AFRL-RI-RS-TM-2020-001
VERSION 3 of 4

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 3 OF 4

TWO SIX LABS

JANUARY 2021

TECHNICAL MEMORANDUM

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

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

NOTICE AND SIGNATURE PAGE

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

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

AFRL-RI-RS-TM-2020-001 Version 3 of 4 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.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JANUARY 2021	TECHNICAL MEMORANDUM	SEP 2019 - JAN 2020

4. TITLE AND SUBTITLE

Edge of the Art in Vulnerability Research
Version 3 of 4

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

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Scott Tenaglia
Perri Adams

5d. PROJECT NUMBER
CHES

5e. TASK NUMBER
S4

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Two Six Labs
901 N Stuart Street, Suite 1000
Arlington, VA 22203

8. PERFORMING ORGANIZATION REPORT NUMBER

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

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

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

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER

AFRL-RI-RS-TM-2020-001 Version 3 of 4

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 32474
Date Cleared: April 8, 2020

13. SUPPLEMENTARY NOTES

14. ABSTRACT

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

15. SUBJECT TERMS

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

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**AMANDA OZANAM**
U	U	U	UU	116	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

1 INTRODUCTION 5
2 SCOPE OF THE DOCUMENT 6
2.1 TOOLS CRITERIA 6
2.2 TECHNIQUE CRITERIA 6
2.3 TOOL AND TECHNIQUE CATEGORIES 7
3 STATIC ANALYSIS 7
3.1 TECHNICAL OVERVIEW 7
3.2 DECOMPILATION FRAMEWORKS 15
3.2.1 Binary Ninja - High Level IL 15
3.2.2 IDA Pro - Updates 23
3.2.3 Ghidra - Updates 24
3.3 BINARY DIFFERENTIATION 25
3.3.1 Hashashin 25
3.3.2 DeepBinDiff 27
3.4 KERNEL STATIC ANALYSIS 29
3.4.1 Crix 29
3.5 TRENDS 33
4 DYNAMIC ANALYSIS 33
4.1 TECHNICAL OVERVIEW 33
4.2 DEBUGGERS 37
4.2.1 Windbg Preview - Time Travel Debugging 37
4.2.2 Binary Ninja Debugger Plugin 43
4.2.3 Plutonium-dbg 46
4.3 COVERAGE ANALYSIS 49
4.3.1 Lighthouse - Version 9.0 Release 49
4.3.1.1 Updates 49
4.3.2 bcov 50
4.4 FRIDA 12.9 (STALKER UPDATE) 54
4.5 TRENDS 56
5 FUZZING 56
5.1 TECHNICAL OVERVIEW 56
5.2 MUTATION-BASED FUZZING 60
5.2.1 TortoiseFuzz 60
5.2.2 IJON 63
5.3 GENERATION-BASED FUZZING 65
5.3.1 Nautilus 65
5.4 PLATFORM SPECIFIC FUZZING 72
5.4.1 HFL 72
5.4.2 Hypercube 77
5.4.3 Syzkaller 79
5.5 TRENDS 82

6 EXPLOITATION 83
6.1 TECHNICAL OVERVIEW 83
6.2 DATA-ORIENTED ATTACKS 86
6.2.1 DOP 86
6.2.2 BOPC 90
6.3 HEAP 93
6.3.1 Gollum 93
6.4 KERNEL EXPLOITATION 97
6.4.1 SLAKE 101
6.5 TRENDS 105
7 CONCLUSION 105
8 BIBLIOGRAPHY 106

List of Figures

Figure 1: Tradeoffs of IRs, Pt. 1 [21, p. 29] 10
Figure 2: Tradeoffs of IRs, Pt. 2 [21, p. 30] 11
Figure 3: Binary Ninja's Intermediate Representation Family [21] [3] 16
Figure 4: Binary Ninja HLIL of Code 1 18
Figure 5: Binary Ninja HLIL SSA Form of Code 1 19
Figure 6: Binary Ninja HLIL of Code 1 20
Figure 7: IDA Decompilation of Code 1 21
Figure 8: Ghidra decompilation of Code 1 22
Figure 9 Folder View of Program Imports 24
Figure 10 - Benchmarking Hashashin on PDF parsers [44] 26
Figure 11: Stages of Operation of DeepBinDiff [46, p. 4]. 28
Figure 12: Missing-check error in the Linux kernel found by Crix 30
Figure 13: Overview of the Crix operation [52, p. 5] 31
Figure 14: Sample Output for CRIX 32
Figure 15: Attaching to an already running Calculator application with TTD enabled 38
Figure 16: Launching Calculator application directly from WinDbg Preview with TTD enabled 39
Figure 17: HelloWorld program, hw.exe, running with TTD 39
Figure 18: TTD execution of hw.exe stopped at the read function call 40
Figure 19: TTD execution of hw.exe reaches third memory access 41
Figure 20: TTD execution of hw.exe hits exception 42
Figure 21: Implementation diagram of BNDP with tested targets 44
Figure 22: BNDP debugging a macOS binary [75] 44
Figure 23: Debugging jumptable in macOS binary before new jump location is dynamically discovered [74] 45
Figure 24: Debugging jump table in a macOS binary after a new jump location is dynamically discovered [74] 45
Figure 25: Using plutonium-dbg 48
Figure 26 New theming support in Binary Ninja and IDA Pro 49
Figure 27 Absolute address format coverage file 50
Figure 28 Modoff format coverage file 50
Figure 29 Sliced microexecution hypotheses [80] 52
Figure 30 bcov workflow [80] 53
Figure 31 bcov compared to other coverage tools [80] 53
Figure 32 Stalker Instrumentation [86] 54
Figure 33 Stalker Example [87] 55
Figure 34: Diagram of AFL workflow [91, p. 2] 58
Figure 35: Example of a grammar used with a grammar-based fuzzer [94] 59
Figure 36: Algorithm of AFL with TortoiseFuzz modifications in gray [91, p. 6] 62
Figure 37: Example annotations of IJON_DISABLE and IJON_ENABLE 64
Figure 38: IJON annotate maze game 64
Figure 39: Exploration comparison of AFL and AFL-IJON on a LIBTPMS (Trusted Platform Module) [104] 65
Figure 40: Nautilus Overview [96, p. 4] 66

Figure 41: Example of Recursive Minimization [96, p. 5] 68
Figure 42: Example of Random Recursive Mutation [96, p. 5] 69
Figure 43: Nautilus Fuzzer [96, p. 7] 70
Figure 44: Nautilus Terminal Stats Screen 71
Figure 45: HFL system call dependency interference system [100, p. 9] 74
Figure 46: "Workflow of nested syscall argument retrieval [100, p. 9]" 75
Figure 47: "Comparison of bug-finding time for 13 known crashes [100, p. 10] " 76
Figure 48: "Coverage results during a [50-hour] run [100, p. 12] " 76
Figure 49: HYPER-CUBE Architecture from paper 78
Figure 50: syzkaller System Architecture 80
Figure 51 Example syscall description program created by syzkaller 80
Figure 52 Syzkaller web GUI 81
Figure 53- MINDOP Language [122, p. 4] 87
Figure 54 - Diagram of gadget and dispatcher interaction [122, p. 5] 88
Figure 55 - Example of gadgets in code [122, p. 3] 88
Figure 56 - Behavior simulated by code in Figure 55 [122, p. 3] 89
Figure 57: Diagram of a BOP Gadget [129, p. 4] 92
Figure 58: High-level overview of BOPC [129, p. 3] 92
Figure 60 - Workflow diagram for Gollum [131, p. 3] 94
Figure 61 - Example code for stages of Gollum [131, p. 4] 96
Figure 62: KEPLER's design [132, p. 1193] 98
Figure 63: Example of a kernel exploit that could be exploited by KEPLER [132, p. 1191] 100
Figure 64: Example kernel exploitation approaches given in [126] 102
Figure 65: Example of SLAKE manipulating the SLAB [126] 104
Figure 66: SLAKE reorganization algorithm 104

1 Introduction

The DARPA 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 fast-moving fields 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 purpose of this second *Edge of the Art (EotA)* report is to document those things that have come into existence (or significantly matured) since the last report.

The EotA reports are produced using an "*aggregate and filter*" approach. The CHECKMATE team constantly monitors many different sources to *aggregate* all known and emerging tools and techniques. This information is then *filtered* into what the CHECKMATE team considers worth reporting. The definition of the *"edge"* is governed by the filter criteria, which differ across tools and techniques. It is anticipated that these criteria, and therefore the definition of *"edge,"* will evolve over the life of the CHESS program.

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 2015 or later). Most of the tools were released in or after 2018, and most of those were released in or after 2019. Those released earlier are included because they have significantly matured since their initial release and now contain notable features.

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 is dependent on the category of tool, and each section of this report starts with an introduction that lays out its specific considerations.

Theory and Approach – Tools which offer novel ideas, approaches, or new research are important even when the tools have poor implementations or do not necessarily outperform the current state-of-the-art.

Usability – In contrast with *Theory and Approach*, Usability considers tools which may not represent groundbreaking research, but enable the user to harness existing capabilities more effectively.

Current State-of-the-Art – The line between edge-of-the-art and state-of- the-art is hazy. There is rarely a single moment where a tool or technique definitively transitions from one category to another. In some cases, including a tool that one might consider state-of-the-art is necessary to compare to the edge-of-the-art. In other cases, the tool has new capabilities which keep it on the edge-of-the-art.

2.2 Technique Criteria

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

2.3 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 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 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 forthe tool categorization used in this report.

This version of the Edge of Art report describes tools in the categories of *Static Analysis, Dynamic Analysis*, *Fuzzing*, and *Exploitation*.

3 Static Analysis

3.1 Technical Overview

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

Disassembly

Relevant EotA Tools: IDA Pro, Ghidra, Binary Ninja, Miasm, BAP 2.0

An assembler converts a program from assembly language to machine code, and a disassembler performs the reverse: it converts machine code to assembly language. Since there is often a one- to-one correspondence between machine instructions and assembly instructions, this translation is much less complicated than decompilation. However, disassembly can pose challenges, especially

with architectures like x86 which have variable length instructions. When overlapping sequences of bytes could themselves be valid instructions, instruction cannot be disassembled at random. Several approaches to disassembly address this challenge, including linear sweep (which disassembles instructions in the order they appear starting from the first instruction) andrecursive descent (which disassembles instructions in the order of their control flow) [1, p. 2]. Many popular disassemblers including IDA Pro [2] and Binary Ninja [3] use the latter technique.

Disassembling machine code is often the first step in binary analysis, and most binary analysis tools rely on some type of disassembler. There are currently a variety of disassemblers available, ranging from simple command line utilities to proprietary platforms with capabilities far beyond basic disassembly. A simple example is objdump [4], a standard tool on Linux operating systems which will output the disassembly of a given binary. Tools like debuggers often rely on more sophisticated disassembly frameworks like Capstone [5] which has features complementary to its core disassembler and is designed to be used via an API. Capstone is a dependency of many tools in this report, such as angr [6], Qiling [7] and Frida [8]. The disassembly framework Miasm [9] (discussed in the second version of this report) can be used similarly to Capstone.

In contrast to frameworks, disassembly platforms are designed primarily for humans to analyze disassembled code through a graphic user interface (GUI). These are often sophisticated user applications which offer a significant range of features beyond disassembling code. For example, many of these applications have built-in APIs that can be used as frameworks for custom, automated analyses. Several of these tools were discussed in the first version of this report: IDA Pro [2], Ghidra [10] and Binary Ninja [3].

Reassembleable Disassembly

Relevant EotA Tools: DDisasm, Retrowrite, angr (Ramblr)

The disassembly techniques discussed to this point are only concerned with moving from machine code to assembly. However, *reassembly* (automatically reassembling disassembled code) has recently become an area of academic interest, in part to support static instrumentation. A 2015 paper, *Reassembleable Dissassembling* [11], claims that at the time "no existing tool is able to disassemble executable binaries into assembly code that can be correctly assembled back in a fully automated manner, even for simple programs. Actually, in many cases, the 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 [11, p. 1]." The paper presented a tool that could disassemble a binary using a set of rules that made the resulting disassembly relocatable, which they assert is the "key" to reassembling [11, p. 1]. Since 2015, this technique has been improved, notably by the creators of angr who built a reassembling tool called Ramblr [12]. More recently, the tool DDisasm
[13] was introduced and was discussed in the first version of this report.

Static Binary Rewriting and Static Instrumentation

Relevant EotA Tools: Retrowrite, LIEF

Binary rewriting modifies a binary executable without needing to change the source code and recompile. One use case is for binary instrumentation, which is often thought of as a dynamic technique. While many dynamic binary instrumentation (DBI) techniques exist, there are also methods for statically instrumenting binaries. Many of these rely on reassembling or relinking the binary. Retrowrite [1], a tool designed to statically instrument binaries for dynamic analysis like fuzzing and memory checking, also uses a reassembleable disassembly technique that builds on previous research in reassembleable disassembly and static binary rewriting. The tool LIEF [14], discussed in the first version of this report, allows the user to statically hook a binary, or statically modify it in a variety of other ways.

Intermediate Representation (IR)

Relevant EotA Tools: IDA Pro, Ghidra, Binary Ninja, Miasm, BAP 2.0, GTIRB, Fuzzilli

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

Intermediate Representations are commonly used in compilers. A familiar example is LLVM [15], the IR used in the Clang compiler [16]. LLVM is helpful as an example not just because it is well known, but because it demonstrates the range of features a well-designed IR can offer. The instruction set and type system for LLVM is language independent, which means there are no high- level types and attributes. This allows LLVM to be ported to many architectures. While the type system is low-level, providing type information enables LLVM to be optimized through various analyses [17]. Unlike machine code, LLVM is designed to be human readable [17].

LLVM uses a technique called Single Static Assignment (SSA), which means each variable is assigned a value only once. SSA enables analysis such as variable recovery because it inherently maps one instruction to many and generates output not intended for human consumption.

These traits are not specific to LLVM but are attributes of many IRs discussed in this report. Clang's compiler works by translating source code languages to LLVM, performing optimizations, and then translating the LLVM bitcode to a specific architecture [18]. The Ghidra's decompiler does something similar but in reverse: a binary program is first lifted (converted to a higher-level representation) to an IR called P-Code [19], on which Ghidra can perform analyses and then decompile by converting the program to pseudo source code. Therefore, Ghidra can decompile anything it can lift to P-Code, because decompilation is performed on a language agnostic IR and not the original machine language [20].

