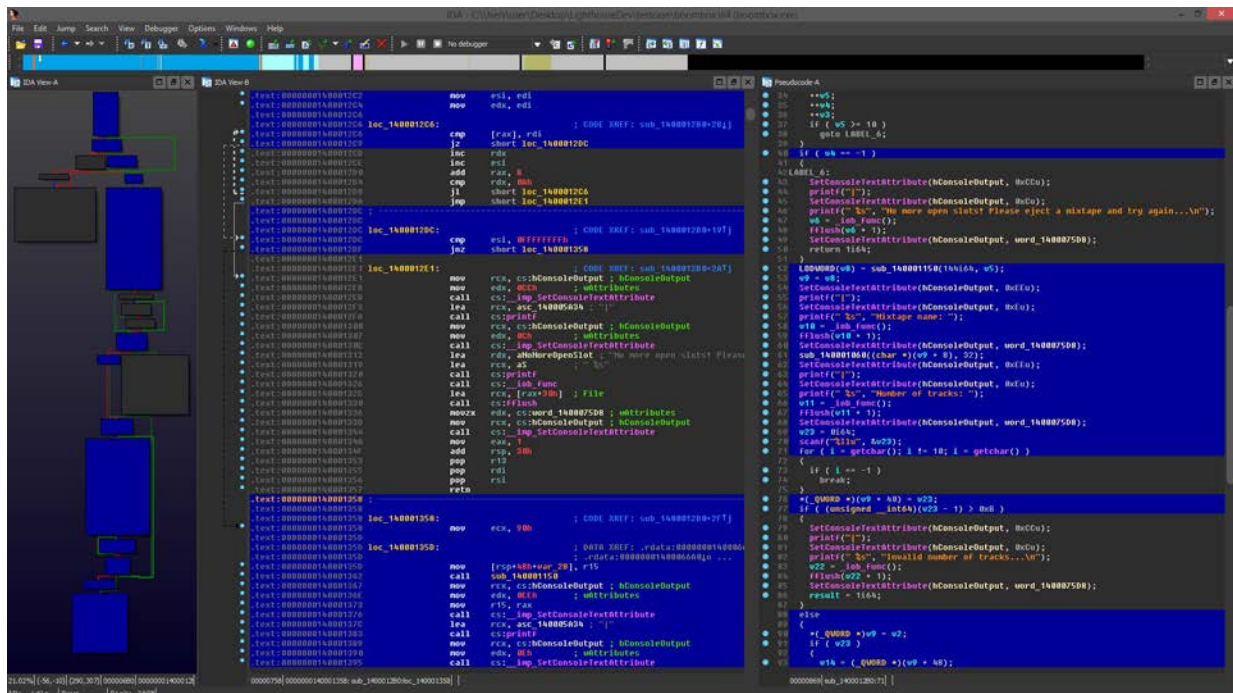


Figure 3-18: Lighthouse Coverage Painting view [10, README]

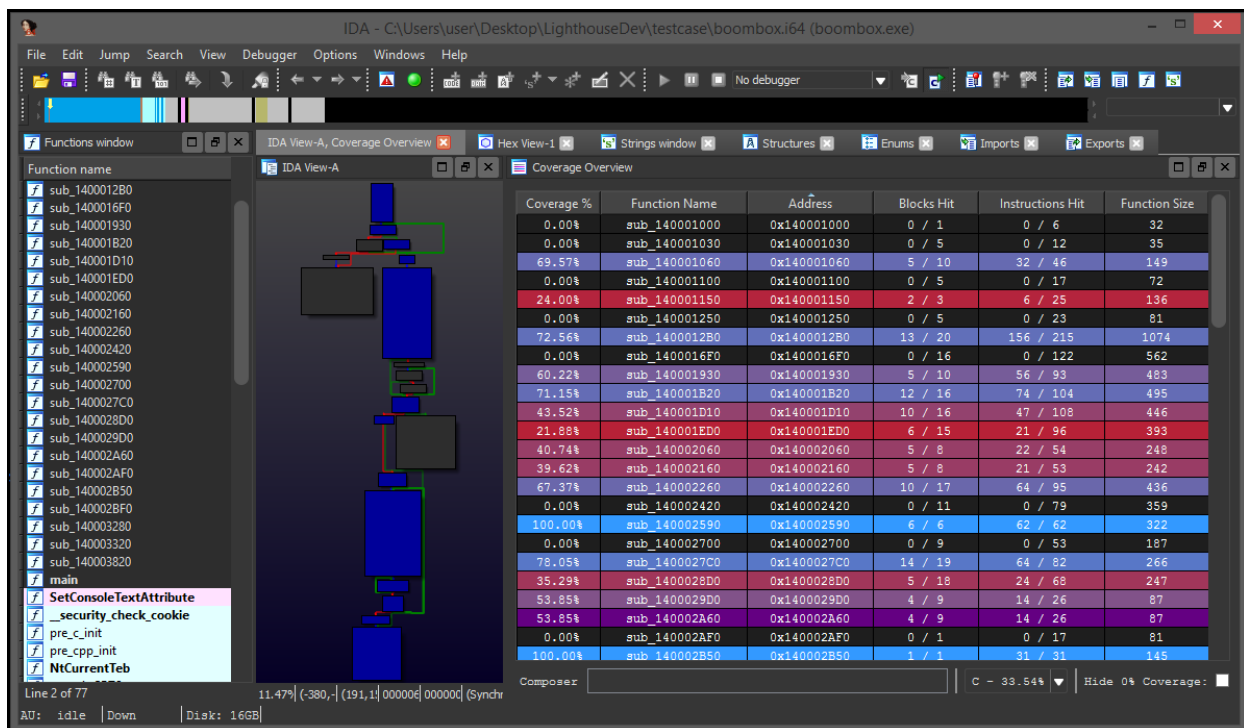


Figure 3-19: Lighthouse Coverage Overview [10, README]

Coverage Overview displays a number of useful metrics, such as cyclomatic complexity, and helps the user manipulate or interact with the data in the table [10]. The user may load multiple coverage sets and create coverage compositions via the composition syntax. Coverage sets

correspond to a symbol and are operated on by logical operators. For example, if A, B and C are each coverage sets,  $(A \ \& \ B) \mid C$  is a valid expression. This can be entered into the input field of the Hot Shell window, a UI that enables the user to explore compositions, including to search it and jump to various addresses [10].

### Use Cases and Limitations

The primary purpose of this tool is to easily examine code coverage. The fact that it integrates with two of the major disassemblers, reads data generated by well-known tools, and requires no extra formatting makes this tool very user friendly. It is useful for differential analysis of multiple program executions and examining and comparing fuzzing runs. It is billed as ‘only a prototype and code resource for the community’ and has had 8 releases, the latest in October 2018 [10].

### 3.5.2 Frida

<b>Reference Link</b>	<a href="https://www.frida.re/">https://www.frida.re/</a>
<b>Target Type</b>	Binary (Application)
<b>Host/Target Operating System</b>	Linux; macOS; iOS; Windows; Android; QNX
<b>Host/Target Architecture</b>	x86 (32, 64); ARM (32, 64)
<b>Initial Release</b>	December 31, 2013
<b>License Type</b>	Open-Source
<b>Maintenance</b>	Maintained by Ole André V. Ravnås and Håvard Sørbø

### Overview

Frida is a dynamic instrumentation toolkit useful in reverse engineering and security research. It injects a JavaScript engine into a native application running on one of the supported operating systems, and then executes JavaScript code in the target process’ context. That script has full access to memory, hooking functions and native functions inside the process. Users primarily interact with Frida via its Python library, which in turn leverages Frida’s JavaScript API to hook into the process and run JavaScript. However, Frida’s capabilities goes far beyond this, giving the user access to a JavaScript API, a C API, a Swift API, and providing a number of tools including Frida CLI, frida-ps, frida-trace, frida-discover, frida-ls- devices, and frida-kill [52], [53].

### Design and Usage

Frida has three modes of operation, Injected, Embedded and Preloaded.

**Injected** – Frida creates a shared object that contains an agent consisting of all user-defined logic that will be injected into a remote process. Under the hood, Frida uses operating system specific means (e.g., ptrace()) to hijack an existing thread and launch a new thread with the agent attached that will then internally communicate with the process. This is a hook that will facilitate access and modification to memory, registers, and the threads of the process. In addition to

setting up the connection, Frida supplies an importable library with calls to inspect a process' functions, modify their arguments, and create custom calls. Frida's library comes with methods like `Memory.alloc()` and `Memory.protect()` to manipulate process memory and create structs and arrays.

With these commands and capabilities, Frida can serve as a way to facilitate the fuzzing and testing of a process by injecting strings/integers/structs into memory, calling a function, and observing the resulting behavior [54].