Ghidra uses SSA in its decompilation, but unlike LLVM, P-Code is not in SSA form by default [20]. Other IRs also have SSA and non-SSA forms. For example, Binary Ninja's IRs offer the ability to toggle between non-SSA and SSA form [3]. SSA demonstrates one of the trade-offs that inform IR design. The developers of Binary Ninja created the charts in Figure 1 and Figure 2 to show the tension between different features of IRs.

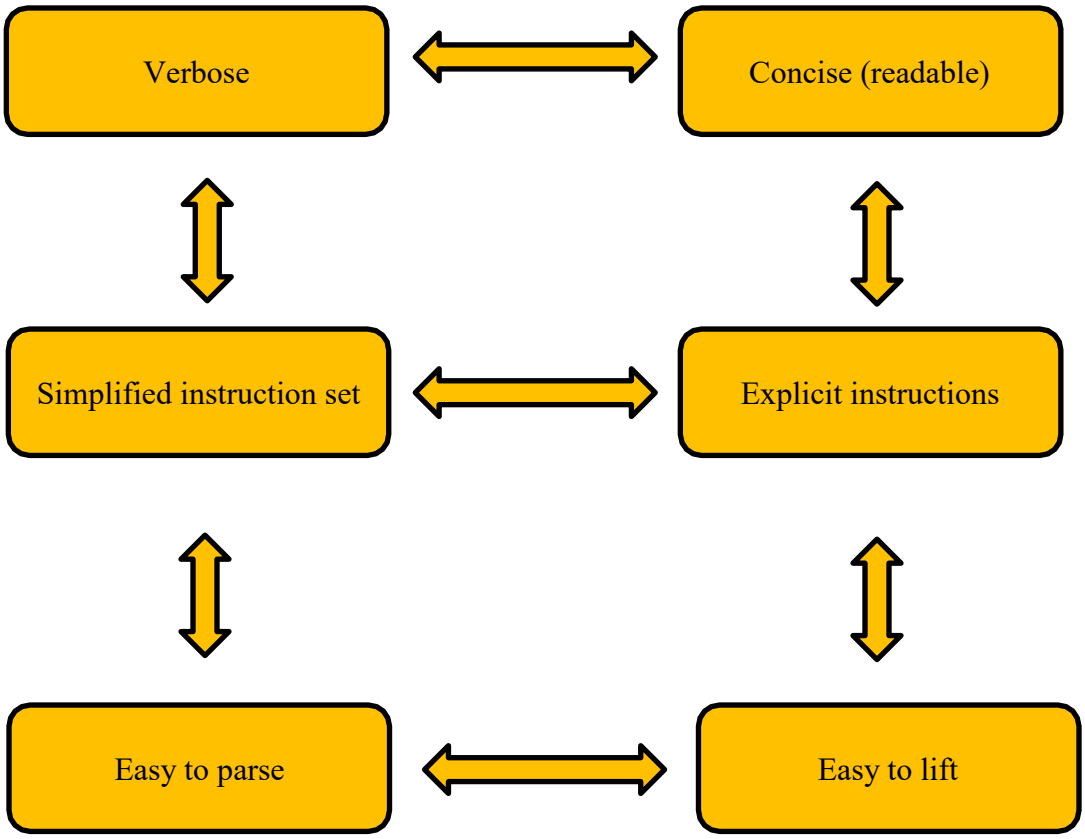


Figure 1: Tradeoffs of IRs, Pt. 1 [21, p. 29]

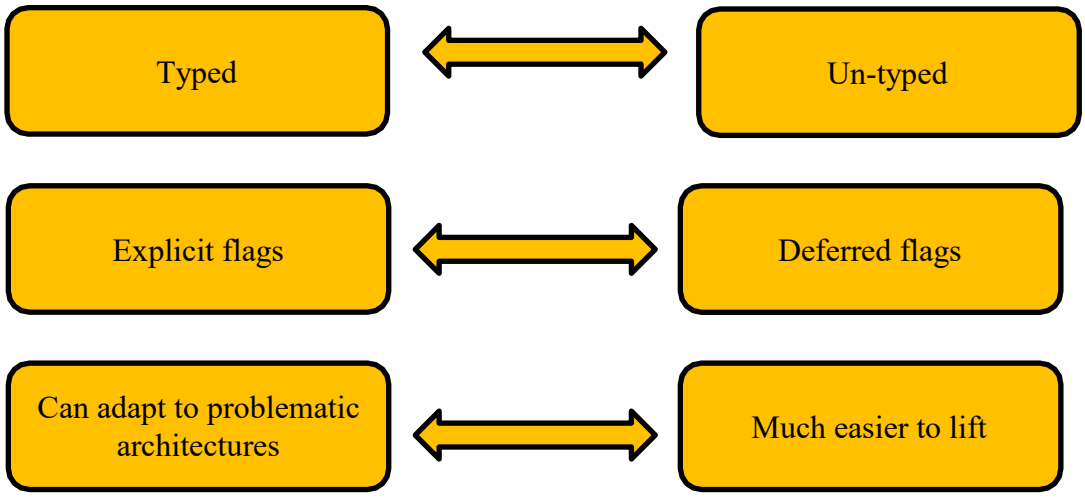


Figure 2: Tradeoffs of IRs, Pt. 2 [21, p. 30]

Intermediate Representations, each with their own mix of features, are used extensively throughout the tools in this report. Decompilers such as IDA Pro, Ghidra and Binary Ninja (which has developed a decompiler that is not yet released) each have their own IRs. These are used not just for decompilation, but also exposed via APIs that allow the user to utilize it for their own analyses. IDA Pro only recently documented their API [22], whereas the public release of Ghidra's P-Code included an API. However, out of these three platforms, Binary Ninja's IRs are designed with the greatest degree of user capability. They offer three levels of IRs each with an optional SSA-form and a feature-heavy API [23]. Their third level is decompilation.

Other IR frameworks discussed in the second version of this report can be used in the same manner, but each offer their own set of features. Binary Analysis Platform (BAP) [24] is a framework designed for program analysis and built around the BAP Intermediate Language (BIL), which has a formally defined grammar [25]. Miasm has an expression-based IR that facilitates tracking memory and registry values [26]. Miasm also has a JIT engine for emulation and has built in support for symbolic execution [27].

Certain IRs are tailored for specific use cases. For example, Fuzzilli, a fuzzer which targets JavaScript JIT engines, uses a custom IR called FuzzIL [28]. Seeds are constructed and mutated in FuzzIL then translated into JavaScript before being fed into the engine [28]. This approach has the benefit of being able to theoretically explore all possible patterns given enough computing power, unlike a JIT fuzzer working from hardcoded JavaScript samples.

By contrast some tools in this report use existing IRs rather than creating their own. The symbolic execution tool angr uses Vex, which is the IR implemented by the memory debugger Valgrind [29]. WinAFL, a version of the AFL fuzzer for the Windows operating system, uses DynamoRIO, a dynamic binary instrumentation engine with its own IR [30].

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

satisfies differing use cases, but also results in compatibility issues. GTIRB [31], which is discussed in the previous version of this report, is an IR designed to convert between different IRs. It is also the IR used in DDisasm.

Decompilation

Relevant EotA Tools: IDA Pro, Ghidra, Binary Ninja

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

Whereas compiler theory has been a popular area of computer science for decades, its reverse has received far less attention. In 1994 Christina Ciafuentes published her PhD thesis on the subject, *Reverse Compilation Techniques* [32]. This work went on to inform the development of multiple decompilers, including Hex-Rays, the decompiler of choice for over a decade. This tool is part of IDA Pro, a disassembler which has been commercially available since 1996. However, Hex-Rays was not released until 2005 [33]. Until recently, it was one of the few decompilers available, and the most technically sophisticated.

As of 2019, the United States National Security Agency (NSA) released Ghidra, a disassembler and decompiler with comparable performance to IDA Pro [34]. In March 2020, Vector35 released a decompiler for their tool, Binary Ninja. Binary Ninja not only exposes its IRs to the user, but makes them a fundamental part of its design, with this new decompilation acting as a third layer in their three-tiered IR system. Their decompilation is available in both SSA and non-SSA form.

Each decompiler has its respective strengths and weaknesses. Although IDA Pro is now experiencing significant competition, it is still considered the most effective decompiler available in many cases. Ghidra and IDA Pro each have over a decade of development, whereas Binary Ninja is a newer tool that still has room to grow when compared to its more mature competitors. Ghidra has comparable performance to IDA Pro, but for certain constructs like jump tables and no return functions, IDA Pro clearly performs better than Ghidra.

In contrast, Binary Ninja arguably offers the best scripting capabilities, intermediate representations, and functionality for program analysis. Both Binary Ninja and IDA Pro make scripting with their decompiler and their intermediate representations far easier than IDA Pro. Binary Ninja exposes both a low-level IR and a medium-level IR (in both SSA and non-SSA form) in addition to their decompilation. Ghidra also makes its P-Code IR readily available to the user. In the case of IDA, although it has an IR, its less publicly accessible and easily scriptable. This divide is also reflected in these tools' respective methods of decompilation. Ghidra and Binary Ninja both decompile from their respective IRs in an architecture-independent manner, meaning that any architecture that can be lifted to their IRs and subsequently decompiled. Although there

is less publicly available information on the internals of the IDA Pro decompiler, it requires a separate decompiler to be purchased for each architecture.

Ghidra is a completely free open-source tool, whereas both Binary Ninja and IDA Pro are closed- source and cost different amounts of money. An IDA Pro license with decompilers for just x86 and x86-64 costs several times that of a Binary Ninja license (which can decompile any architecture which lifts to its IR).

Static Vulnerability Discovery

Relevant EotA Tools: Crix

There are a number of tools and techniques designed to statically discover vulnerabilities. Many target source code analysis, including tools such as Coverity [35], CodeSonar [36], and Semmle [37]. These use static analysis algorithms to find possible vulnerabilities and common vulnerability patterns in the code base. Additionally, there exist program analysis techniques designed to statically identify vulnerabilities in binary code, like graph-based vulnerability discovery and value-set analysis (VSA) [29, p. 5]. The tool Crix, which is discussed in this section, is a static analysis tool designed to find memory-check errors.

Static Program Analysis

Relevant EotA Tools: IDA Pro, Binary Ninja, Ghidra, angr, BAP 2.0, Retrowrite, etc.

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

- **Control Flow Recovery:** A binary program can be broken into basic blocks separated by branches: a basic block is a sequence of instructions that contains no jumps, except at the entry and exit. A control flow graph (CFG) models a program as a graph in which the basic blocks of the program are represented as nodes, and the jumps, or branches, are represented as edges. A CFG is instrumental to many forms of static program analysis and vulnerability discovery. Recovering it is done by disassembling the program and identifying the basic blocks and the jumps between them (both direct and indirect) [29, p. 4].

- **Variable and Type Information Recovery:** Variable and type information is used by the compiler but is not present in final binary executable form (unless the binary is compiled to explicitly include this information for debugging purposes). Therefore, it is often necessary to recover this information when analyzing a binary [1, p. 5]. One attribute of many IRs is that their lifters will recover variable and type information and

include it in the IR. This is also necessary for decomplication.

- **Function Identification:** Function information is also often left out of the final binary form of a computer program, and it is also necessary in various forms of analysis. Methods have been developed to identify distinct functions within a binary [1, p. 5].

- **Value Set Analysis (VSA):** VSA is a form of static analysis which attempts to track values and references throughout a binary [29, p. 5]. This analysis has a variety of uses, including identifying indirect jumps or finding vulnerabilities like out of bound accesses.

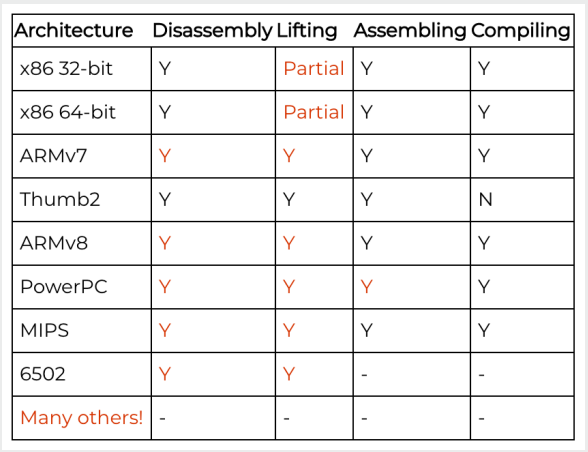
- **Graph-based vulnerability discovery:** This form applies graph analysis to a CFG to identify vulnerabilities [29, p. 5].

- **Symbolic Execution:** Symbolic execution replaces program inputs with symbolic values, and then symbolically executes over the program. Symbolic execution be done either statically or dynamically.

- **Abstract Interpretation, data-flow analysis, etc.:** There are many types of formal static analysis which apply mathematical approaches to program analysis. These include abstract interpretation and data-flow analysis. The tools BAP has implemented support these forms of analysis, including in a recent update [38].

3.2 Decompilation Frameworks

3.2.1 Binary Ninja - High Level IL

Reference Link	**https://binary.ninja/**
Target Type	Binary
Host Operating System	Linux; macOS; Windows
Target Operating System	**Windows:** PE (Portable Executable); etc. **macOS/iOS:** Mach-O; etc. **Linux:** ELF; etc. **Other:** COFF (Common Object File Format); Raw Binary; etc.
Host Architecture	x86 (32, 64)
Target Architecture	 Supported Architectures [35]
Initial Release	2016
License Type	Proprietary
Maintenance	Maintained by Vector 35

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!	-	-	-	-

Overview

Binary Ninja is built for contemporary reverse engineering and ease of use, while also incorporating complex program analysis techniques. It lacked a decompiler until March 2020, when Vector35 finally released one. Their decompiler is an extension of their three-tiered intermediate representation (IR) family and is termed HLIL. It is also accompanied by a HLIL in SSA form. The HLIL and its SSA form are deeply integrated with Binary Ninja's scripting engine, which facilitates development of automated analyses and plug-ins. [3] [21]

With the introduction of Ghidra in in March 2019, there are now three actively developed, high quality decompilers available. This represents a significant disruption to a market that has for years had few viable options for effective decompilation. These three reverse engineering platforms are, in some respects, complementary, each having their own benefits and drawbacks.

Intermediate Representations

One of the most useful capabilities of Binary Ninja are its intermediate languages. Binary Ninja offers an entire family of IRs, at low, medium and high levels. The user can easily switch between these IRs in both Linear and Graph view. In contrast with IDA Pro's IR, and to a lesser extent Ghidra's P-Code IR, the IRs designed for Binary Ninja are meant to be human-readable. Binary Ninja's IRs are also more sophisticated in that they are optionally SSA, have deferred flag calculation and the ability to transform assembly instructions to generic ones. These IRs, and their SSA forms, are easily available with Binary Ninja's verbose Python scripting engine.

Figure 3 shows a diagram of the Binary Ninja IRs in relation to one another.

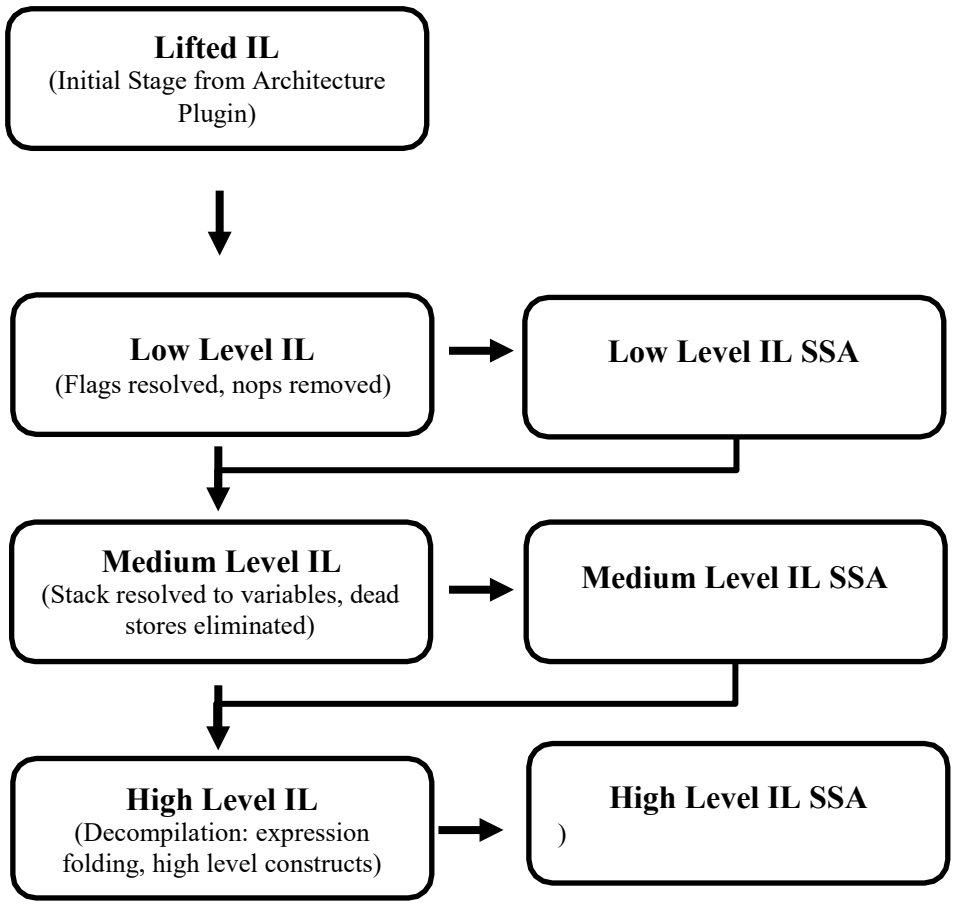


Figure 3: Binary Ninja's Intermediate Representation Family [21] [3]

HLIL

HLIL is the highest level of Binary Ninja's IRs and serves as its decompiler. As Ryan Stortz writes in [39],

“HLIL has aggressive dead code elimination, constant folding, switch recovery, and all the other things you’d expect from a decompiler, with one exception: Binary Ninja’s decompiler doesn’t target C.”

Unlike other decompilers Binary Ninja’s tiered IRs are the core of their decompilation. This makes it architecture independent because anything that can lift to the LLIL can then be decompiled. However, using heuristics can help create more readable decompilation.

Code 1 is an example function to be compiled and then decompiled to Binary Ninja’s HLIL.

```
1  int main()
2  {
3      pid_t child = fork();
4      char buffer[128] = {0};
5      int syscall = 0;
6      int status = 0;
7
8      if(child == 0)
9      {
10         prctl(PR_SET_PDEATHSIG, SIGHUP);
11         ptrace(PTRACE_TRACEME, 0, NULL, NULL);
12
13         /* this is all you need to worry about */
14         puts("just give me some shellcode, k");
15         gets(buffer);
16     }
17     else
18     {
19         /* mini exec() sandbox, you can ignore this */
20         while(1)
21         {
22             wait(&status);
23             if (WIFEXITED(status) || WIFSIGNALED(status)){
24                 puts("child is exiting...");
25                 break;
26             }
27
28             /* grab the syscall # */
29             syscall = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);
30
31             /* filter out syscall 11, exec */
32             if(syscall == 11)
33             {
34                 printf("no exec() for you\n");
35                 kill(child, SIGKILL);
36                 break;
37             }
38         }
39     }
40
41     return EXIT_SUCCESS;
42 }
43
```

Code 1: Example function source code

Figure 4 shows the HLIL for Code 1, displayed in Linear mode.

```
int32_t main(int32_t argc, char** argv, char** envp)

0804898f  int32_t ebp
0804898f  int32_t var_4 = ebp
08048992  int32_t edi
08048992  int32_t var_8 = edi
08048993  void* var_c = _GLOBAL_OFFSET_TABLE_
0804899d  void* eax = fork()
080489b7  void* edi_1 = &arg_10
080489bb  for (int32_t ecx = 0x20; ecx != 0; ecx = ecx - 1)
080489bb      *edi_1 = 0
080489bb      edi_1 = edi_1 + 4
080489c8  envp = 0
080489d0  if (eax == 0)
080489e9      prctl(1, 1, var_8, var_4)
08048a0d      ptrace(0, 0, 0, 0)
08048a19      puts(0x8048b84)  {"just give me some shellcode, k"}
08048a25      gets(&arg_10)
08048a36  else
08048a36      while (true)
08048a36          wait(&envp, var_c, var_8, var_4)
08048a50          if ((envp & 0x7f) != 0)
08048a6c              void* eax_7
08048a6c              eax_7:0.b = (envp & 0x7f) + 1:0.b s>> 1
08048a6e              if (eax_7:0.b s<= 0)
08048a80                  var_4 = 0
08048a88                  var_8 = 0x2c
08048a97                  var_c = eax
08048abf                  if (ptrace(3, var_c, 0x2c, 0) == 0xb)
08048abf                      puts(0x8048bb7)  {"no exec() for you"}
08048ad6                      kill(eax, 9)
08048adb                      break
08048ab6                  continue
08048a79          puts(0x8048ba3)  {"child is exiting..."}
08048a7e          break
08048aed  return 0
```

Figure 4: Binary Ninja HLIL of Code 1

Figure 5 shows the HLIL in SSA for Code 1, also displayed in Linear mode.

```
int32_t main(int32_t argc, char** argv, char** envp)

0804898f  int32_t ebp
0804898f  int32_t var_4#1 = ebp#0
08048992  int32_t edi
08048992  int32_t var_8#1 = edi#0
08048993  void* var_c#1 = _GLOBAL_OFFSET_TABLE_
0804899d  void* eax#1 = fork() @ mem#0 -> mem#1
080489b7  void* edi_1#1 = &arg_10
080489bb  for (int32_t ecx#1 = 0x20; ecx#2 = φ (ecx#1, ecx#3)
080489bb  edi_1#2 = φ (edi_1#1, edi_1#3)
080489bb  mem#2 = φ (mem#1, mem#3)
080489bb  , ecx#2 != 0; ecx#3 = ecx#2 - 1)
080489bb      *edi_1#2 @ mem#3 = 0 @ mem#2
080489bb      edi_1#3 = edi_1#2 + 4
080489c8  envp = 0
080489d0  if (eax#1 == 0)
080489e9      prctl(1, 1, var_8#1, var_4#1) @ mem#2 -> mem#4
08048a0d      ptrace(0, 0, 0, 0) @ mem#4 -> mem#5
08048a19      puts(0x8048b84) @ mem#5 -> mem#6  {"just give me some shellcode, k"}
08048a25      gets(&arg_10) @ mem#6 -> mem#7
08048a36  else
08048a36      while (var_c#2 = φ (var_c#1, var_c#3)
08048a36      var_8#2 = φ (var_8#1, var_8#3)
08048a36      var_4#2 = φ (var_4#1, var_4#3)
08048a36      eax_7#1 = φ (eax_7#0, eax_7#2)
08048a36      mem#8 = φ (mem#2, mem#10)
08048a36      , true)
08048a36          wait(&envp, var_c#2, var_8#2, var_4#2) @ mem#8 -> mem#9
08048a50          if ((envp & 0x7f) != 0)
08048a6c              void* eax_7
08048a6c              eax_7#2:0.b = (envp & 0x7f) + 1:0.b s>> 1
08048a6e              if (eax_7#2:0.b s<= 0)
08048a80                  var_4#3 = 0
08048a88                  var_8#3 = 0x2c
08048a97                  var_c#3 = eax#1
08048abf                  if (ptrace(3, var_c#3, 0x2c, 0) @ mem#9 -> mem#10 == 0xb)
08048abf                      puts(0x8048bb7) @ mem#10 -> mem#11  {"no exec() for you"}
08048ad6                      kill(eax#1, 9) @ mem#11 -> mem#12
08048adb                      break
08048ab6                  continue
08048a79          eax_7#3 = φ (eax_7#1, eax_7#2)
08048a79          puts(0x8048ba3) @ mem#9 -> mem#13  {"child is exiting..."}
08048a7e          break
08048aed  var_c#4 = φ (var_c#1, var_c#2, var_c#3)
08048aed  var_8#4 = φ (var_8#1, var_8#2, var_8#3)
08048aed  var_4#4 = φ (var_4#1, var_4#2, var_4#3)
08048aed  eax_7#4 = φ (eax_7#0, eax_7#2, eax_7#3)
08048aed  mem#14 = φ (mem#7, mem#12, mem#13)
08048aed  return 0
```

Figure 5: Binary Ninja HLIL SSA Form of Code 1

Figure 6 shows the HLIL for Code 1, displayed in Graph view.

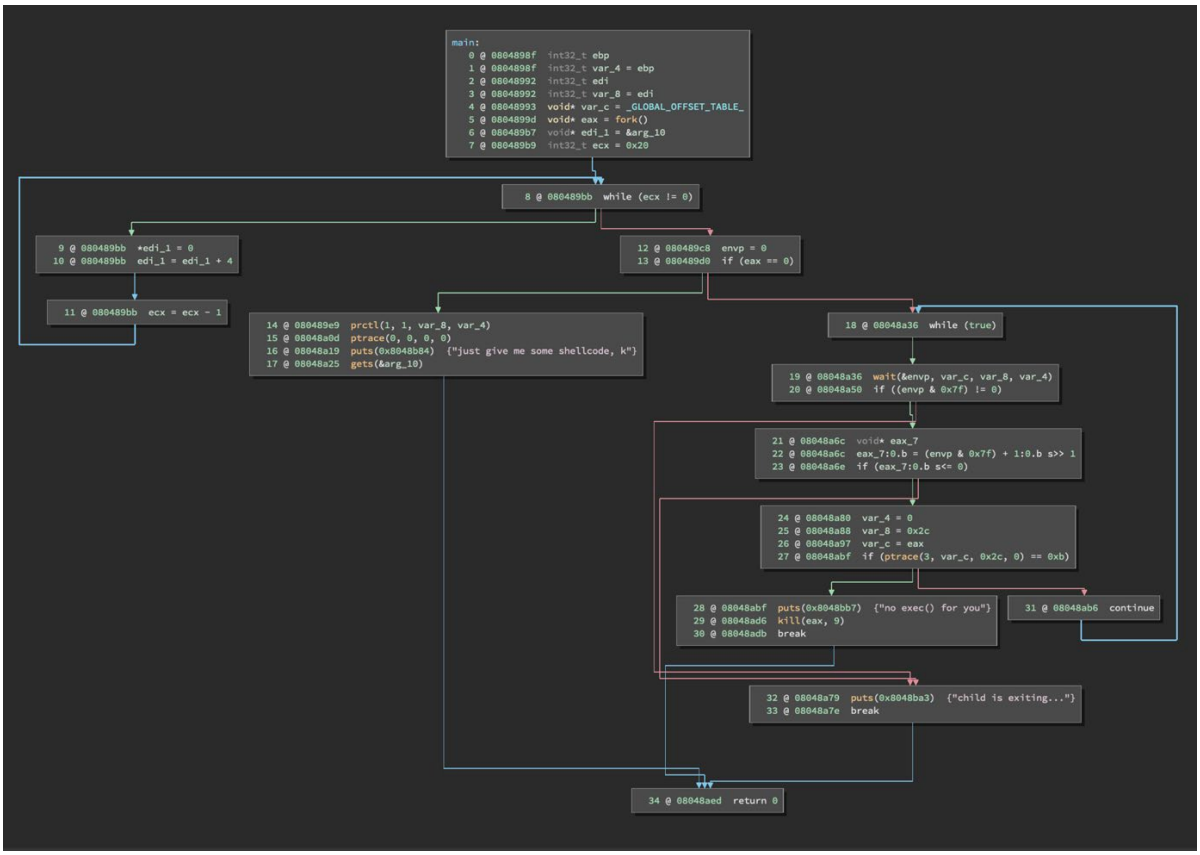


Figure 6: Binary Ninja HLIL of Code 1

Use Cases and Limitations

Although a side by side comparison of each tools' decompilation of only one test file is not an absolute measure of quality, it is helpful to understand the differences between the three tools. Figure 7 and Figure 8 show IDA Pro's and Ghidra's decompilation of Code 1, respectively.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int stat_loc; // [esp+1Ch] [ebp-9Ch]
  char s; // [esp+20h] [ebp-98h]
  int v6; // [esp+A0h] [ebp-18h]
  int v7; // [esp+A4h] [ebp-14h]
  int v8; // [esp+A8h] [ebp-10h]
  int v9; // [esp+ACh] [ebp-Ch]

  v9 = fork();
  memset(&s, 0, 0x80u);
  v8 = 0;
  stat_loc = 0;
  if ( v9 )
  {
    do
    {
      wait(&stat_loc);
      v6 = stat_loc;
      if ( !(stat_loc & 0x7F) || (v7 = stat_loc, (char)((char)((stat_loc & 0x7F) + 1) >> 1) > 0) )
      {
        puts("child is exiting...");
        return 0;
      }
      v8 = ptrace(PTRACE_PEEKUSER, v9, 44, 0);
    }
    while ( v8 != 11 );
    puts("no exec() for you");
    kill(v9, 9);
  }
  else
  {
    prctl(1, 1);
    ptrace(0, 0, 0, 0);
    puts("just give me some shellcode, k");
    gets(&s);
  }
  return 0;
}
```

Figure 7: IDA Decompilation of Code 1

Decompile: main - (lab3B)