**Embedded** – In embedded mode, Frida provides a shared library, `frida-gadget`, which leverages injection mode capabilities on incompatible systems, such as those that require explicit root privileges and/or are jail-broken. The gadgets modify the code of a program, patch certain libraries, and reflect these changes into the dynamic linker at run-time. This enables functions like `ptrace()` to be called with normal user privileges. Frida-gadgets comes with “Listen” mode, similar to Frida-server in the injection mode, which can be used to listen to a specific connection. Embedded mode can also be used in conjunction with LIEF to modify ELF formats by embedding the frida-agent as a dependency of native libraries. After invoking the `parse` method of LIEF on a Frida-agent then calling `.add_library` to inject it into a process' native library, it is possible to observe, log, and modify code on the process [54].

**Preloaded** – Preloaded mode uses dynamic linker capabilities like `LD_PRELOAD` and `DYLD_INSERT_LIBRARIES` to load pre-existing scripts from the filesystem into a process before it is executed. This is useful when running a process over a connection, to ensure that Frida-gadgets are executed from the entry-point after successfully connecting to a socket. Frida also has `ScriptDirectory` interaction which is used to run scripts based on the particular process. This is specified in a configuration file that has a filter object that defines under what conditions a script should be run [54].

## Use Cases and Limitations

Frida is designed for dynamic reverse engineering and offers a powerful range of capabilities. Developers can use it to easily add custom diagnostics and logging without creating an entirely new build or black-box test production code. Reverse engineers can inject code to trace execution, spy on processes and dump runtime information. The ability to inject custom JavaScript into a process opens the door to new methods of reverse engineering and dynamic binary instrumentation and enables many operations that are currently performed by more cumbersome tooling. It can also be used with Lighthouse, the code coverage tool also described in this report. Frida is supported by a number of operating systems and has bindings for a wide variety of platforms like Swift, .NET and Node.js. [53].

Frida is designed primarily for native desktop applications, and therefore lacks the ability to work with low level software and less common operating systems and architectures. Even native desktop applications may not be feasible to use with a tool that uses `ptrace` to inject a JavaScript engine into a target [53].

### 3.5.3 rr

Reference Link	<a href="https://rr-project.org/">https://rr-project.org/</a>
Target Type	Binary
Host/Target Operating System	Linux; macOS; Windows;
Host/Target Architecture	x86 (32, 64);
Initial Release	March 24, 2014
License Type	Open-Source
Maintenance	Maintained by rr project community, Mozilla

#### Overview

rr is an open-source debugger which enables the user to deterministically replay execution. It also enables users to ‘reverse-execute’ to where a breakpoint was hit in gdb. rr seeks to address the inability to deterministically retrigger crashes and events found with fuzzing and debugging [55]. In their 2017 paper *Engineering Record and Replay for Deployability, Extended Technical Report* [56] by Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll and Nimrod Partush, the authors of rr described how and why they developed this tool.

“The ability to record and replay program executions with low overhead enables many applications, such as reverse-execution debugging, debugging of hard-to-reproduce test failures, and “black box” forensic analysis of failures in deployed systems. Existing record-and-replay approaches limit deployability by recording an entire virtual machine (heavyweight), modifying the OS kernel (adding deployment and maintenance costs), requiring pervasive code instrumentation (imposing significant performance and complexity overhead), or modifying compilers and runtime systems (limiting generality). We investigated whether it is possible to build a practical record-and-replay system avoiding all these issues. The answer turns out to be yes — if the CPU and operating system meet certain non-obvious constraints. Fortunately, modern Intel CPUs, Linux kernels and user-space frameworks do meet these constraints, although this has only become true recently. With some novel optimizations, our system RR records and replays real-world low parallelism workloads with low overhead, with an entirely user-space implementation, using stock hardware, compilers, runtimes and operating systems [56].”

A program can be run and recorded with rr and then re-run with the exact same execution as the recorded run. This replay can then be debugged using common gdb commands [55].

#### Design and Usage

rr is run via command line, with the user first recording their application with the command:

```
rr record /path/to/application --arguments
```

This recording will be saved to the disk, including any crash or failure that caused the issue. The user can then run this recording with the command:



rr replay

This enables the user to debug the recording, which replays every part of the recorded execution exactly as it happened (to include memory allocations), rather than a nondeterministic re-execution of the program.

### Use Cases and Limitations

The ability to deterministically replay crashes and failures is useful in a number of areas of reverse engineering, program analysis and exploitation, from determining how a program works, to examining an input discovered from fuzzing, to testing an exploit.

## 3.5.4 FUZE

Reference Link	<a href="https://github.com/ww9210/Linux_kernel_exploits">https://github.com/ww9210/Linux_kernel_exploits</a>
Target Type	Linux Kernel (32, 64)
Host/Target Operating System	Linux;
Host/Target Architecture	x86 (32, 64);
Initial Release	August 14, 2018
License Type	Open-Source
Maintenance	Last commit September 2018

### Overview

FUZE is a Python tool for exploiting Linux kernels. It is an implementation of a 2018 USENIX paper entitled *FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities* by Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xin, Xiaorui Gong, Wei Zou. [57] FUZE, as the title suggests, provides a framework for exploiting UAF Linux Kernel vulnerabilities. The authors describe it thusly,

“The design principle behind this technique is that we expect the ease of crafting an exploit could augment a security analyst with the ability to evaluate the exploitability of a kernel UAF vulnerability. Technically, FUZE utilizes kernel fuzzing along with symbolic execution to identify, analyze and evaluate the system calls valuable and useful for kernel UAF exploitation. In addition, it leverages dynamic tracing and an off-the-shelf constraint solver to guide the manipulation of vulnerable object [57, p. 1].”

The underlying technique of FUZE consists of three parts. First, it extracts information relating to the spatial and temporal metadata of the vulnerability, then uses fuzzing to find different contexts which trigger kernel panics, and subsequently symbolic execution to explore exploitability in those contexts.

The Heap Attacks subsection of the Other Techniques portion of this report contains a discussion of exploit templates surrounding another somewhat similar (in intention) framework created by security researcher Sean Heelan and others. In a blog post on the subject, Heelan



predicts that the “future of automatic exploit generation is going to look more like template-based approaches than end-to-end solutions [58].” Although this tool is arguably a beta implementation of a research paper, is not actively maintained, and lacks the documentation necessary to be easily useable, it is worth discussing as an example of an exploit framework.

## **Design and Usage**

As described above, FUZE extracts metadata information about the vulnerability, fuzzes to find new contexts which trigger kernel panics, and then symbolically executes over those contexts to explore their exploitability. It is worth reading their summary of how FUZE works:

“To be more specific, our system first takes as input a PoC program which does not perform exploitation but causes a kernel panic. Then, it utilizes kernel fuzzing to explore various system calls and thus to mutate the contexts of the kernel panic. Under each context pertaining to a distinct kernel panic, FUZE further performs symbolic execution with the goal of tracking down the primitives potentially useful for exploitation. To pinpoint the primitives truly valuable for exploiting a UAF vulnerability and even bypassing security mitigation, FUZE summarizes a set of exploitation approaches commonly adopted, and then utilizes them to evaluate primitives accordingly [57, p. 2].”

This tool lacks usage documentation and exists as a set of Python libraries. Given its lack of current maintenance, it is incumbent on the user to read through the source code and examples to discern usage. (There are numerous examples provided, corresponding to CVEs.) It is, however, configured for installation which is easily done with Python’s `setuptools` package. The tool consists of four packages, `vminstance`, `concolicexecutor`, `statebroker` and `kernelrop`. The tool relies on a modified version of `angr` [59].

## **Use Cases and Limitations**

The largest and most apparent limitation of this tool is the lack of usability and maintenance. However, it does provide an example framework for partial automated exploit development, specifically for developing multiple exploits for a known vulnerability. This is useful in evaluating exploitability and enabling software vendors to assess on which vulnerabilities to focus. Conversely, this also helps attackers determine which vulnerabilities to focus on and helps them craft exploits.

## 3.6 Symbolic and Concolic Execution

Symbolic execution is an aptly named technique that involves executing a program with symbols rather than concrete values. In order to reason about a symbolically executed path, the engine compiles a logical expression representing the constraints that resulted from the execution of the path. Then an SMT solver (e.g., z3) solves for a concrete value to each constraint. The primary purpose of symbolic execution in a vulnerability research context is similar to that of fuzzing, to explore program execution to find crashes and vulnerabilities.

There are many symbolic execution engines available today, often available as open-source tools. The current state-of-the-art began with KLEE, which was presented in a 2008 paper and is built on top of LLVM. Another symbolic execution engine, SAGE, was developed by Microsoft as an internal tool. Many subsequent engines have emerged, including angr, which was released in 2013 and has become the state-of-the-art.

Symbolic execution, although powerful, comes with significant limitations. For example, it notoriously suffers from the path explosion problem, which results in exorbitant computational and memory costs. Concolic execution was developed to address these issues by combining concrete execution with symbolic (many symbolic execution machines like angr use this approach). Merging concolic execution with fuzzing has produced hybrid fuzzing engines like Driller, which is derived from a combination of angr and AFL.