```
1
2  undefined4 main(void)
3
4  {
5    int iVar1;
6    undefined4 *puVar2;
7    byte bVar3;
8    uint local_a4;
9    undefined4 local_a0 [32];
10   uint local_20;
11   uint local_1c;
12   long local_18;
13   __pid_t local_14;
14
15   bVar3 = 0;
16   local_14 = fork();
17   iVar1 = 0x20;
18   puVar2 = local_a0;
19   while (iVar1 != 0) {
20     iVar1 = iVar1 + -1;
21     *puVar2 = 0;
22     puVar2 = puVar2 + (uint)bVar3 * 0x3ffffffe + 1;
23   }
24   local_18 = 0;
25   local_a4 = 0;
26   if (local_14 == 0) {
27     prctl(1,1);
28     ptrace(PTRACE_TRACEME,0,0,0);
29     puts("just give me some shellcode, k");
30     gets((char *)local_a0);
31   }
32   else {
33     do {
34       wait(&local_a4);
35       local_20 = local_a4;
36       if (((local_a4 & 0x7f) == 0) ||
37          (local_1c = local_a4, '\0' < (char)(((byte)local_a4 & 0x7f) + 1) >> 1)) {
38         puts("child is exiting...");
39         return 0;
40       }
41       local_18 = ptrace(PTRACE_PEEKUSER,local_14,0x2c,0);
42     } while (local_18 != 0xb);
43     puts("no exec() for you");
44     kill(local_14,9);
45   }
46   return 0;
47 }
48
```

Figure 8: Ghidra decompilation of Code 1

Each decompiler has its own strengths and weaknesses. Although IDA Pro is now experiencing significant competition, for now it is still considered the most effective decompiler available. While Ghidra and IDA Pro have each been in development for over a decade, for certain constructs like jump tables and no return functions, IDA Pro clearly outperforms Ghidra. Binary Ninja is a newer tool that still has room to grow but has features that IDA Pro and Ghidra do not.

Binary Ninja arguably offers the best scripting capabilities, intermediate representations, and functionality for program analysis. Both Binary Ninja and IDA Pro make scripting with their decompiler and their intermediate representations far easier than IDA Pro. Binary Ninja exposes both a low-level IR and a medium-level IR (in both SSA and non-SSA form) in addition to their decompilation. Ghidra also makes its P-Code IR, easily accessible to the user. In the case of IDA, although it has an IR, it's less publicly accessible and easily scriptable. This divide is also reflected in these tools' respective methods of decompilation. Ghidra and Binary Ninja both decompile from their respective IRs in an architecture-independent manner, meaning that any architecture that can be lifted to their IRs and subsequently decompiled. Although there is less publicly available information on the internals of the IDA Pro decompiler, it requires a separate decompiler to be purchased for each architecture.

Ghidra is a completely free open-source tool, whereas both Binary Ninja and IDA Pro are closed- source and cost different amounts of money. An IDA Pro license with decompilers for just x86 and x86-64 costs several times that of a Binary Ninja license (which can decompile any architecture which lifts to its IR).

3.2.2 IDA Pro - Updates

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

Updates
In May 2020, Hex-rays released the 7.5 release of IDA Pro, which incorporated many new features, including some UI changes that seem to have been inspired by Ghidra. IDA Pro now offers folders across many of its views to help categorize names and types more clearly. IDA Pro also introduced decompiler support for multiple MIPS variants including big-endian MIPS32, little-endian

MIPS32, MIPS16e, and microMIPS. Type libraries for macOS and iPhone SDKs have been included as well, and Lumina has been updated to support MIPS and PPC. There are several major instruction enhancements as well, including decompiler support for ARM atomic instructions, the Intel Control-flow Enforcement Technology instructions, and improvements for better decomplication of optimized MOVW and MOVT instruction pairs. [40]

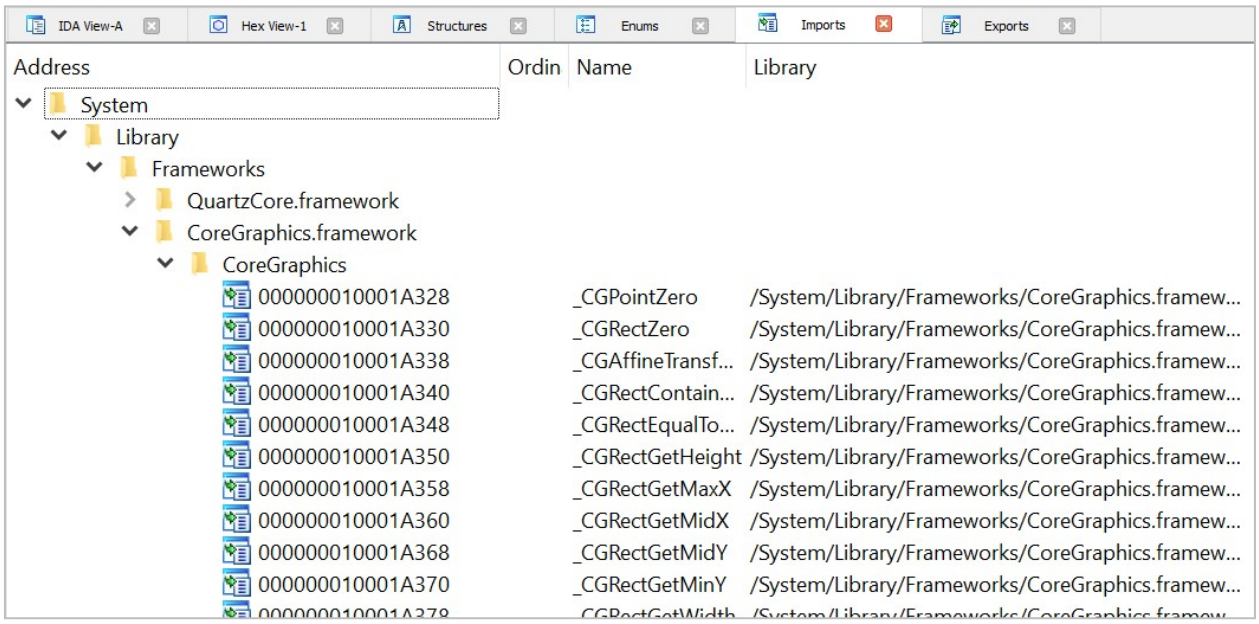


Figure 9 Folder View of Program Imports

In October 2019, Hex-rays released IDA pro 7.4. The update included Python 3 support, added the S390, Renesas M16C/80, M32C, and R32C processor modules, along with various improvements to the decompiler, structure editor, and other features. [41]

3.2.3 Ghidra - Updates

Reference Link	**https://ghidra-sre.org/**
Target Type	Binary
Host Operating System	Linux; macOS; Windows
Target Operating System	**Windows:** PE (Portable Executable); etc. **macOS/iOS:** Mach-O; etc. **Linux:** ELF; etc. **Android:** DEX Format; etc. **Other:** COFF (Common Object File Format); Raw Binaries; etc.
Host Architecture	x86 (32, 64)
Target Architecture	**Disassembler and Decompiler:** 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.
Initial Release	March 2019
License Type	Open-Source
Maintenance	Maintained by the National Security Agency (NSA)

Updates
Ghidra has had one major update and 2 minor updates since the last EotA report. The 9.1 release in October 2019 added bit-field support to data types, Eclipse integration for the Sleigh Editor, and various GUI improvements. Additionally, Ghidra now has a MachO executable file format importer, can preserve the imported program's original memory map, and added support for the following processors: [42]

- Intel MCS-96
- SH1/2/2a/SH4
- Tricore
- HCS12X
- HCS05/HCS08
- MCS-48

3.3 Binary Differentiation

3.3.1 Hashashin

Reference Link	**https://github.com/riverloopsec/hashashin**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2019
License Type	Open-Source
Maintenance	Maintained by River Loop Security

Overview

Hashashin is a tool for Binary Ninja which detects function similarities between two binaries, and ports these basic block annotations from one to another. Hashashin is implemented as Python tool that generates tags for each of the functions within a program's Binary Ninja database file, which can then be applied to a different program by generating a Binary Ninja database file with those tags. Hashashin uses Locality Sensitive Hashing (LSH) and graph hashing to identify similarities between binaries even if they are not identical. This is useful for comparing different versions of the same software, comparing different software which performs the same function, patch inspection, etc. [43] [44] [45]

Design and Implementation

There are two different kinds of hashing used in Hashashin. The first is basic block hashing, which is done using Locality Sensitive Hashing (LSH). Unlike other kinds of hashing (i.e., cryptographic hashing), LSH is designed to generate the same hash for similar but not necessarily identical inputs. LSH is used to generate a signature for each basic block in the binary. Those signatures act as the

label for the corresponding basic blocks in the function's CFG. Then, graph-based hashing is used (i.e., Weisfeiler Lehman Isomorphism Test) to check for similarities between graph structures.

To generate signatures for a Binary Ninja database file, the user runs the *generate_signatures.py* script as follows:

```
./src/generate_signatures.py <input_binary_ninja_db_file>
                    <signature_file> [43]
```

To apply signatures, the *apply_signatures.py* file is used.

```
./src/apply_signatures.py <input_binary> <signature_file> [43]
```

Use Cases and Limitations

Riverloop developed this tool while they were participating in the DARPA program SafeDocs, in which was intended "to develop new methods for understanding and simplifying complex document formats (e.g., PDF) to safer, clearly understandable, and 'verification-friendly' subsets [44]." Hashashin was developed in order to compare distinct parsers that had similar behavior. However, there are many use cases for a tool which ports Binary Ninja tags between binaries, including patch comparison, comparison between different version of the same library or program, and comparing different programs which perform the same function.

Riverloop found that this tool was most limited when porting tags between different architectures, but it was highly effective doing so between programs of the same architecture. A benchmarking table for Hashashin is shown in Figure 10. Using a benchmarking script, Riverloop performed an analysis of their tool on different PDF parsers.

Binary Name	Version	Architecture	Debug Symbols[4]	Percent of Tags Ported
pdfimages	4.01.01	x86-64	Y	98.5%
pdfinfo	4.01.01	x86-64	Y	98.3%
pdftotext	4.01.01	x86	Y	30.5%
pdfimages	4.01.01	x86	Y	30.5%
pdftotext	4.02	x86-64	Y	94.8%
pdfimages	4.02	x86-64	Y	90.5%

Figure 10 - Benchmarking Hashashin on PDF parsers [44]

3.3.2 DeepBinDiff

Reference Link	https://github.com/deepbindiff/DeepBinDiff
Target Type	Binary (C/C++)
Host/Target Operating System	Linux, Windows
Host/Target Architecture	x86 (32, 64)
Initial Release	January 2020
License Type	Open-Source
Maintenance	Maintained by Cornell University

Overview

DeepBinDiff is a tool to perform static binary differentiation, or 'diffing,' of compiled programs. This technique was presented in the 2020 NDSS paper *DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing* [46] by Duan et al. Like Zynamic's BinDiff [47], a mature, popular tool for the same purpose, this tool is designed to be used by reverse engineers, security researchers, and vulnerability analysts to accurately identify salient differences between two binaries. DeepBinDiff distinguishes itself by using an unsupervised neural network pattern recognition approach to analyze binaries. [46] [48] [49]

DeepBinDiff performs its analysis on the basic blocks of an interprocedural CFG (ICFG). Like BinDiff, DeepBinDiff is designed to use an ICFG generated by IDA Pro. DeepBinDiff can also use angr to generate the ICFG. The authors describe their approach as follows,

> "We propose an unsupervised deep neural network-based program-wide code representation learning technique for binary diffing. In particular, our technique first learns basic block embeddings via unsupervised deep learning. Each learned embedding represents a specific basic block by carrying both the semantic information of the basic block and the contextual information from the ICFG. These embeddings are then used to efficiently and accurately calculate the similarities among basic blocks. [46, p. 2]"

Design and Implementation

The operational stages for DeepBinDiff are shown in Figure 11.

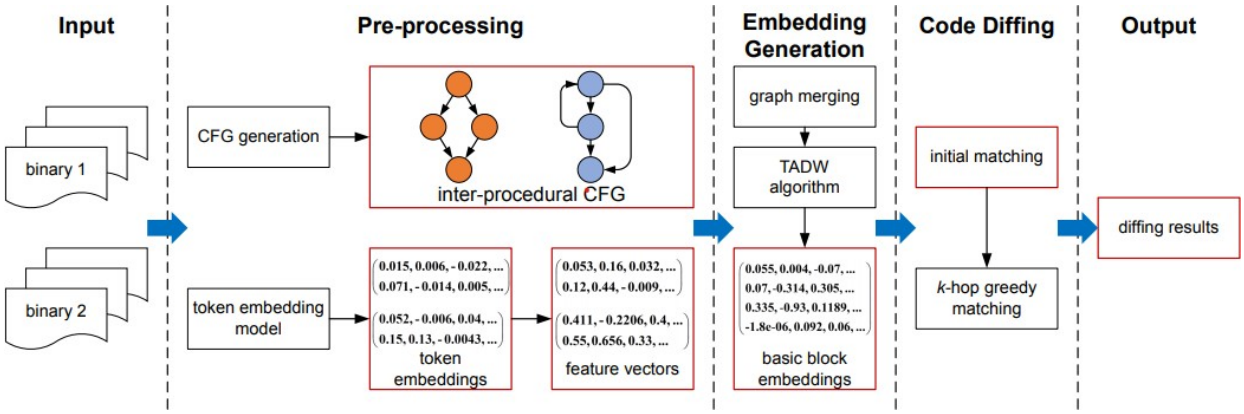


Figure 11: Stages of Operation of DeepBinDiff [46, p. 4].

The system is divided into three components:

1. **Pre-processing:** For each input binary, Interprocedural Control Flow Graphs (ICFG) are created to describe the program functional flow. Next, tokens consisting of opcodes and operands are generated using the Word2Vec unsupervised learning model [46, p. 3]. These are then used to create feature vectors for each basic block in the program [[46, p. 4].

2. **Embedding Generation:** The ICFGs and basic block feature vectors are used as inputs to the Text-Associated DeepWalk (TADW) unsupervised graph learning algorithm to generate embeddings of the basic blocks [46, p. 5]. These embeddings describe the basic blocks and the vertices between them (i.e. code flow) as vectors, meaning that similar embeddings will have values close to each other [46, p. 5].

3. **Code Diffing:** The final stage is to perform the actual code diffing to identify changes between the two binaries. The basic block embeddings are fed to a *k*-hop greedy matching algorithm proposed by the paper's authors [46, p. 6]. The output of this final stage is three sets: the matched pairs, inserted blocks, and deleted blocks [46, p. 6].

Implementation: DeepBinDiff is written in Python3 and has a number of dependencies, including TensorFlow [50] for the machine learning component and angr [6] or IDA to generate the ICFG.

Use Cases and Limitation

The use cases for DeepBinDiff are generally the same as those of BinDiff: security patch analysis, malware analysis, patch-based exploit generation, and plagiarism detection. The authors assert that their approach results in higher accuracy, and when used with GPUs, faster performance.

To judge the efficacy of their approach, the authors use source code to determine the ground truth for all matching basic blocks in two versions of the program (the set *G*). The result *M* is the set of matching basic blocks found by DeepBinDiff. Then, a correct match, *Mc,* is the intersection of the result *M* and the ground truth *G*, or $Mc = M \cap G$. A "better" performing tool has a larger set *Mc* than another.

In their evaluations, the authors found that DeepBinDiff outperformed BinDiff on average. However, DeepBinDiff is a far more computationally and memory intensive tool, due to the needs of the neural net. The authors note, and we witnessed, a correlation between runtime and binary size, since embedding generation and matching are bounded by the size of the ICFG. C++ programs can also introduce problems generating complete ICFGs, which would have a direct impact on the accuracy performance of DeepBinDiff since it relies on these to provide contextual information for basic block embedding. Finally, the technique requires training the model in order to identify tokens of opcodes and operands, which is not required by BinDiff. This can add a significant runtime that is not incurred by traditional diffing approaches.

When we first evaluated DeepBinDiff in early April 2020, the documentation for the author's implementation of DeepBinDiff, found in [49], was sparse. We encountered a number of difficulties when attempting to run DeepBinDiff against the OpenSSH client [51], versions 8.1 and 8.2, and were ultimately unsuccessful in obtaining useful results. Since our initial evaluation, the authors have added more documentation, as well as modified their source code to fix our build errors. As of mid-May 2020, the authors appear to be still actively updating the GitHub project with additional documentation and source code updates.

3.4 Kernel Static Analysis

3.4.1 Crix

Reference Link	**https://github.com/umnsec/crix**
Target Type	Source (Linux kernel)
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	October 2019
License Type	Open-Source
Maintenance	Maintained by University of Minnesota

Overview

Crix (Criticalness and constraints Inferences for detecting missing-cheCKS) is a static analysis tool that analyzes Linux kernel source code and detects a software error known as missed-check. The tool was presented in the USENIX 2019 paper, *Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences* [52], which addresses the need for a systematized approach to identify absent validation checks in OS kernels. These missed-check errors are a bug class that arises when intended security checks are not, or are incorrectly, enforced. Crix is a static analysis tool for detecting these missing checks in a scalable fashion. Crix [52] is related to the tool CheQ [53] which was created by the same authors at the University of Minnesota. The authors claim to have successfully identified over 200 missing check bugs by using Crix to evaluate the Linux kernel. [52] [54] [55]

Design and Usage

Missed-check errors are a class of semantic bugs that arise when an intended security check isnot enforced on a critical variable (CV) (i.e., those variables that are validated by a security check). Concrete examples of this include making a kmalloc() system call and then failing to examinethe return value to ensure it completed successfully. Or checking that the size parameter to memcpy() is not larger than the destination buffer. These checks will be missed in various locations in the code, but present in others. An example of a missing-check error found by Crix is shown in Figure 12.

```
1  /* Linux: net/smc/smc_ib.c */
2  static void smc_ib_remove_dev(struct ib_device *ibdev...)
3  {
4    struct smc_ib_device *smcibdev;
5      /* ib_get_client_data may fail and return NULL */
6    smcibdev = ib_get_client_data(ibdev, &smc_ib_client);
7      // ERROR1: NULL-pointer deference
8    list_del_init(&smcibdev->list);
9      /* ERROR2: device cannot be removed or unregistered */
10   smc_pnet_remove_by_ibdev(smcibdev);
11   ib_unregister_event_handler(&smcibdev->event_handler);
12     /* ERROR3: memory leak */
13   kfree(smcibdev);
14     /* No return value: caller cannot know the errors */
15 }
```

Figure 12: Missing-check error in the Linux kernel found by Crix

Crix's stages of operation are shown in Figure 13. Using LLVM compiled bitcode, control flow and call graphs are constructed and used to identify critical variables. For each CV, "peer slices of source code are constructed that share similar semantics and contexts" [52, p. 2]. Next, it "models the constraints of conditional statements [(e.g., `if`, `switch`)] in each slice" and cross checks them with each peer slice to detect deviations [52, p. 2]. These deviations are then reported as potential missed-check bugs.

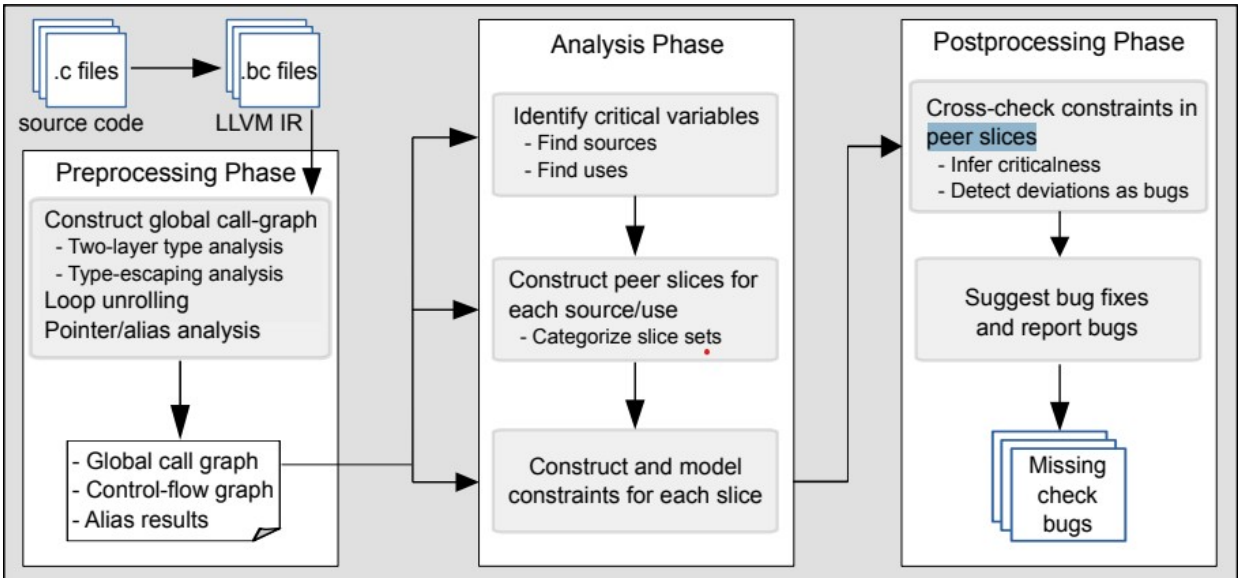


Figure 13: Overview of the Crix operation [52, p. 5]

There are several inherent challenges in detecting missing-check bugs:

1. CVs that require checking take many forms. For example, return values that are not used in arithmetic operations, a parameter of a critical function, or global variables.

2. Identifying security checks requires semantic understanding. About 70% of conditional statements are not security checks but normal execution branching.

3. Missing-check bugs are context dependent. For example, the error code might be used in a debugging function and so is not needed for error checking.

4. OS kernels are large and analyzing every variable will not scale.

Crix solves these problems with the introduction of four new techniques:

1. A two-layer type analysis to identify indirect call targets. LLVM does provide targets of indirect calls, and Crix requires them in order to augment the LLVM control and call flow graphs to identify peer slices and critical variables.

2. An automated method for finding CVs by identifying security checked variables, which significantly narrows the analysis scope and allows large complex code bases, such as the Linux kernel, to be analyzed.

3. For each CV, Crix looks for slices of program code that share similar semantics and constraints with the code path that checks the current CV. It finds these peer slices by identifying sources and uses of the CV and data flow analysis to find the program slices for each source and use.

4. Crix extracts the constraints from the conditional statements in the slices in a way that preserves the semantics. It then cross-checks them, via statistical analysis, to calculate the relative frequency of the source or use in its peer slices. “If significant, not having a constraint would be identified as a deviation, and a slice that does not have the constraint is identified as a potential missing-check bug and reported [52, p. 5].”

Implementation: Crix is implemented in C++ and targets kernel files compiled to LLVM bitcode.

We ran Crix (`kanalyzer -mc`) against the Linux kernel networking files that the authors provided precompiled to LLVM bitcode. An example of the output from the tool is shown in Figure 14. We randomly chose several of the files identified as possible issues, and they did show function calls that were not explicitly checked for an error in the return value. In one case, the value was checked with a macro that was bitwise XORing it with another variable, and then raised an error if the result was zero. In another, the value was a pointer cast to a struct, and “checked” via a function that returns the VLAN ID, if present. In those cases, Crix found potential missing checks. However, upon closer inspection the values were being checked but in a non-standard way.

```
== [Src-retval]: Rating: 0.200, Checks: 4, Unchecks: 1, Total: 5 | Arg: -1

 [Code] net/smc/smc_core.c +568:


== [Src-retval]: Rating: 0.091, Checks: 10, Unchecks: 1, Total: 11 | Arg: -1

 [Code] net/llc/llc_conn.c +371:


== [Src-param]: Rating: 0.200, Checks: 4, Unchecks: 1, Total: 5 | Arg: 0

 [Code] net/core/rtnetlink.c +3218:
```

Figure 14: Sample Output for CRIX

Use Cases and Limitation

The current use case for Crix is detecting missing-check errors in the Linux kernel. This target was chosen to demonstrate the scalability of the author’s fast CV detection and slice construction features. The authors envision it to be portable, with some effort, to other software targets, but that goal has not yet been demonstrated. They note specific porting issues (described below) that would need to be addressed in order to use Crix with another source code base.

Other tools that detect missing-check bugs only analyze specific critical operations, such as arithmetic division and array indexing, and are therefore limited in what they can detect. Other approaches use machine learning, statistical cross-checking, and inconsistency analysis, but are limited in that they do not differentiate on what conditions are security checks versus another type of conditional statement (i.e., they are not semantically aware). They also rely on manual specification of critical variables, and therefore are susceptible to a high false negative rate.

Crix contains an RF (relative frequency) threshold value that is used to determine whether a deviation from a constraint should be classified a potential bug. The authors have tuned this value for the Linux kernel, but it may not be appropriate for other software targets. Also, Crix contains code that looks specifically for Unix style error codes and handling functions. In Unix-like kernels, these patterns are the same, but other software may follow different conventions. Before using Crix against these other types of software projects, the header files containing these error codes and function prototypes need to be identified and Crix must be able to analyze them in order to create error control flow graphs. Depending on the code base of the target application and its complexity, this could be a non-trivial task for porting Crix for other targets.

While the source code is available on Github, there is no interface or API for extension or customization. As an academic research project, the source was written with a particular goal in mind and was not designed to be applied to source targets other than the Linux kernel. Additional work would be needed to evaluate Crix's use against source code for other OS's or software tools. Since it is dependent on the bitcode generated by LLVM, any source that can be built successfully this way should be amenable to Crix's analysis.

3.5 Trends

Recent trends in static analysis tools have improved upon existing ideas. The competition between IDA Pro, Binary Ninja, and Ghidra sees them adopting features from each other in order to further their own usability. Additionally, there's been a notable move to improve APIs to enable easier and more powerful tool development, as seen with the major reverse engineering platforms. Additionally, tools like Hashashin, DeepBinDiff, and Crix are modernized takes on existing ideas, furthering the state of the art for binary analysis tools by incorporating new analytics and machine learning.

4 Dynamic Analysis

4.1 Technical Overview

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

This section discusses debuggers (drgn [56]), instrumentation frameworks (Qiling [7]), and dynamic binary analysis frameworks (Triton [57]). Triton can be used for a range of analyses, including taint analysis and symbolic execution. BAP and Miasm are two analysis frameworks that are often used for symbolic execution and other dynamic techniques. They were discussed in the static analysis section because of their heavy dependence on their IR's. This section also includes constraint solving (JFS [58]), because of its use in symbolic/concolic execution tools.

Debuggers

Relevant EotA Tools: WinDbg, Binary Ninja, rr, drgn

Among other uses, interactive debuggers can pause a program during execution and step through one instruction at a time, to inspect the current state of registers and memory. Debuggers can be used to reverse engineering a program to determine how it operates, to inspect a crash found by a fuzzer, or to debug an exploit. Like many dynamic analysis tools, debuggers utilize both static and dynamic techniques. For example, the popular debugger GDB uses a disassembler and the tracing utility ptrace [59].

Recordable, replayable debugging is one of the most powerful additions to modern debugging tools. This allows a user to record and replay program execution while debugging the process. In addition to forward debugger actions like step and continue, replayable debugging allows the user to step backwards and continue backwards. TTD is a tool discussed in this section that allows for replayable debugging from within the Windows debugger Windbg [60]. rr [61], enables recordable and replayable debugging on Linux and was discussed in the first version of this report.

Dynamic Binary Instrumentation (DBI)

Relevant EotA Tools: Frida, Qiling

DBI, which underlies many dynamic binary analysis techniques, entails modifying the binary, either before or during execution, often by hooking the binary and injecting code. DBI frameworks implement custom instrumentation which the user can access through an API. These include Intel Pin [62] and DynamoRIO [63], which underlie many of the tools discussed in these reports. Both can be used to drive the Windows fuzzer WinAFL [64]. The dynamic binary analysis tool Triton is built around Intel Pin [57]. DBI frameworks are implemented in a variety of ways. Intel Pin works by intercepting instructions before they are executed and recompiling them into a similar Intel Pin-controlled instruction [62]. It is analogous to Just-In-Time (JIT) compilers. DynamoRIO operates similarly in that it sits in between the application and the kernel, like a "process virtual machine," to observe and manipulate each instruction prior to execution [63]. Other DBI options are less granular and intrusive, and rely on hooking the program through dynamically loaded libraries (e.g., this is how the tool Frida [8] operates).

Dynamic Fuzzing Instrumentation

Relevant EotA Tools: Frida, Qiling,

Although fuzzing is discussed at length in the next section, fuzzing often requires dynamic binary instrumentation to feed input quickly and easily to the program. This can be done with various tools (e.g., Frida, Qiling, etc.) that allow the user to hook and redirect the input to the binary.
These tools can hook the binary and redirect execution around the problematic code like checksums, or other functionality that can inhibit fuzzing. The fuzzer Frizzer uses Frida to instrument it.

Memory Checking

Memory checking, whether to find memory bugs or analyze them is a valuable form of dynamic analysis in vulnerability research. To do this, a program is instrumented such that if a memory error is triggered during runtime (e.g., an out of bounds access, null pointer dereference, or segmentation fault) it will be recorded, along with additional contextual information. Several tools that do this are Valgrind [65], Dr. Memory (a part of the DynamoRIO framework) and LLVM's Sanitizer Suite which includes Address Sanitizer (ASAN) [15].

Dynamic Taint Analysis

Relevant EotA Tools: Triton, angr

Dynamic taint analysis is a form of dynamic binary analysis that 'taints' data (often some kind of input) such that its flow throughout the program can be traced. This can be done on the byte or bit- level with a tradeoff between the fidelity of the analysis and the time and memory resources required. Dynamic taint analysis is often built on top of dynamic binary instrumentation. Data transfer instructions are hooked to check whether the source memory or register value is tainted and then taint the subsequent destination (or conversely, remove a taint from a destination if the source lacks a taint). Dynamic taint analysis is not just useful for tracking values throughout a program. It is also helpful in concolic execution because it can identify which instructions are not affected by user input. Triton is particularly effective at dynamic taint analysis and discussed in this section.

Symbolic and Concolic Execution

Relevant EotA Tools: angr, Triton, Miasm, BAP 2.0, Manticore, QSYM

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

Symbolic execution can be performed "dynamically," and this is called dynamic symbolic execution (DSE). However, throughout the symbolic execution literature there are generally two competing definitions of DSE. The first kind of DSE refers to any form of symbolic execution which "explores programs and generates formulas on a per-path basis [66, p. 1]". This does not mean that only one path is followed, just that a distinct formula is generated for each path. When a branch condition is reached, and both branches are feasible, execution will "fork" and follow both possible paths [66, p. 3]. In the paper *(State of) The Art of War: Offensive Techniques in Binary Analysis* [29], Shoshitaishvili et al. describe this kind of DSE:

> "Dynamic symbolic execution, a subset of symbolic execution, is a dynamic technique in the sense that it executes a program in an emulated environment. However, this execution occurs in the *abstract* domain of *symbolic variables.*
>
> ...
>