QSYM offers a new approach to the hybrid fuzzing technique, along with two other new tools that have emerged in the last two years. MemSight relies on symbolic pointer reasoning to address the cost issue of mapping concrete memory addresses to data. Manticore, on the other hand, does not provide a novel technique, but implements a simple and usable symbolic execution engine addressing many of the usability issues of these tools.

Symbolic execution continues to become more sophisticated, effective, and less costly with regard to time and space. There are currently numerous options for symbolic execution, and much cross pollination (e.g., MemSight could be integrated into angr). As both symbolic execution and fuzzing advance, so too will hybrid approaches. Symbolic execution will also become available to more platforms, including embedded systems.

### 3.6.1 angr

Reference Link	<b>angr:</b> <a href="http://angr.io/">http://angr.io/</a> <b>Driller:</b> <a href="https://github.com/shellphish/driller">https://github.com/shellphish/driller</a>
Target Type	Binary
Host/Target Operating System	<b>angr:</b> Linux; macOS; Windows <b>Driller:</b> Linux
Host/Target Architecture	<b>angr:</b> x86 (32, 64); ARM (32, 64); PPC (32, 64); MIPS (32, 64); Java via Soot, etc. <b>Driller:</b> x86 (32, 64)
Initial Release	<b>angr:</b> 2013 <b>Driller:</b> 2016
License Type	Open-Source
Maintenance	Maintained by Computer Security Lab at UCSB and SEFCOM at Arizona State University

#### Overview

angr is one of the current state-of-the-art symbolic execution engines and offers a wide array of capabilities. Built as a Python framework, angr targets binaries of many different architectures. In addition to symbolic execution angr includes **angrop** which builds ROP chains, **angr-managment**, a GUI for analyzing binaries with angr, **rex** which automatically generates exploits, and the **Ramblr** reassembler [60], [82].

#### 3.6.1.1 Driller

Driller is a concolic execution engine that combines the symbolic execution of angr with the AFL fuzzer. This tool implements the paper *Driller: Augmenting Fuzzing Through Selective Symbolic Execution* by Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel and Giovanni Vigna. The following excerpt from the paper explains how Driller’s core components function.

**“Input test cases.** Driller can operate without input test cases. However, the presence of such test cases can speed up the initial fuzzing step by pre-guiding the fuzzer toward certain compartments.

**“Fuzzing.** When Driller is invoked, it begins by launching its fuzzing engine. The fuzzing engine explores the first compartment of the application until it reaches the first complex check on specific input. At this point, the fuzzing engine gets “stuck” and is unable to identify inputs to search new paths in the program.

**“Concolic execution.** When the fuzzing engine gets stuck, Driller invokes its selective concolic execution component. This component analyzes the application, pre-constraining the user input with the unique inputs discovered by the prior fuzzing step to prevent a path explosion. After tracing the inputs discovered by the fuzzer, the concolic execution

component utilizes its constraint-solving engine to identify inputs that would force execution down previously unexplored paths. If the fuzzing engine covered the previous compartments before getting stuck, these paths represent execution flows into new compartments.

**“Repeat.** Once the concolic execution component identifies new inputs, they are passed back to the fuzzing component, which continues mutation on these inputs to fuzz the new compartments. Driller continues to cycle between fuzzing and concolic execution until a crashing input is discovered for the application [23, p. 4].”

The example usage from the Driller repository uses a testcase’s trace to generate new test cases and is shown in Figure 3-20.

```
import driller

d = driller.Driller("./CADET_00001", # path to the target binary
                  "racecar", # initial testcase
                  "\xff" * 65535, # AFL bitmap with no discovered transitions
                  )

new_inputs = d.drill()
```

Figure 3-20: Example of using Driller from the README [61]

## New or Notable Features

**Veritesting** – Veritesting is a symbolic execution technique implemented in the paper, *Enhancing Symbolic Execution with Veritesting* by Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley of Carnegie Mellon University [62]. Veritesting is described in the paper as follows.

“At a high level, there are two main approaches for generating formulas. First, dynamic symbolic execution (DSE) explores programs and generates formulas on a per-path basis. Second, static symbolic execution (SSE) translates program statements into formulas, where the formulas represent the desired property over any path within the selected statements.

...

“The path-based nature of DSE introduces significant overhead when generating formulas, but the formulas themselves are easy to solve. The statement-based nature of SSE has less overhead and produces more succinct formulas that cover more paths, but the formulas are harder to solve.

“Veritesting alternates between SSE and DSE. The alternation mitigates the difficulty of solving formulas, while alleviating the high overhead associated with a path-based DSE approach. In addition, DSE systems replicate the path-based nature of concrete execution, enabling them to handle cases such as system calls and indirect jumps where static approaches would need summaries or additional analysis. Alternating enables [the symbolic execution system] with veritesting to switch to DSE-based methods when such cases are encountered [62, p. 1].”

angr implements the veritesting technique in the script `angr/veritesting`.

**Ramblr** – Ramblr is a static binary rewriter presented in the 2017 paper *Ramblr: Making Reassembly Great Again* by Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna of University of California Santa Barbara. [43] It is implemented in the `angr/reassembler.py` script. For further discussion on reassemblers, refer to the DDisasm tool.

**Symbion** – Symbion is a new path exploration technique designed to replace modeling for complex procedures. Modeling enables more deliberate symbolic execution and minimizes path explosion, however not every possible procedure can have a prebuilt model. Symbion addresses this by concretely executing through a complex procedure and then switching to symbolic execution. A blogpost by the maintainers of angr expands on its capability in the following excerpt accompanied by the graphic in Figure 3-21.

“Analysts may wish to symbolically reason about control flow of a program between two program points B and C but cannot even execute from point A to point B due to unmodeled behaviors. With Symbion, they can execute concretely up to point B, switch into angr’s symbolic context, and compute the program input needed to reach point C. The solution obtained by angr can then be written into the program’s memory and by resuming the concrete execution reaching beyond point C [63].”

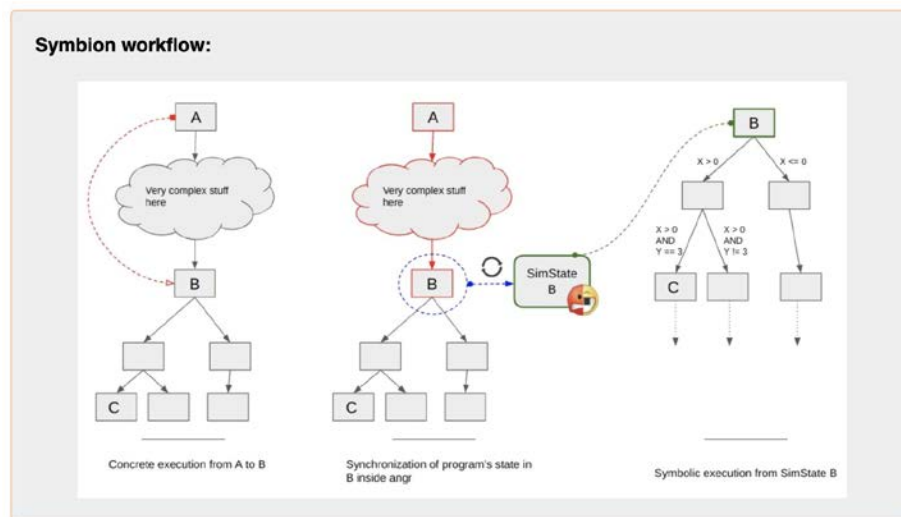


Figure 3-21: Overview of the Symbion path exploration technique [63]

## 3.6.2 Manticore

Reference Link	<a href="https://github.com/trailofbits/manticore/">https://github.com/trailofbits/manticore/</a>
Target Type	Binary
Host/Target Operating System	Linux; macOS
Host/Target Architecture	x86 (32, 64); ARM (32, 64); EVM/Native Bytecode Executables
Initial Release	May 5, 2017
License Type	Open-Source
Maintenance	Maintained by Trail of Bits

### Overview

Trail of Bits bills their Manticore tool as “symbolic execution for humans”, a tool which enables the user to avail themselves of “symbolic execution, taint analysis and instrumentation” to analyze binaries. Manticore’s main capabilities are input generation, error discovery, execution tracing and a programmatic interface, all of which were designed with Manticore’s goal of simplicity and usability in mind [64], [65].