> "Unlike fuzzing, dynamic symbolic execution has an extremely high semantic insight into the target application: such techniques can reason about how to trigger specific desired program states by using the accumulated path constraints to retroactively produce a proper input to the application when one of the paths being executed has triggered a condition in which the analysis is interested. This makes it an extremely powerful tool in identifying bugs in software and, as a result, dynamic symbolic execution is a very active area of research. [29, p. 6]"

Symbolic execution can be combined with concrete execution in a variety of ways and this is often referred to by the portmanteau "concolic" execution. "Concolic" is another term with competing definitions but is often used as a synonym for DSE. Concolic execution can refer to the kind of DSE described in the previous excerpt, in which symbolic (not concrete) inputs are used, and all possible paths are explored, but the program execution will switch between concrete and symbolic emulation, depending on whether the instruction handles symbolic values [6].

The other common definition of DSE and concolic execution refers solely to symbolic execution as "driven by a specific concrete execution [67, p. 6]." A program will be executed both concretely and symbolically using a chosen concrete input, and the symbolic execution will only follow the specific path taken by the concrete input [68] [67, pp. 5-6]. After doing this, additional paths can be explored by negating one (or more) of the collected branch conditions for the path of the concrete input, and then solve for the new path with these negated conditions using an SMT solver in order to generate a new input [67, p. 6]. This kind of DSE or concolic execution is often used in symbolic assisted fuzzing, also known as hybrid fuzzing, which use symbolic techniques to gain semantic insight while fuzzing a program. QSYM [69] (discussed in the first version of this report) is an example of hybrid fuzzing.

There are many tools for symbolic execution, including Triton and Miasm. angr [6] (discussed in the first version of this report) is one of the best [70], publicly available tools, and uses emulation to perform symbolic execution.

While symbolic execution does provide powerful insights into program semantics, it is greatly limited by space and time complexity issues. Path explosion is one of the challenges in symbolic execution. Unbounded loops might result in an exponential number of new paths. Symbolic execution is also hindered by the memory needed to store a growing number of path constraints. It is also difficult to apply to real-world systems, because system calls and library calls can be hard to manage with symbolic values [67]. Additionally, constraint solving is a difficult and time- consuming task. As such, symbolic execution is in many cases not a feasible option or must be constrained to a small area of the program.

Constraint Solving

Relevant EotA Tools: JFS (Constraint Solver), angr, Triton, Miasm

Symbolic execution relies on the ability to solve for the collected path constraints, which is a challenging problem. These constraints can be modeled by satisfiability modulo theories (SMT) which generalize the boolean satisfiability problem (SAT). SAT is an NP-complete problem that looks for a set of values which will satisfy the given boolean formula. An SMT formula models a SAT problem with more complex logic that involves constructs like inequalities or arrays. A SAT formula is a boolean expression made up of boolean variables, and the boolean operators AND, OR and NOT.

One major limitation of SMT solvers is their performance. These solvers are hindered by time complexity and the difficulty of determining satisfiability. The second version of this report discussed the SMT solver JFS which used fuzzing to solve floating point SMTs.

4.2 Debuggers

4.2.1 Windbg Preview - Time Travel Debugging

Reference Link	**https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview**
Target Type	Binary
Host Operating System	Windows
Target Operating System	Windows
Host Architecture	x86 (32, 64), ARM
Target Architecture	LLVM IR
Initial Release	December 2019
License Type	Proprietary - Free
Maintenance	Microsoft

Overview

WinDbg Preview is Microsoft's updated version of WinDbg, their veteran Windows debugger. As described in the announcement, WinDbg Preview improves on its predecessor with "more modern visuals, faster windows, a full-fledged scripting experience, built with the easily extensible debugger data model front and center [72]" while using the "same underlying engine [72]." Excitingly, Preview now offers users a 'Dark Mode.'

WinDbg comes with all the capabilities of a standard debugger, including memory inspection, symbol loading, breakpoint setting, and instruction stepping. But the most notable new feature of WinDbg, and that which merits its inclusion in this report, is its Time Travel Debugging (TTD) capability. TTD allows the user to record an execution and then replay it in the debugger, enabling backwards debugging features like step backwards and continue backwards. WinDbg preview also includes the Debugger Data Model which enables querying UI elements and scripting. WinDbg can also be used to debug kernels, as well as user space binaries. [60] [73] [72]

Time Travel Debugging (TTD)

TTD records a trace of a program execution and then allows the user to 'replay' this recording while debugging. The program initially executes without the user debugging it, and debugging begins once recording has finished.

TTD can be enabled when WinDbg Preview is run as a process or attached to an already running process. This differs from the Linux-based recordable-replayable debugger rr, which cannot attach to an already running process and begin recording. Figure 15 shows attaching to an already running *Calculator.exe* process with TTD enabled. Figure 16 shows launching *calc.exe* directly from Windbg Preview.

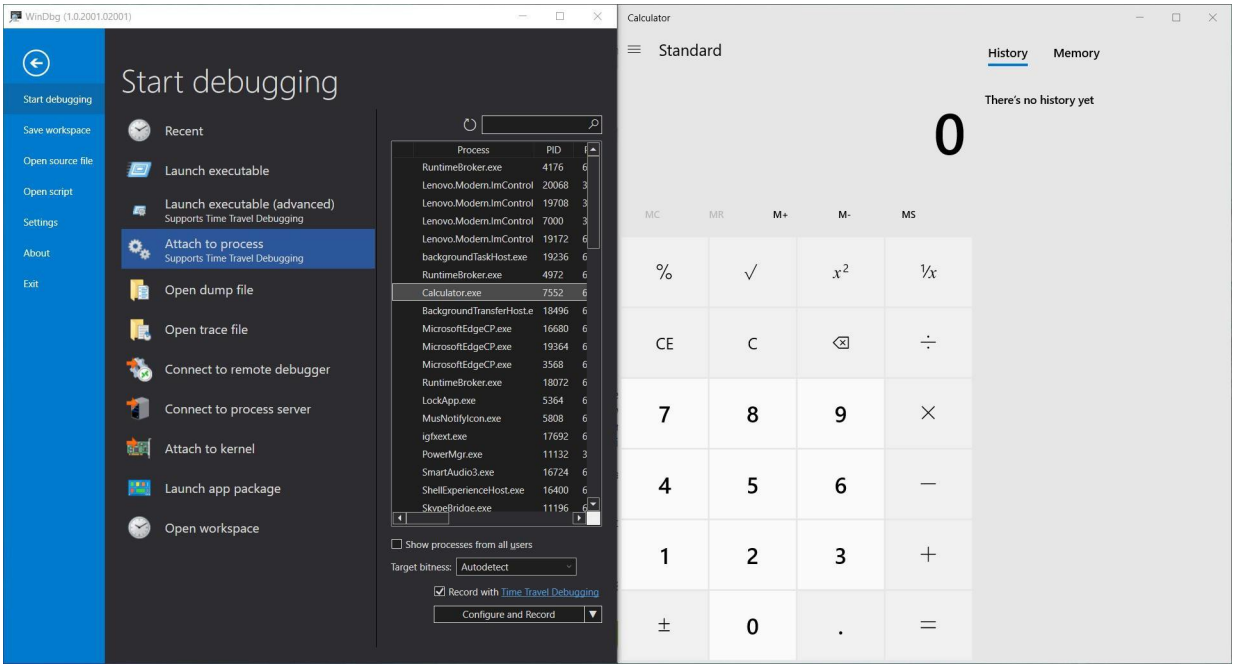


Figure 15: Attaching to an already running Calculator application with TTD enabled

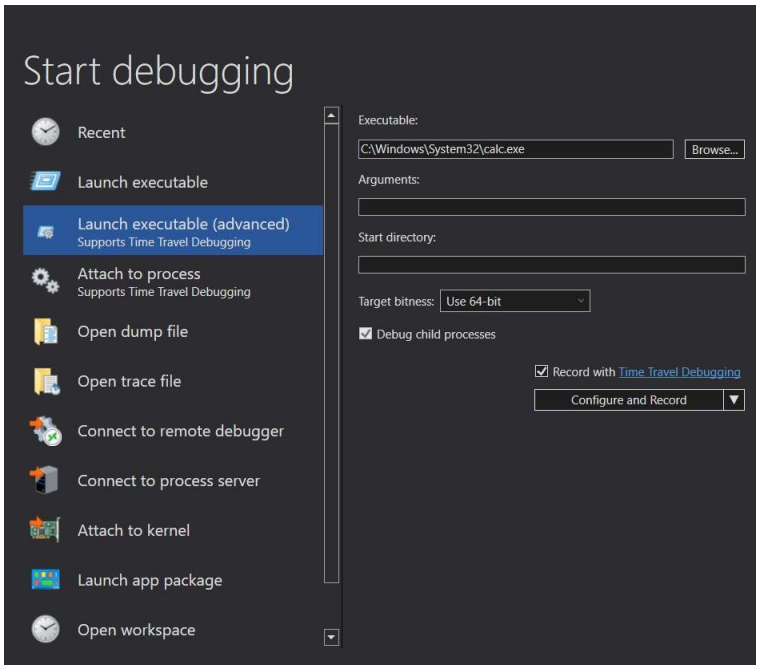


Figure 16: Launching Calculator application directly from WinDbg Preview with TTD enabled

Once a recording has finished can be replayed. An example of this using a toy Hello World program is shown in Figure 17.

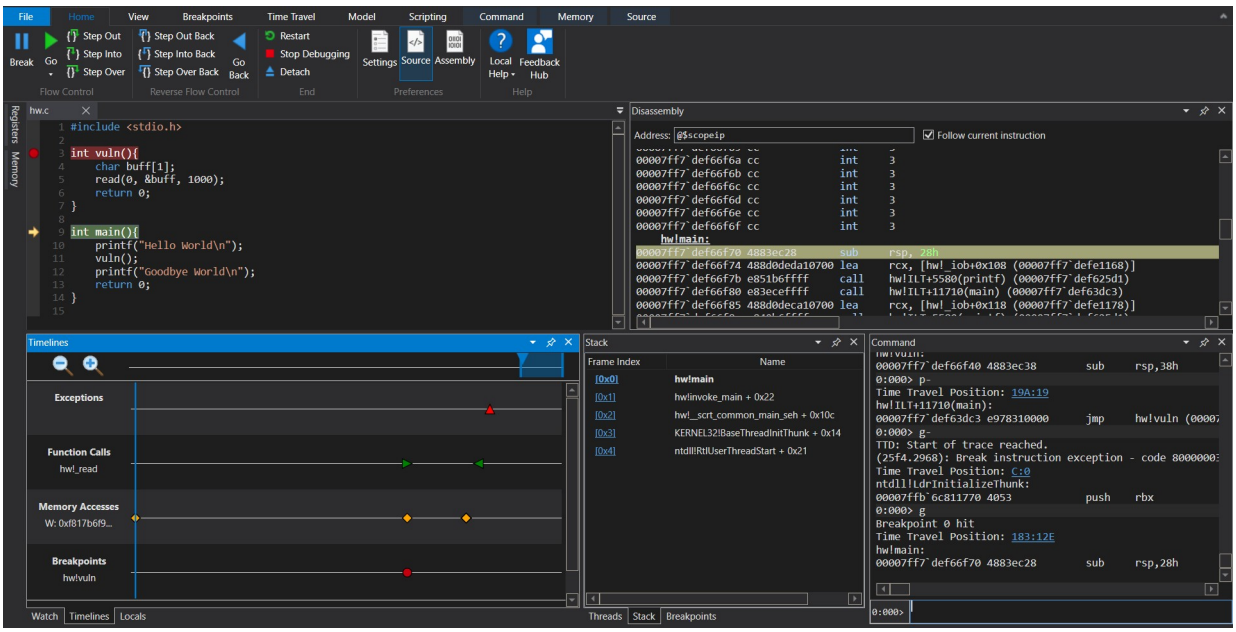


Figure 17: HelloWorld program, hw.exe, running with TTD

In addition to the usual commands such as *Go, Step Out, Step Into,* and *Step Over*, there are the corresponding commands *Go Back, Step Out Back, Step Into Back,* and *Step Over Back.* These can be seen in the upper left corner of Figure 17 and can also be input as commands in the Command window (bottom right corner of Figure 17).

Figure 17 shows a recording of the program *hw.exe* being executed. A source code window shows execution stopped at the beginning of the *main* function, on line 9 of *hw.c*. The program also has a stack-buffer overflow on line 5, which was triggered during the original execution, resulting in an exception when *vuln* returns on line 6.

The Timelines window is one of the most useful features of TTD. This is shown in the bottom left corner of Figure 17. The blue line shows the current execution point (stopped at *main*). Underneath the timeline is an Exceptions timeline, which displays where the exception will eventually be thrown. The user can add specialized timelines for three other events: Function Calls, Memory Accesses, and Breakpoints. Hovering over the events on each of these timelines will show additional information. Double-clicking will transfer execution to the instruction that initiated the event.

There is a breakpoint on *vuln*, which can be seen not just in the Timelines window but also in the Source Code window. Executing a few instructions past that, Figure 18 shows the program about to execute the *read* libc function call.

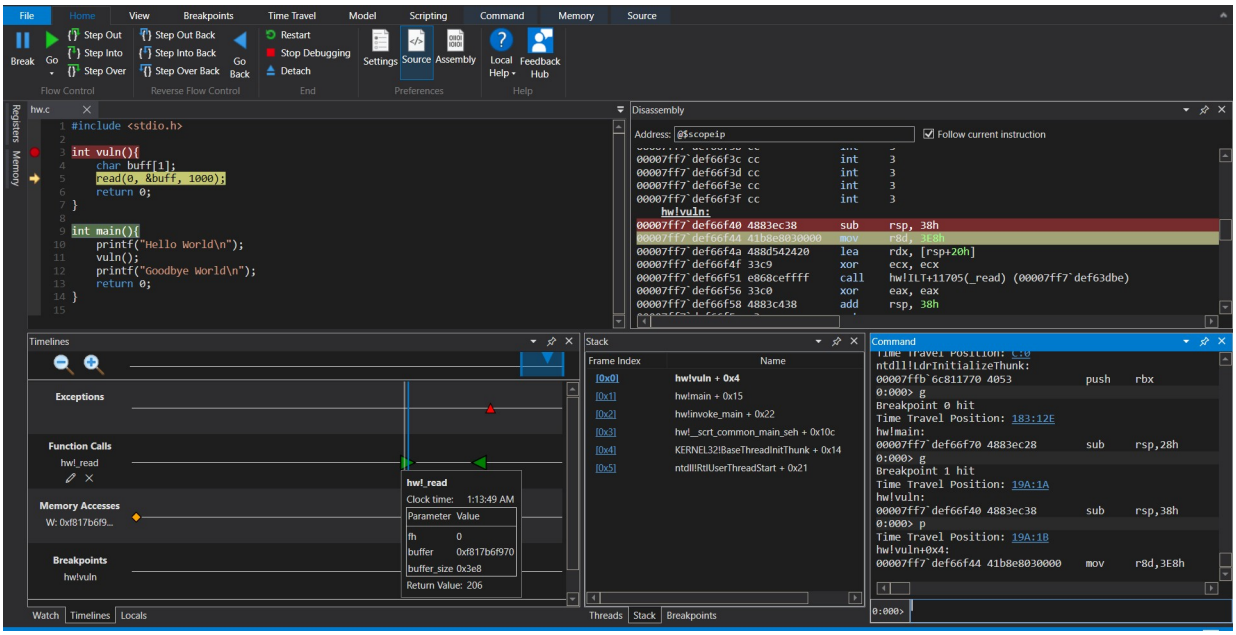


Figure 18: TTD execution of hw.exe stopped at the read function call

This function call has been given a Function Call Timeline, with the green arrows representing the start and finish of the function. Hovering over them shows additional information about the function, including the original time it was executed, its arguments, and return value. Double- clicking the latter green arrow would take execution to the end of the *read* call. However, before that occurs the third memory access is reached, as depicted in Figure 19.

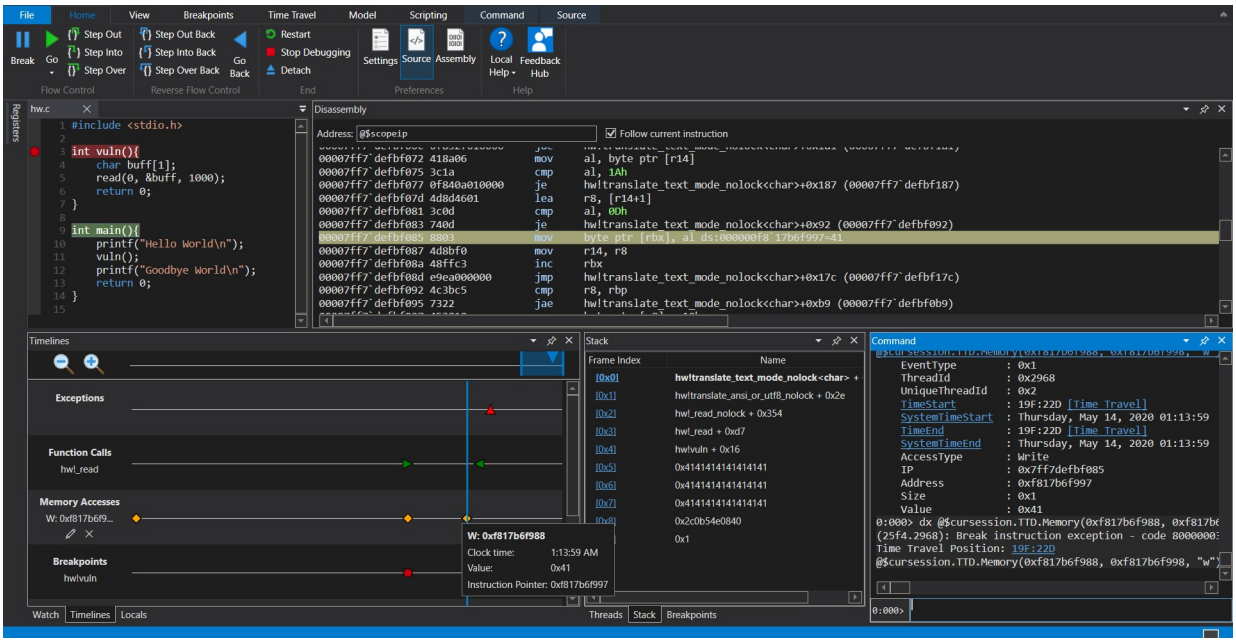


Figure 19: TTD execution of hw.exe reaches third memory access

The memory access being watched is the stack return address for *vuln* which previously held the address of its caller, *main.* Figure 19 shows that this value now holds 0x41, which is reflected in the adjacent Stack window. If this were a normal debugging session and ASLR enabled, then the address on the stack where the return address is held would change with every execution. One of the benefits of replayable debugging is that the execution is deterministic, and all the addresses remain the same for the execution recording.

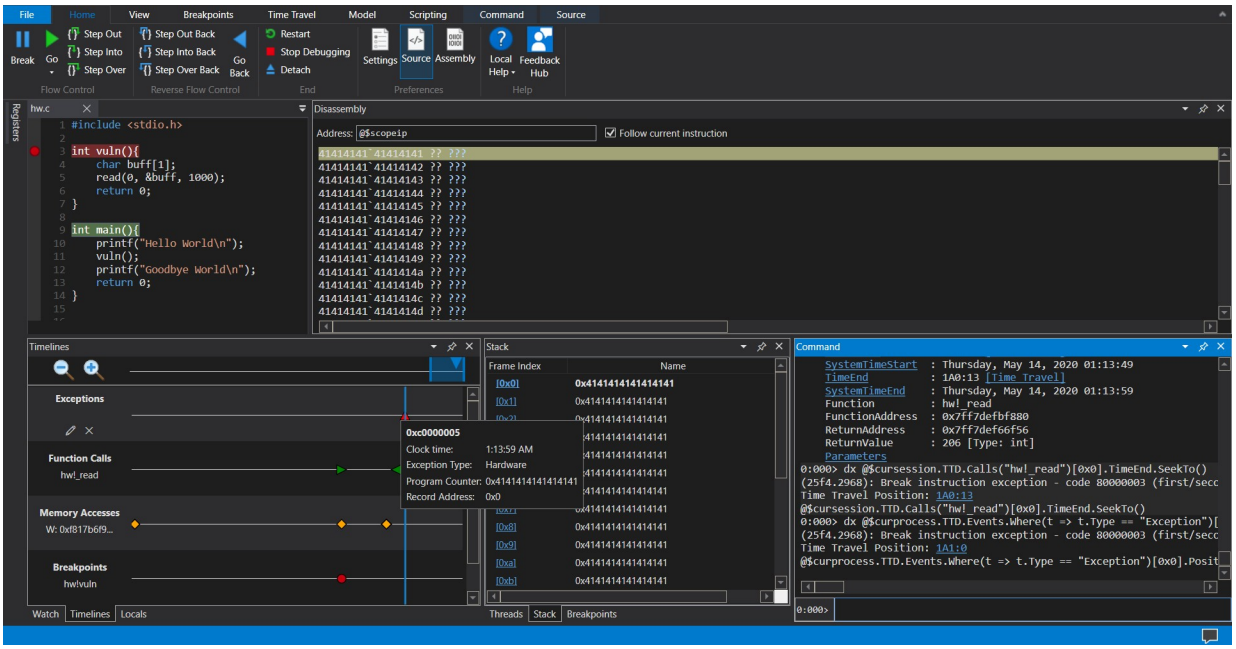


Figure 20: TTD execution of hw.exe hits exception

Finally, Figure 20 shows the TTD execution having hit the exception. Hovering over the exception point on its timeline shows that a corrupted program counter is the reason.