### Design and Usage

Manticore prioritizes simplicity and usability, by eschewing an intermediate representation and minimizing external dependencies while creating a usable Python API and command line tool. The Manticore Documentation offers the following overview of its capabilities:

- “Input Generation: Manticore automatically generates inputs that trigger unique code paths” [65]
- “Error Discovery: Manticore discovers bugs and produces inputs required to trigger them” [65]
- “Execution Tracing: Manticore records an instruction-level trace of execution for each generated input” [65]
- “Programmatic Interface: Manticore exposes programmatic access to its analysis engine via a Python API” [65]

Manticore is a tool based around Python and as such can be installed using pip. Once installed it can either be used via the command line or API. Figure 3-22 shows an example of using Manticore as a command line tool [65].

```
$ manticore ./path/to/binary          # runs, and creates a mcore_* directory with analysis results
$ manticore ./path/to/binary ab cd    # use concrete strings "ab", "cd" as program arguments
$ manticore ./path/to/binary ++ ++    # use two symbolic strings of length two as program arguments
```

*Figure 3-22: Example of using Manticore as a command line tool [65]*

Manticore’s more extensive capability is its Python API. Figure 3-23 shows an example of a basic Manticore Python script to evaluate a binary that takes an input and decrypts a key for a

certain set of inputs. The “hook” call acts as a breakpoint which freezes the state and enables the user to execute a function before the instruction at that address is called [64], [65].

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  from manticore.native import Manticore
5
6  m = Manticore("./lab1B")
7  m.verbosity(1)
8
9  """
10 This lab has 21 unique cases equivalent to
11 switch(0x1337d00d - input):
12     case(1):
13         ...
14     case(2):
15         ...
16     ...
17     case(21):
18         ...
19
20 by setting our input to 0x1337d00d - 1, we ensure we will hit the first case
21 """
22 m.context["password"] = 0x1337D000 - 1
23
24
25 @m.hook(0x8048A55)
26 def bad_password(state):
27     """
28     If this address is reached, the password check has failed. Luckily, there
29     are a limited number of possible cases. We can decrement our input to reach
30     the next case, then manually jump back to the switch
31     """
32     with m.locked_context() as context:
33         print("[~] abandoning path (invalid password)")
34
35         context["password"] -= 1
36         state.cpu.EIP = 0x8048BF6
37

```

Figure 3-23: A basic Manticore Python script to evaluate a binary that takes an input and decrypts a key for a certain set of inputs

```

38
39 @m.hook(0x8048A4E)
40 def success(state):
41     """
42     If this code is reached, our password must have been correct. Dump our input
43     when this address is reached.
44     """
45     with m.locked_context() as context:
46         print("[+] found success path")
47         print("[+] password: {}".format(context["password"]))
48         m.terminate()
49
50
51 @m.hook(0x8048BF6)
52 def inject_data(state):
53     """
54     Instead of sending out input through stdin, it's more efficient to jump
55     over calls to i/o functions like fgets or puts and inject our data
56     manually onto the stack. Because these libc functions are so massive, this
57     can give us significant performance improvements.
58     """
59     with m.locked_context() as context:
60         # skip ahead several instructions to jump over puts/fgets/scanf
61         state.cpu.EIP = 0x8048C52
62
63         print("[+] injecting " + hex(context["password"]))
64         state.cpu.EAX = context["password"]
65
66
67 m.run(procs=10)

```

Figure 3-24: manticore-examples/RPISEC\_MBE/lab1B.py [65]

Manticore can be used for crash analysis and exploit generation; rather than exploring all possible paths, the user can explore the path known to trigger a crash to create a working payload.

### Use Cases and Limitations

Manticore is useful in cases where the user needs to perform symbolic analysis and tracing (with objectives such as program exploration, crash analysis and exploit generation), with a simple tool that has an easy to use API. Manticore was designed to be used without an extensive background in program analysis or reverse engineering. In their documentation, Trail of Bits offers a comparison between Manticore and angr that describes their intentions with the tool:

“Manticore is simpler. It has a smaller codebase, fewer dependencies and capabilities, and an easier learning curve. If you come from a reverse engineering or exploitation background, you may find Manticore intuitive due to its lack of intermediate representation and overall emphasis on staying close to machine abstractions [66].”

However, the simplicity also brings limitations. Manticore lacks the breadth of angr’s analysis capabilities. Manticore is mainly a simple symbolic execution engine, and because it lacks an IR, it cannot offer analysis support similar to angr.

One of Manticore’s unique capabilities is its support for Ethereum smart contracts, including capabilities and documentation specific to the Ethereum EVM.

### 3.6.3 QSYM

Reference Link	<a href="https://github.com/sslabs-gatech/qsym">https://github.com/sslabs-gatech/qsym</a>
Target Type	Binary
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	August 16, 2018
License Type	Open-Source
Maintenance	Last commit December 2018

#### Overview

This hybrid fuzzer is the implementation of a paper presented at USENIX 2018 entitled *QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing* [67]. The intention of the fuzzer is to reform current hybrid fuzzing techniques to make them scalable and more effective. The authors write that

“The key idea is to tightly integrate the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained, so faster, instruction-level symbolic emulation.



Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, yet takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance optimizations, e.g., optimistically solving constraints and pruning uninteresting basic blocks [67 p. 1].”

The authors found that they were able to achieve performance gains in their evaluations over state-of-the-art fuzzers, in addition to finding multiple vulnerabilities. QSYM is built with AFL.

## Design and Usage

The user works within a Python virtualenv and then can run Hybrid fuzzing with AFL via the following sample command-line input from the GitHub README.

```
# require to set the following environment variables
#   AFL_ROOT: afl directory (http://lcamtuf.coredump.cx/afl/)
#   INPUT: input seed files
#   OUTPUT: output directory
#   AFL_CMDLINE: command line for a testing program for AFL (ASAN
+ instrumented)

#   QSYM_CMDLINE: command line for a testing program for
QSYM (Non-instrumented)

# run AFL master
$ $(AFL_ROOT)/afl-fuzz -M afl-master -i $(INPUT) -o $(OUTPUT) --
$(AFL_CMDLINE)
# run AFL slave
$ $(AFL_ROOT)/afl-fuzz -S afl-slave -i $(INPUT) -o $(OUTPUT) --
$(AFL_CMDLINE)
# run QSYM
$ bin/run_qlsym_afl.py -a afl-slave -o $(OUTPUT) -n qlsym --
$(QSYM_CMDLINE)
```

## Use Cases and Limitations

QSYM presents a technique which can be implemented with current state-of-the-art fuzzers and symbolic execution engines to increase the efficacy and yield of hybrid execution, while making this software testing technique more scalable than it has been.

### 3.6.4 MemSight

Reference Link	<a href="https://github.com/season-lab/memsight">https://github.com/season-lab/memsight</a>
Target Type	Binary
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	2017
License Type	Open-Source
Maintenance	Maintained by SEASON Lab

#### Overview

MemSight is an open-source symbolic execution tool that is built on angr and based on a 2017 paper presented at the ASE conference, *Rethinking Pointer Reasoning in Symbolic Execution*.

“When memory addresses are symbolic, and could possibly refer to large areas of memory, this is handled by concretizing the address [69]. MemSight seeks to address this by introducing a different approach. “Rather than mapping address instances to data as previous tools do, our technique maps symbolic address expressions to data, maintaining the possible alternative states resulting from the memory referenced by a symbolic address in a compact, implicit form [69, p. 1].”

The implementation accompanying the paper is written in Python and built on angr.

#### Usage

The MemSight repository contains two main Python scripts for exploration. MemSight/explore.py does exploration line-by-line, and MemSight/run.py does not. Both take as arguments a path to the ‘MetaBinary’ which is a binary and a configuration script to setup exploration.

#### Use Cases and Limitations

The paper claims MemSight reduces the need for concretization and enables more precise pointer reasoning and broader state explorations. This explores alternative program states which are missed due to current symbolic memory concretization techniques. This increases the number of possible programmatic paths explored, expanding and making more comprehensive the analysis a symbolic execution can perform.

## 4 Other Techniques

Many new and popular techniques were discussed in the preceding Tools section, such as new research into concolic fuzzing or replayable debugging. This section will address the techniques that do not directly correspond with one or more tools.

### 4.1 JIT Attacks

Browser based attacks have become one of the most popular areas of exploitation, with specific focus on JIT based attacks. A JIT engine generates executable code at runtime, which provides a number of potentially exploitable attack vectors.

**JIT Spray** – In their 2018 paper, *Make JIT Spray Great Again* [70], Robert Gawlik and Thorsten Holz of Ruhr-Universitat Bochum surveyed the landscape of JIT attacks and their current mitigations. They discuss several different types of JIT attacks, like JIT Spray:

“If expressions with constant values of a high-level language are *Just-In-Time (JIT)* compiled into native code, they can be abused to embed malicious code bytes at run time. This bypasses DEP because data is (indirectly) injected as code. Additionally, if the adversary manages to create many regions of this code, their locations become predictable. Hence, by *spraying* many code regions, she can predict the address of one region to bypass ASLR. Finally, only control over the instruction pointer is needed to redirect the control flow to the injected code. Thereby, a *use-after-free*, *type confusion* or *heap-buffer overflow* vulnerability is sufficient [70, p. 1].”

Gawlik and Holz found that, while protections against JIT-Spray attacks have improved, JIT-Spraying is still an effective attack. In 2016, they noted that “WebGL *shaders* were usable inside JavaScript of Internet Explorer and Microsoft Edge.

“The WARP JIT compiler produced native code not protected by MS-CFG. Thus, the authors were able to inject code to predictable addresses with the *Windows Advanced Rasterization Platform (WARP) Shader JIT* compiler [67, p. 4].”

**JIT-Based Code Reuse** – A more recent attack, they claim, is JIT-Based Code Reuse:

“The first (academic) work which used runtime compiled gadgets from a JIT compiler arose from the need to bypass code-reuse protections in 2015. If static code of a program is *gadget-free*, then code-reuse is usually not an option. However, if gadgets are produced by the JIT compiler, code-reuse becomes feasible again. Athanasakis et al. targeted IonMonkey on 32-bit Linux and Chakra of 64-bit Internet Explorer 9 on Windows. In general, they provoked the JIT compiler to emit gadgets containing only a few instructions and were using two-byte JavaScript constants. This bypassed constant blinding in Internet Explorer and various other JIT-related defenses which were incorporated at that time. However, note that the authors needed memory disclosures to locate the gadgets in memory. Hence, we do not count it as JIT-Spray because JIT-Spray does not require info leaks but only control over the

instruction pointer to redirect control flow to a predetermined address containing the JIT-compiled attacker code [67, p. 7].”

**Mitigation Bypasses** – Mitigations are often what give rise to new techniques, developed to bypass them. Gawlike and Holz discuss several such bypasses, including one from 2017, in which a data-only attack on Microsoft Edge’s JavaScript Engine targeted Chakra’s Intermediate Representation (IR). “Instead of creating/modifying code or code pointers, they crafted malicious C++ object representing IR statements with the prerequisite of a read/write primitive from within JavaScript. As the JIT compiler uses these objects to generate native code, the authors were able to create and execute their code of choice [67, p. 8].”

In 2015, a paper which laid out the technique JIT Code Reuse was published for the IEEE Symposium on Security and Privacy by Kevin Snow, Fabian Monroe, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad- Reza Sadeghi. In the paper, titled *Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization*, Snow et al. describe the technique they term JIT-ROP to bypass ASLR. Their “key observation” is that “by exploiting a memory disclosure multiple times we violate implicit assumptions of the fine-grained exploit mitigation model and enable the adversary to iterate over mapped memory to search for all necessary gadgets on-the-fly, regardless of the granularity of code and memory randomization [71, p. 1].” They offer a diagram of their attack (Figure 4-1) and explain it as follows:

“An adversary constructing a new exploit need only conform their memory disclosure to our interface and provide an initial code pointer in Step 1, then let our framework take over in Steps 2 to 5 to automatically (and at exploit runtime) harvest additional code pages, find API functions and gadgets, and just-in-time compile the attacker’s program to a serialized payload useable by the exploit script in Step 6 [71, p.2].”

**JIT Compiler Logic Errors** – As JIT machines are hardened against these kinds of attacks, JIT Compilers are an increasingly common attack vector, especially as optimizations unintentionally add logic bugs. These bugs are ones that lead to generating incorrect executable code that can subsequently be exploited. Some of these include bugs in bounds-check elimination, escape analysis, register allocation and redundancy elimination [72]. A common bug pattern is assumptions at compile time without corresponding runtime checks [72]. These are the kind of logic bugs sought by the fuzzers Fuzzilli and CodeAlchemist, which are discussed in the tools section.

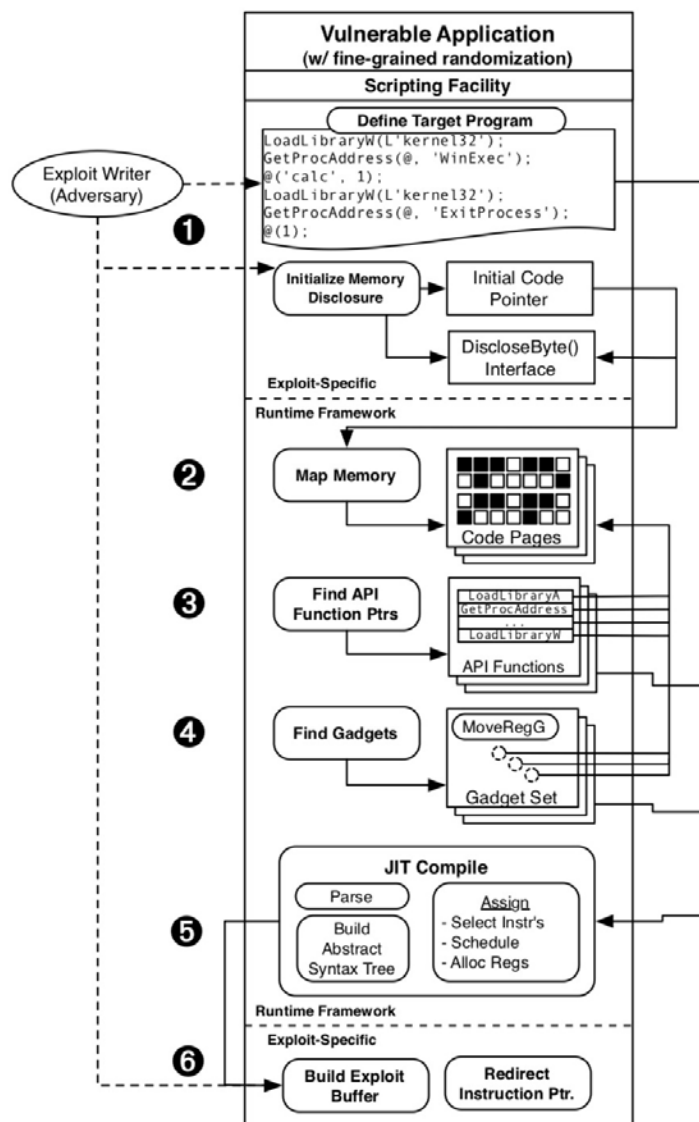


Figure 4-1: An overview of JIT code reuse [71]

## 4.2 ROP-Based Attacks

In addition to JIT-ROP, described above, several attack techniques based on Return-Oriented Programming (ROP) attacks have emerged in recent years.

**Blind Return Oriented Programming (BROP)** – Around 2014, BROP emerged as a way to use ROP without having a binary or source code. A Stanford University paper titled *Hacking Blind* by Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, Dan Boneh [73] first described the technique. A subsequent Stanford University post elaborated as follows:

“The BROP attack makes it possible to write exploits without possessing the target's binary. It requires a stack overflow and a service that restarts after a crash. Based on whether a service crashes or not (i.e., connection closes or stays open), the BROP attack is able to construct a full remote exploit that leads to a shell. The BROP attack remotely leaks enough gadgets to perform the write system call, after which the binary is transferred from memory to the attacker's socket. Following that, a standard ROP attack can be carried out [73].”

Apart from attacking proprietary services, BROP is very useful in targeting open-source software for which the particular binary used is not public (e.g., installed from source setups, Gentoo boxes, etc.) [74].

They also provide a more specific attack outline (quoted below):

1. “Break ASLR by "stack reading" a return address (and canaries).
2. “Find a "stop gadget" which halts ROP chains so that other gadgets can be found.
3. “Find the BROP gadget which lets you control the first two arguments of calls.
4. “Find a call to strcmp, which as a side effect sets the third argument to calls (e.g., write length) to a value greater than zero.
5. “Find a call to write.
6. “Write the binary from memory to the socket.
7. “Dump the symbol table from the downloaded binary to find calls to dup2, execve, and build shellcode [74].”

**Sigreturn Oriented Programming (SROP)** – In 2014, Erik Bosman and Herbert Bos of Vrije Universiteit laid out their technique, SROP, in a paper for the IEEE Symposium. They described their attack as similar to ROP in that “sigreturn oriented programming constructs what is known as a ‘weird machine’ that can be programmed by attackers to change the behavior of a process [75].”

However, to program this machine, “attackers set up fake signal frames and initiate returns from signals that the kernel never really delivered. This is possible, because UNIX stores signal frames on the [process’s] stack.” They assessed the significance of their technique as follows.

“Sigreturn oriented programming is interesting for attackers, OS developers and academics. For attackers, the technique is very versatile, with preconditions that are different from those of existing exploitation techniques like ROP. Moreover, unlike ROP, sigreturn oriented programming programs are portable. For OS developers, the technique presents a problem that has been present in one of the two main operating system families from its inception, while the fixes (which we also present) are non-trivial. From a more academic viewpoint, it is also interesting because we show that sigreturn oriented programming is Turing complete [75, p. 1].”

**Tainted-Based Return Oriented Programming (T-ROP)** – In 2018, Colas Le Guernic and François Khourbiga laid out their T-ROP technique [76]. They describe it as follows.

“There are roughly two kinds of tools for return oriented programming (ROP):

“\_syntactic\_ tools that return the disassembly of gadgets and sometimes perform template based automatic chaining, and \_symbolic\_ tools that compute a symbolic representation of the output state for each gadget and enable more powerful manipulations. The former are very fast but only enable regex queries, the latter enable symbolic queries but are much slower. We propose an intermediate approach, faster than symbolic tools and enabling more expressive queries than syntactic tools: taint-based ROP (T-Brop). T-Brop uses a coarse semantic of instructions. Instead of a precise symbolic I/O relationship, it only relies on a dependency matrix reflecting how a taint would be propagated by a given gadget [77].”

## 4.3 Heap Attacks

**Automatic Heap Layout Manipulation for Exploitation** – The heap has long been a popular target for exploitation, and as such many heap attacks, especially generalized ones, have been well known for decades. However, not much had been done in the way of automating and facilitating heap attacks, until recently. In their 2018 paper, *Automatic Heap Layout Manipulation for Exploitation* [78], presented at the USENIX conference, Sean Heelan, Tom Melham, and Daniel Kroenig present “the first automatic approach” to heap layout manipulation. Heelan summarizes the paper as follows:

“The main idea of the paper is that we can isolate heap layout manipulation from much of the rest of the work involved in producing an exploit, and solve it automatically using blackbox search [58].”

Reasoning about heap layout is a major step in constructing heap-based exploits such that the exploit will function properly while maintaining the integrity of the heap. Heap layout manipulation is hindered by other unwanted but unavoidable heap calls which create interference, and by the diversity and constraints of heap allocators. The authors offer the following overview of heap manipulation.

“An analyst examines the allocator’s implementation to gain an understanding of its internals; then, at run-time, they inspect the state of its various data structures to determine what interactions are necessary in order to manipulate the heap into the required layout. Heap layout manipulation primarily consists of two activities: creating and filling *holes* in memory. A hole is a free area of memory that the allocator may use to service future allocation requests. Holes are filled to force the positioning of an allocation of a particular size elsewhere, or the creation of a fresh area of memory under the management of the allocator. Holes are created to capture allocations that would otherwise interfere with the layout one is trying to achieve [78, p. 4].”

The authors seek to automate heap layout manipulation by addressing a problem they construct with a limited scope: “finding a solution that places two allocations in memory at a specified distance from each other [58].”

They attempt a pseudo-random black box search approach for this and found that with no interference and no segregated storage it performed very well and could be leveraged with the appropriate computational resources. They subsequently apply the technique to the PHP Language Interpreter [78].

Moving forward more research in this area could produce tooling that effectively automates heap layout manipulation, making it easier for exploiters to manage the heap.

**Exploit Templates** – The paper *Automatic Heap Layout Manipulation for Exploitation* also explores the idea of exploit templates, which Heelan expands on subsequently in a blog post on the subject. He describes an exploit template as follows and includes an example found in Figure 4-2.

“An exploit template is a simply a partially completed exploit where the incomplete parts are to be filled in by some sort of automated reasoning engine. In the case of the above paper, the parts filled in automatically are the inputs required to place the heap into a particular layout. Here’s an example template [Figure 4-2], showing part of an exploit for the PHP interpreter. The exploit developer wants to position an allocation made by *imagecreate* adjacent to an allocation made by *quoted\_printable\_encode*.

“SHRIKE (the engine that parses the template and searches for solutions to heap layout problems) takes as input a .php file containing a partially completed exploit, and searches for problems it should solve automatically. Directives used to communicate with the engine begin with the string *X-SHRIKE*. They are explained in full in the above paper, but are fairly straightforward: *HEAP-MANIP* tells the engine it can insert heap manipulating code at this location, *RECORD-ALLOC* tells the engine it should record the nth allocation that takes place from this point onwards, and *REQUIRE-DISTANCE* tells the engine that at this point in the execution of the PHP program the allocations associated with the specified IDs must be at the specified distance from each other. The engine takes this input and then starts searching for ways to put the heap into the desired layout [58].”

```
1  $quote_str = str_repeat("\xf4", 123);
2
3  #X-SHRIKE HEAP-MANIP 384
4  #X-SHRIKE RECORD-ALLOC 0 1
5  $image = imagecreate(1, 2);
6
7  #X-SHRIKE HEAP-MANIP 384
8  #X-SHRIKE RECORD-ALLOC 0 2
9  quoted_printable_encode($quote_str);
10
11 #X-SHRIKE REQUIRE-DISTANCE 1 2 384
```

Figure 4-2: An example exploit template [58]



Heelan goes on to expound on the benefits of this approach,

“So, what are the benefits of this approach? The search is black-box, doesn’t require the exploit developer to analyze the target application or the allocator, and, if successful, outputs a new PHP file that achieves the desired layout and can then be worked on to complete the exploit. This has the knock-on effect of making it easier for the exploit developer to explore different exploitation strategies for a particular heap overflow. In ‘normal’ software development it is accepted that things like long build cycles are bad, while REPLs are generally good. The reason is that the latter supports a tight loop of forming a hypothesis, testing it, refining and repeating, while the former breaks this process. Exploit writing has a similar hypothesis refinement loop and any technology that can make this loop tighter will make the process more efficient [58].”

Heelan also notes that the security industry tends to focus myopically on problems with only fully automated solutions as opposed to partially automated ones like this, and predicts that the “future of automatic exploit generation is going to look more like template-based approaches than end-to-end solutions” [58].

**Revery** - The paper *Revery: From Proof of Concept to Exploitable* by Wang et al. [84] proposes what they consider to be a novel method of Automatic Exploit Generation for heap-based vulnerabilities. This paper aims to address several issues with automatically generating exploits for heap-based vulnerabilities, including manipulating the program to put it in an exploitable state, issues caused by path-explosion in symbolic execution, and the complexity of heap management systems [84, p. 2].

Revery takes an already identified vulnerability from which it attempts to derive an exploit. This is a three step process that is described in the paper as follows:

- 1. Vulnerability Analysis:** Revery analyzes the given vulnerability by constructing “a layout-contributor diagram data structure” which characterizes the memory layout of the vulnerability [84, p. 3-4].
- 2. Diverging Path Exploration:** This step uses the constructed diagram to guide a fuzzer in order to find divergent paths with exploitable states. This process does not require symbolic execution, thus avoiding the issue of path explosion [84, p.3-4].
- 3. Exploit Synthesis:** Finally this process uses “a novel control-flow stitching solution, to stitch crashing paths and diverging paths together” and then uses “lightweight symbolic execution” to synthesize the exploit [84, p. 3-4].

## 4.4 Side Channel Attacks

Side channels seek to exploit a program through a “side door,” or by exploiting the implementation itself rather than a flaw in the code. These attacks have become very popular in recent years. Side channel attacks encompass a broad range of attacks, including software-based side channel attacks on which this section will primarily focus. Some software is more susceptible to these attacks, such as cryptographic implementations which require no information leaks and often involve cryptographic primitives with repetitious arithmetic operations. These attacks are also made easier by increasingly complex process level functionality, like multithreading and different kinds of caching. Cloud computing provides a good platform for these attacks, which can exploit the ability for one user to share hardware with a separate user.

**Page Cache Attacks** – A 2019 paper entitled *Page Cache Attacks* [79] describes a technique that exploits the page cache state as follows:

“Bringing the page cache into a known state is not trivial, as it behaves like a fully associative cache. Previous approaches for page cache eviction can lead to out-of-memory situations or consume too much time and impose system pressure. This is not practical when evicting pages often, e.g., multiple times per second. Hence, they have not been used in published side-channel attacks so far, but only to support other attacks, e.g., relocation of a page for Rowhammer. For Linux, we devise a working- set-based eviction strategy that efficiently accesses groups of other pages more frequently than the page to evict [79, p. 3].”

The paper also provides the following attack overview accompanied by the diagram in Figure 4-3.

“The attacker wants to measure when the function `foo()` is called by a victim program. The attacker determines the page which contains the function `foo()`. By observing when the page is in the page cache, the attacker learns when `foo()` was called.

Our attack continuously runs through the following steps: Initially, the target pages are in the page cache (on Linux) respectively the working set of the victim process (on Windows). After the eviction, the page is not in the page cache (Linux) or process working set (Windows) anymore. The attacker can now continuously probe when the page is added back in. As soon as the page is found in the page cache (Linux) or the process working set (Windows), the attacker logs the memory access and evicts the page again [79, p. 4].”

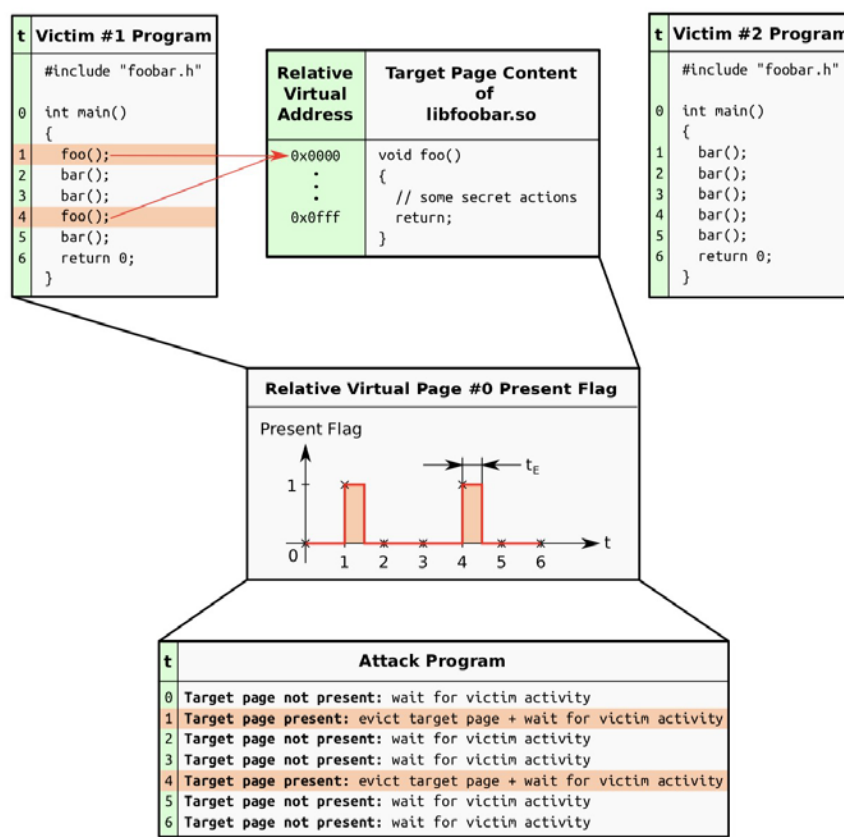


Figure 4-3: Overview of page cache attacks [79, Fig. 1]

**Differential Address Trace Analysis (DATA)** – DATA is a framework to detect address-based side channels in binaries using trace analysis. Although put forth as a defensive tool, its trace analysis techniques can also be used to locate these side channels and exploit them.

The methodology is described in the paper which presented the technique [80]:

“In the *difference detection phase*, we execute the target program multiple times with varying secret inputs and record all accessed addresses with dynamic binary instrumentation in so-called address traces. Thereby, we ensure to capture both, control flow and data leakages at their exact origin. The recorded address traces are then compared, and address differences are reported.

“The *leakage detection phase* verifies whether reported address differences are actually secret-dependent and filters all that are statistically independent. For this step, the program is repeatedly executed with one fixed secret input and a set of varying (random) secret inputs. In contrast to the previous phase, only the initially reported differences need to be monitored. The address traces belonging to the fixed input are then compared to those of the random inputs using a *generic* leakage test. Statistical differences are reported as true information leaks [80, p. 2].”

**Hardware-Based Attacks** – In recent years hardware-based side channel attacks have become increasingly popular. SPECTRE and MELTDOWN have disrupted the security industry with their severity and lack of simple solutions, such as patching. These attacks exploit speculative execution and branch prediction to leak data. The most recent in this series are ZombieLoad, Fallout and RIDL. Many of these attacks are not remote, reducing their efficacy significantly.

## 5 Conclusion

This Edge of the Art report has captured both the state-of-the-art and significant advances to it. In doing so it defines what “edge” means, both in abstract as well as in concrete terms. Subsequent EotA reports will discuss what new and novel tools and techniques are developed after this report, to keep pace with the ever-expanding boundary that is the “edge” of the vulnerability discovery and exploitation discipline.

## 6 References

- [1] Trail of Bits, “How to Spot Good Fuzzing Research,” Trail of Bits Blog, 05-Oct- 2018. [Online]. Available: <https://blog.trailofbits.com/2018/10/05/how-to-spot-good-fuzzing-research/>. [Accessed: 21-Jun-2019].
- [2] J. Choi, J. Jang, C. Han, and S. K. Cha, “Grey-box Concolic Testing on Binary Code,” in Proceedings of the 41st International Conference on Software Engineering, Piscataway, NJ, USA, 2019, pp. 736–747.
- [3] “Fuzzing Unit Tests with DeepState and Eclipser,” Trail of Bits Blog, 31-May-2019. [Online]. Available: <https://blog.trailofbits.com/2019/05/31/fuzzing-unit-tests-with-deepstate-and-eclipser/>. [Accessed: 21-Jun-2019].
- [4] Batelle, AFL Unicorn. Batelle, 2018. [Online] Available: <https://github.com/Battelle/afl-unicorn>
- [5] Google Project Zero, WinAFL. 2016. [Online] Available: <https://github.com/googleprojectzero/winafl>
- [6] aflgo, AFLGo. 2017 [Online]. Available: <https://github.com/aflgo/aflgo>
- [7] SoftSec-KAIST, Eclipser. 2019. [Online]. Available: <https://github.com/SoftSec-KAIST/Eclipser>
- [8] P. Chen, Hao Chen, Angora. 2019. [Online]. Available: <https://github.com/AngoraFuzzer/Angora>
- [9] Trail of Bits, Deep State. 2019. [Online]. Available: <https://github.com/trailofbits/deepstate>
- [10] M. Gaasedelen, Lighthouse. 2017. [Online]. Available: <https://github.com/gaasedelen/lighthouse>
- [11] M. Zalewski, “Technical ‘whitepaper’ for afl-fuzz.” [Online]. Available: [http://lcamtuf.coredump.cx/afl/technical\\_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt). [Accessed: 21-Jun-2019]
- [12] N. Voss, “afl-unicorn: Fuzzing Arbitrary Binary Code,” Hacker Noon, 31-Oct-2017. [Online]. Available: <https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf>. [Accessed: 21-Jun-2019].
- [13] N. Voss, “afl-unicorn: Part 2 — Fuzzing the ‘Unfuzzable,’” Hacker Noon, 15-Nov- 2017. [Online]. Available: <https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5>. [Accessed: 21-Jun-2019].
- [14] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed Greybox Fuzzing,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS ’17, Dallas, Texas, USA, 2017, pp. 2329–2344.
- [15] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based Greybox Fuzzing as Markov Chain,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16, Vienna, Austria, 2016, pp. 1032–1043.

- [16] M. Böhme, AFLFast. 2016. [Online]. Available: <https://github.com/mboehme/aflfast>
- [17] Van Hauser, AFLPlusPlus. 2019. [Online]. Available: <https://github.com/vanhauser-thc/AFLplusplus>
- [18] Google Project Zero, “WinAFL Dynamorio Instrumentation mode,” 21-Jun-2019. [Online]. Available: [https://github.com/googleprojectzero/winafl/blob/master/readme\\_dr.md](https://github.com/googleprojectzero/winafl/blob/master/readme_dr.md). [Accessed: 21-Jun-2019].
- [19] Google Project Zero, “WinAFL Intel PT Mode,” 21-Jun-2019. [Online]. Available: [https://github.com/googleprojectzero/winafl/blob/master/readme\\_pt.md](https://github.com/googleprojectzero/winafl/blob/master/readme_pt.md). [Accessed: 21-Jun-2019].
- [20] Google Project Zero, “WinAFL Statically Instrument A Binary Via syzygy,” 21-Jun-2019. [Online]. Available: [https://github.com/googleprojectzero/winafl/blob/master/readme\\_syzygy.md](https://github.com/googleprojectzero/winafl/blob/master/readme_syzygy.md). [Accessed: 21-Jun-2019].
- [21] LLVM Project, “libFuzzer – a library for coverage-guided fuzz testing,” LLVM Compiler Infrastructure. [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>. [Accessed: 21-Jun-2019].
- [22] Trail of Bits, “Using libFuzzer with McSema,” 21-Jun-2019. [Online]. Available: <https://github.com/trailofbits/mcsema>. [Accessed: 21-Jun-2019].
- [23] N. Stephens et al., “Driller: Augmenting Fuzzing Through Selective Symbolic Execution,” in Proceedings 2016 Network and Distributed System Security Symposium, San Diego, CA, 2016.
- [24] P. Chen and H. Chen, “Angora: Efficient Fuzzing by Principled Search,” arXiv:1803.01307 [cs], Mar. 2018.
- [25] Google Project Zero, Fuzzilli. 2019. [Online]. Available: <https://github.com/googleprojectzero/fuzzilli>
- [26] S. Groß, “FuzzIL: Coverage Guided Fuzzing for JavaScript Engines,” Karlsruhe Institute of Technology (KIT), 2018
- [27] S. Groß, “Fuzzilli (Guided-)fuzzing for JavaScript engines,” presented at the Offensive Con 2019.
- [28] H. Han, D. Oh, and S. K. Cha, “CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines,” in Proceedings 2019 Network and Distributed System Security Symposium, San Diego, CA, 2019.
- [29] SoftSec-KAIST, Code Alchemist. 2019. [Online]. Available: <https://github.com/SoftSec-KAIST/CodeAlchemist>
- [30] Hex-Rays, “About Us,” Hex-Rays. [Online]. Available: <https://www.hex-rays.com/about.shtml>. [Accessed: 21-Jun-2019].
- [31] Hex-Rays, “IDA Pro: What’s new in 7.1,” Hex-Rays. [Online]. Available: <https://www.Hex-Rays.com/products/ida/7.1/index.shtml>. [Accessed: 21-Jun-2019].

- [32] Hex-Rays, "IDA Pro: What's new in 7.2," Hex-Rays. [Online]. Available: <https://www.Hex-Rays.com/products/ida/7.2/index.shtml>. [Accessed: 21-Jun-2019].
- [33] Hex-Rays, "IDA Pro: About," Hex-Rays. [Online]. Available: <https://www.hex-rays.com/products/ida/>. [Accessed: 21-Jun-2019].
- [34] National Security Agency, "Ghidra," 22-Jun-2019. [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra>. [Accessed: 21-Jun-2019].
- [35] Vector 35, LLC, "Binary Ninja User Documentation." [Online]. Available: <https://docs.binary.ninja/>. [Accessed: 21-Jun-2019].
- [36] A. Bulazel, "Three Heads Are Better Than One: Mastering NSA's Ghidra Reverse Engineering Tool," 20-Jun-2019.
- [37] A. Bulazel, "Working With Ghidra's P-Code To Identify Vulnerable Function Calls." [Online]. Available: </blog/2019/05/pcode/>. [Accessed: 21-Jun-2019].
- [38] Vector 35, LLC, "Binary Ninja Intermediate Language Series, Part 1: Low Level IL." [Online]. Available: <https://docs.binary.ninja/dev/bnil-llil/index.html>. [Accessed: 21-Jun-2019].
- [39] J. Watson, "Breaking Down Binary Ninja's Low Level IL," Trail of Bits Blog, 31- Jan-2017. [Online]. Available: <https://blog.trailofbits.com/2017/01/31/breaking-down-binary-ninjas-low-level-il/>. [Accessed: 21-Jun-2019].
- [40] E. Bauman, Z. Lin, and K. W. Hamlen, Multiverse. 2019. [Online] Available: <https://github.com/utds3lab/multiverse>
- [41] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics," in Proceedings 2018 Network and Distributed System Security Symposium, San Diego, CA, 2018.
- [42] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in Proceedings of the 24th USENIX Conference on Security Symposium, Berkeley, CA, USA, 2015, pp. 627–642.
- [43] R. Wang et al., "Ramblr: Making Reassembly Great Again," in Proceedings 2017 Network and Distributed System Security Symposium, San Diego, CA, 2017.
- [44] A. Flores-Montoya and E. Schulte, "Datalog Disassembly," arXiv:1906.03969 [cs], Jun. 2019.
- [45] Gramma Tech, ddisasm. GrammaTech, 2019.
- [46] Quarkslab, "LIEF Documentation." [Online]. Available: <https://lief.quarkslab.com/doc/stable/index.html>. [Accessed: 22-Jun-2019].
- [47] T. Romain, "LIEF - Library to Instrument Executable Formats." [Online]. Available: <https://blog.quarkslab.com/lief-library-to-instrument-executable-formats.html>. [Accessed: 22-Jun-2019].
- [48] Quarkslab, "Introduction — LIEF 0.9.0 documentation," LIEF. [Online]. Available: <https://lief.quarkslab.com/doc/stable/Intro.html>. [Accessed: 22-Jun-2019].

- [49]Quarkslab, “01 - Parse and manipulate formats — LIEF 0.9.0 documentation,” LIEF. [Online]. Available:[https://lief.quarkslab.com/doc/stable/tutorials/01\\_play\\_with\\_formats.html](https://lief.quarkslab.com/doc/stable/tutorials/01_play_with_formats.html). [Accessed: 22-Jun-2019].
- [50]M. Gaasedelen, “What’s New in Lighthouse v0.7,” RET2 Systems Blog, 07-Dec- 2017. [Online]. Available: <https://blog.ret2.io/2017/12/07/lighthouse-v0.7/>. [Accessed: 21-Jun-2019].
- [51]M. Gaasedelen, “What’s New in Lighthouse v0.8 | RET2 Systems Blog.” [Online]. Available: <https://blog.ret2.io/2018/10/10/lighthouse-v0.8/>. [Accessed: 21-Jun-2019].
- [52]O. A. V. Ravnås, “Frida,” Frida • A world-class dynamic instrumentation framework. [Online]. Available: <https://www.frida.re/>. [Accessed: 22-Jun-2019].
- [53]O. A. V. Ravnås, “Frida Documentation,” Frida Documentation, 20-Jun-2019. [Online]. Available: <https://www.frida.re/docs/home/>. [Accessed: 22-Jun-2019].
- [54]O. A. V. Ravnås, “Modes of Operation,” Frida, 20-Jun-2019. [Online]. Available: <https://www.frida.re/docs/modes/>. [Accessed: 22-Jun-2019].
- [55]“rr: lightweight recording & deterministic debugging.” [Online]. Available: <https://rr-project.org/>. [Accessed: 22-Jun-2019].
- [56]R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, “Engineering Record And Replay For Deployability: Extended Technical Report,” arXiv:1705.05937 [cs], May 2017.
- [57]W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, “FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities,” in USENIX Security Symposium, 2018.
- [58]Sean Heelan, “Automation in Exploit Generation with Exploit Templates,” Sean Heelan’s Blog, 05-Mar-2019.
- [59] W. Wu, Yueqi Chen, Jun Xu, Xinyu Xin, Xiaorui Gong, and Wei Zou, Linux\_kernel\_exploits. 2019. [Online]. Available: [https://github.com/ww9210/Linux\\_kernel\\_exploits](https://github.com/ww9210/Linux_kernel_exploits)
- [60]angr, angr. 2013. [Online]. Available: <http://angr.io/>
- [61]Shellphish, Driller. 2019. [Online]. Available: <https://github.com/shellphish/driller>
- [62]T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing Symbolic Execution with Veritesting,” p. 12.
- [63]Shellphish, “symbion: fusing concrete and symbolic execution,” angr Blog. [Online]. Available: [http://angr.io/blog/angr\\_symbion/](http://angr.io/blog/angr_symbion/). [Accessed: 22-Jun-2019].
- [64]Mark Mossberg, “Manticore: Symbolic execution for humans,” Trail of Bits Blog, 27-Apr-2017.
- [65]Trail of Bits, Manticore. Trail of Bits, 2019. [Online]. Available: <https://github.com/trailofbits/manticore>



- [66]Trail of Bits, Manticore Wiki. Trail of Bits, 2019. [Online]. Available: <https://github.com/trailofbits/manticore/wiki>
- [67]I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing,” in Proceedings of the 27th USENIX Conference on Security Symposium, Berkeley, CA, USA, 2018, pp. 745–761.
- [68]sslab-gatech, QSYM , sslab-gatech, 2019. [Online]. Available: <https://github.com/sslab-gatech/qsym>
- [69]season-lab, MemSight: Rethinking Pointer Reasoning in Symbolic Execution (ASE 2017). SEASON Lab, 2019. [Online]. Available: <https://github.com/season-lab/memsight>
- [70]R. Gawlik and T. Holz, “SoK: Make JIT-spray Great Again,” in Proceedings of the 12th USENIX Conference on Offensive Technologies, Berkeley, CA, USA, 2018, pp. 10–10.
- [71]K. Z. Snow, F. Monroe, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,” in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013, pp. 574–588.
- [72]saelo, “Compile Your Own Type Confusions - Exploiting Logic Bugs in JavaScript JIT Engines,” Phrack.
- [73]A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking Blind,” in 2014 IEEE Symposium on Security and Privacy, 2014, pp. 227–242.
- [74]“Blind Return Oriented Programming (BROP).” [Online]. Available: <http://www.scs.stanford.edu/brop/>. [Accessed: 22-Jun-2019].
- [75]E. Bosman and H. Bos, “Framing Signals - A Return to Portable Shellcode,” in 2014 IEEE Symposium on Security and Privacy, 2014, pp. 243–258.
- [76]Colas Le Guernic, François Khourbiga. Taint-Based Return Oriented Programming. STIC 2018 - Symposium sur la sécurité des technologies de l’information et des communications, 2018, Rennes, France, pp.1-30.
- [77]Recon, “lecture: Taint-based return oriented programming.” [Online]. Available: <https://recon.cx/2018/montreal/schedule/events/129.html>. [Accessed: 22-Jun-2019].
- [78]S. Heelan, T. Melham, and D. Kroening, “Automatic Heap Layout Manipulation for Exploitation,” in Proceedings of the 27th USENIX Conference on Security Symposium, Berkeley, CA, USA, 2018, pp. 763–779.
- [79]D. Gruss et al., “Page Cache Attacks,” arXiv:1901.01161 [cs], Jan. 2019.
- [80]S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, “DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries,” in USENIX Security Symposium, 2018.
- [81]“New RIDL and Fallout Attacks Impact All Modern Intel CPUs,” BleepingComputer.

- [82] 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.
- [83] George Klees et al., “Evaluating Fuzz Testing,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), 2018.
- [84] Yan Wang et al., “Revery: From Proof-of-Concept to Exploitable,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), 2018.