Use Cases and Limitations

The TTD feature makes it easier to trace and analyze the conditions and events of a bug by providing information about the state of the executable both before and after the crash. Replayable debugging is one of the most powerful additions to modern debugging. TTD allows a user to deterministically inspect an execution with static addresses. A user can execute to the end of a long sequence of loops before a crash, and then replay to reach the last iteration of the loop. These examples are only a few use cases for replayable debugging. Unlike rr, Windbg Preview with TTD has a rich user interface, and more functionality.

However, both tools require significant storage. Since WinDbg tracks all data during each execution step, there can be large overhead in the form of huge data files. Another major downside to replayable debugging is that a memory or register value cannot be dynamically modified. Replayable debugging is also limited to user space code. Even though Windbg has built-in kernel debugging functionality, it is currently incompatible with TTD.

4.2.2 Binary Ninja Debugger Plugin

Reference Link	https://github.com/Vector35/debugger
Target Type	Binary
Host Operating System	Linux; macOS; Windows
Target Operating System	Linux; macOS; Windows; Android
Host Architecture	x86 (32, 64);
Target Architecture	x86 (32, 64); ARM (32; 64)
Initial Release	2020
License Type	Open-Source
Maintenance	Maintained by Vector 35

Overview

The Binary Ninja Debugger (BNDP) is an official Binary Ninja plug-in, developed by Vector 35. However, unlike its parent software, BNDP is wholly open source. Depending on the selected target's operating system BNDP integrates with GDB, LLDB, or dbeng. BNDP has the general functionality of a typical debugger, such as stepping through a process and inspecting registers and memory as they are executed. All functionality is completely integrated with the Binary Ninja GUI, allowing the user a far more verbose and interactive process than a command line debugger. Additionally, BNDP can dynamically update information within the binary's Binary Ninja Database (BNDB). [74] [75]

Design and Implementation

BNDP is implemented on top of three different debuggers:

- **GDB**: This has been tested for x86 Linux (in both 32 and 64 bit but tested mainly for the latter) and for ARM Android binaries (in both 32 and 64 bit).
- **LLDB:** This has been tested for x86 (in both 32 and 64 bit but tested mainly for the latter) macOS binaries.
- **dbgeng:** This has been tested for x86 (in both 32 and 64 bit but tested mainly for the latter) Windows binaries. In this case, BNDP is linked to the Windows debugger engine at runtime.

Figure 21 shows a diagram of how BNDP is implemented.

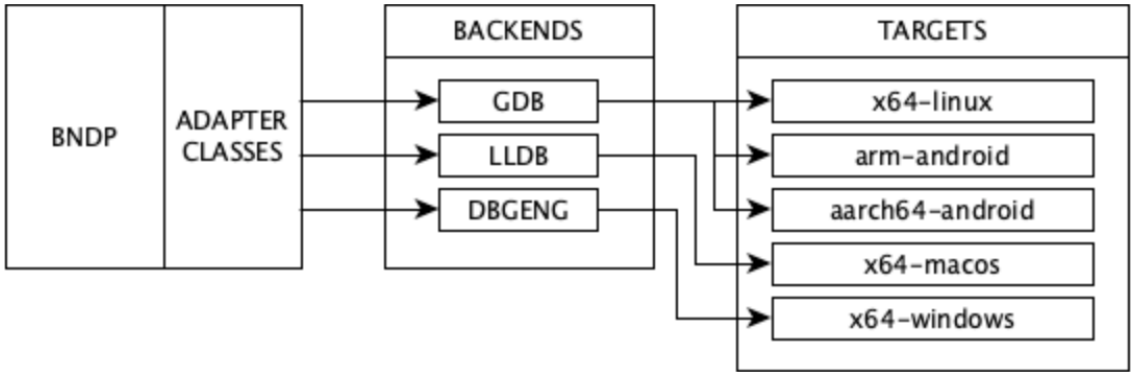


Figure 21: Implementation diagram of BNDP with tested targets

An example of BNDP debugging a MacOS binary is shown in Figure 22.

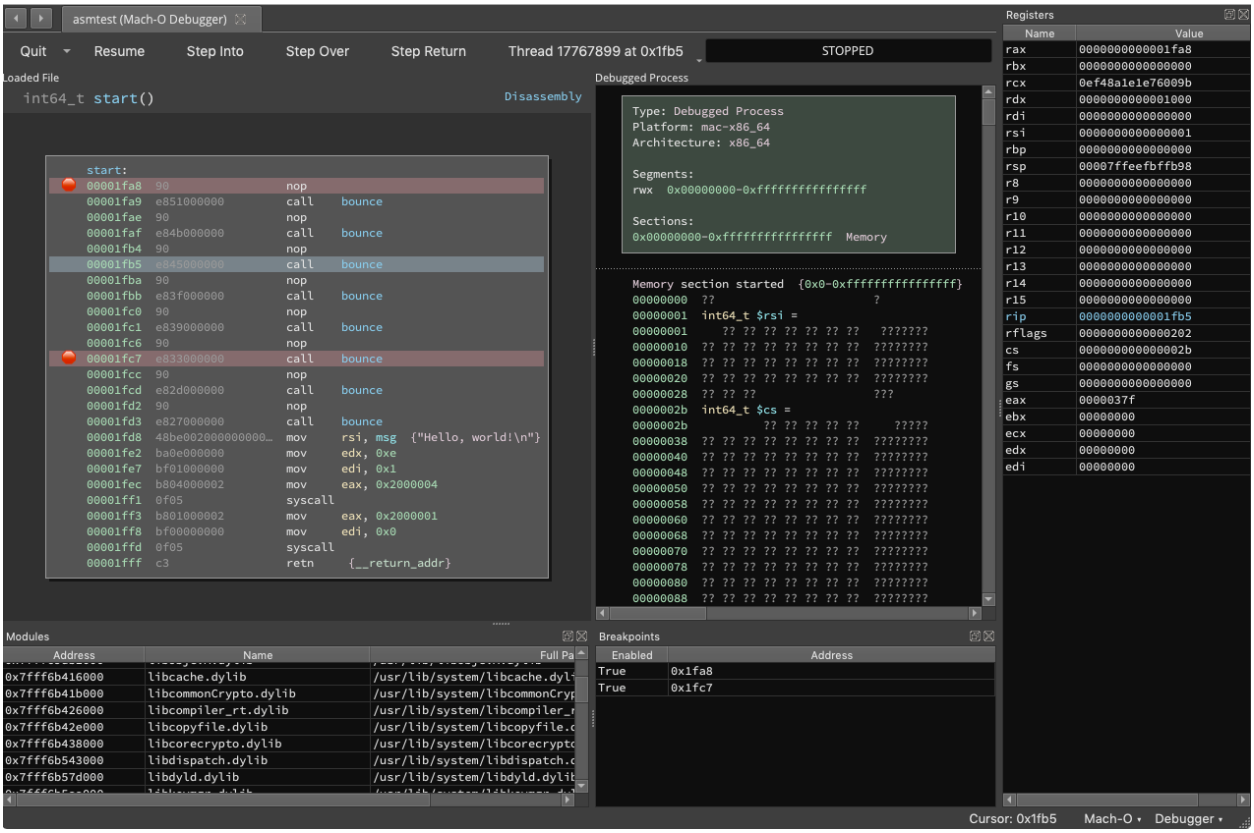


Figure 22: BNDP debugging a macOS binary [75]

As discussed in [74], BNDP allows a user to perform typical debugging operations, such as pause/resume, step into/step over/step return, detach, inspect memory, see register/memory state, and disassemble regions of memory as they are being executed [74]. BNDP also allows the user to incorporate dynamic analysis discoveries back into the BNDB. This is particularly useful for areas of the binary that are resistant to static analysis. Jump tables can be difficult for static analysis, so if the BNDP discovers a new case of a jump table at runtime, then the BNDB's CFG is updated to reflect this. A 'before and after' example of this is shown in Figure 23 and Figure 24,

respectively. Before the *jmp rdx* instruction was reached, the graph showed four outgoing edges from the source basic block. After the *jmp rdx* instruction is reached however, the value in *rdx* shows that there's actually a fifth location. Figure 24 shows the CFG that displays the newly discovered destination basic block.

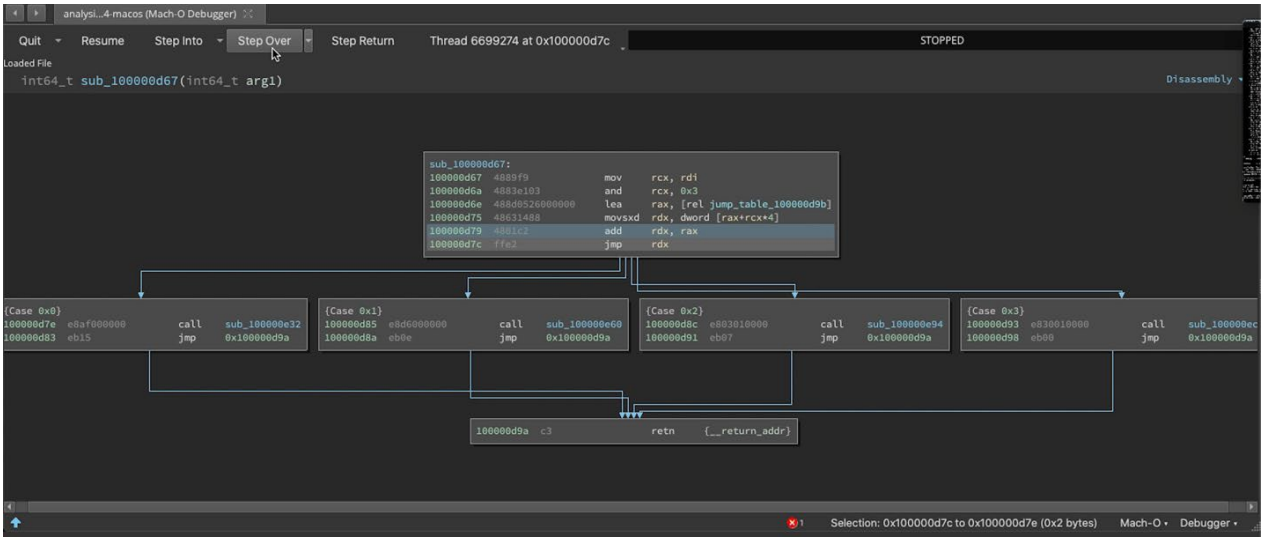


Figure 23: Debugging jumptable in macOS binary before new jump location is dynamically discovered [74]

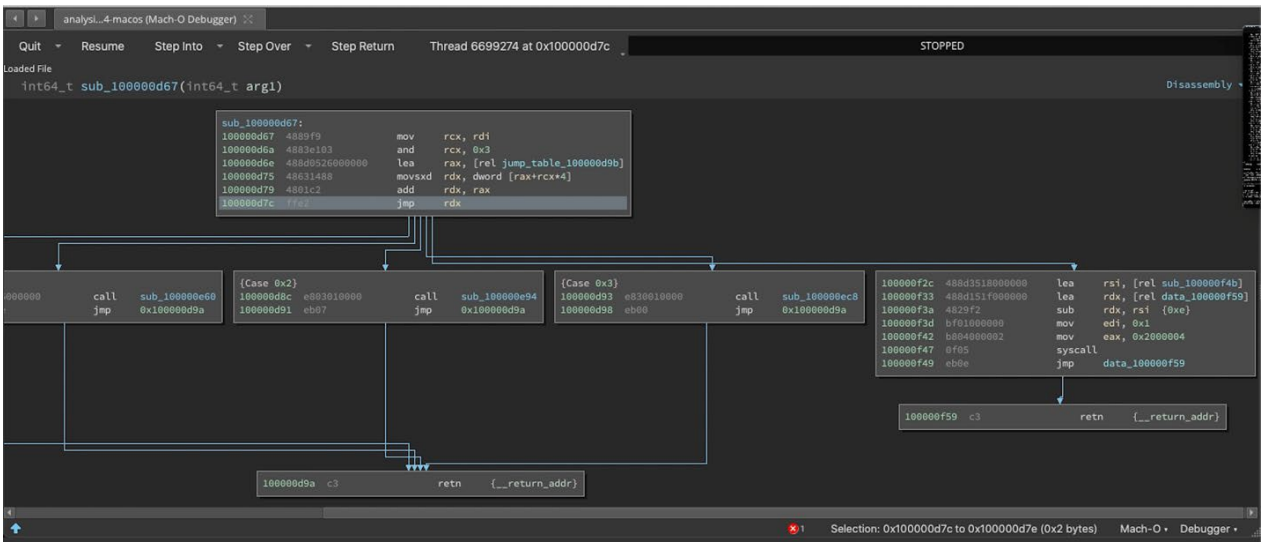


Figure 24: Debugging jump table in a macOS binary after a new jump location is dynamically discovered [74]

This functionality is not limited to switch statements. For example, if an instruction that performs a *call* on a dynamically determined address identify an address that the BNDB previously considered data, then that address would now be rendered as instructions. BNDP also can append annotations to functions. In the case of Indirect Jumps this means dynamically discovered context and target information will be logged.

Use Cases and Limitations

Although there are likely ways that BNDP is more cumbersome and less versatile than running a lightweight command-line debugger, these debuggers lack the comprehensive GUI-based experience that Binary Ninja provides. BNDP has a somewhat symbiotic relationship with the BNDB. The BNDB allows the user to debug a program with the full force of an interactive GUI, in some ways mimicking the experience of an IDE's debugger when debugging source code. In turn, the information gleaned from BNDP relays useful information back to the BNDB, making the BNDB more accurate and useful.

Releasing BNDP as an open-source plugin allows the user to not only build on top of it but use it as a model for their own Binary Ninja plug-ins.

4.2.3 Plutonium-dbg

Reference Link	https://blog.deepsec.net/tag/plutonium-dbg/ https://vimeo.com/307238462 http://www.roots-conference.org/2018accepted.html https://github.com/plutonium-dbg/plutonium-dbg
Target Type	*Linux user-space applications, anti-debugging targets*
Host Operating System	*Linux*
Target Operating System	*Linux (preferably under QEMU KVM)*
Host Architecture	*X86 (64)*
Target Architecture	*X86 (64)*
Initial Release	2018
License Type	Open-Source
Maintenance	Not currently maintained - last updated January 2019

Overview

plutonium-dbg is a debugger for Linux written by Tobias Holl, et. al. at Technical University of Munich and presented at ROOTS 2018. Unlike almost all other Linux debuggers it does not use ptrace. This is useful for cases where a ptrace-debugger like GDB cannot be used, as often happens with malware or closed-source applications attempting to armor themselves against reverse engineering. plutonium-dbg also aims to improve on other limitations of GDB such as debugging multiple threads in a thread group at once without suspending all of them.

The goal is to make it impossible for an application to detect that it is being debugged by plutonium-gdb through techniques like PTRACE_TRACEME, environment variables, manual SIGTRAP/int3, etc. plutonium-dbg comes close to defeating all these techniques, but still has a few detectable tell-tale signs. For instance, it cannot prevent detection of single stepping because of how x86 handles the Trap Flag.

Design and Implementation

plutonium-dbg's primary component is a Linux kernel module. It uses modern (i.e., added in the last 10 years) kernel APIs. Breakpoints in plutonium-dbg are implemented with *uprobes*, a Linux kernel feature that allows tracing of user space code. The advantage of uprobes is that they do not have to halt all threads to maintain consistency when replacing a breakpoint. plutonium-dbg uses *tracepoints* to handle `clone`, `exit` and `exec`. Lastly, it uses *kprobes* to suppress `SIGTRAP`s sent to the application when using the kernel's single stepping.

plutonium-dbg only replaces the backend and uses a custom gdbserver-like Python script as a front-end that is compatible with GDB clients.

Use Cases and Limitations

The primary use case for this tool is for debugging of malware or applications attempting to protect their code from dynamic analysis. If the debugger cannot be detected, the reverse engineer does not need to perform the laborious process of patching out the anti-debugging code.

There are several limitations noted in the author's paper [76]:

- Breakpoints can only be placed into locations within mapped files. However, the authors have an experimental patch that bypasses this uprobes limitation.
- Some GDB server commands, such as memory watchpoints, are not supported.
- The primary limitation, given that this tool's raison d'être is to be undetectable by the debugee, is that it is still detectable by a specially designed debugee in most common cases. As the authors note in their ROOTS 2018 talk [77], a program that scans for breakpoints or examines the Trap Flag (TF) can detect the software breakpoints inserted by plutonium-dbg or single-stepping, respectively.

Given requirements that including running on QEMU-KVM, patching the Linux kernel, and debuging remotely, plain GDB (or a variant such as pwndbg) remains a more convenient choice. plutonium-dbg should be used only if the target application has anti-debugging tricks that are not easily to patch out. The additional features in plutonium-dbg such as memory read speed are not worth the additional effort to setup the environment. However, it does achieve the goal of providing a GDB-like option for target binaries with anti-debugging code.

Without much trouble plutonium-dbg debugging was setup in a Debian QEMU-KVM instance running Linux kernel 4.17.0, as shown in Figure 25.

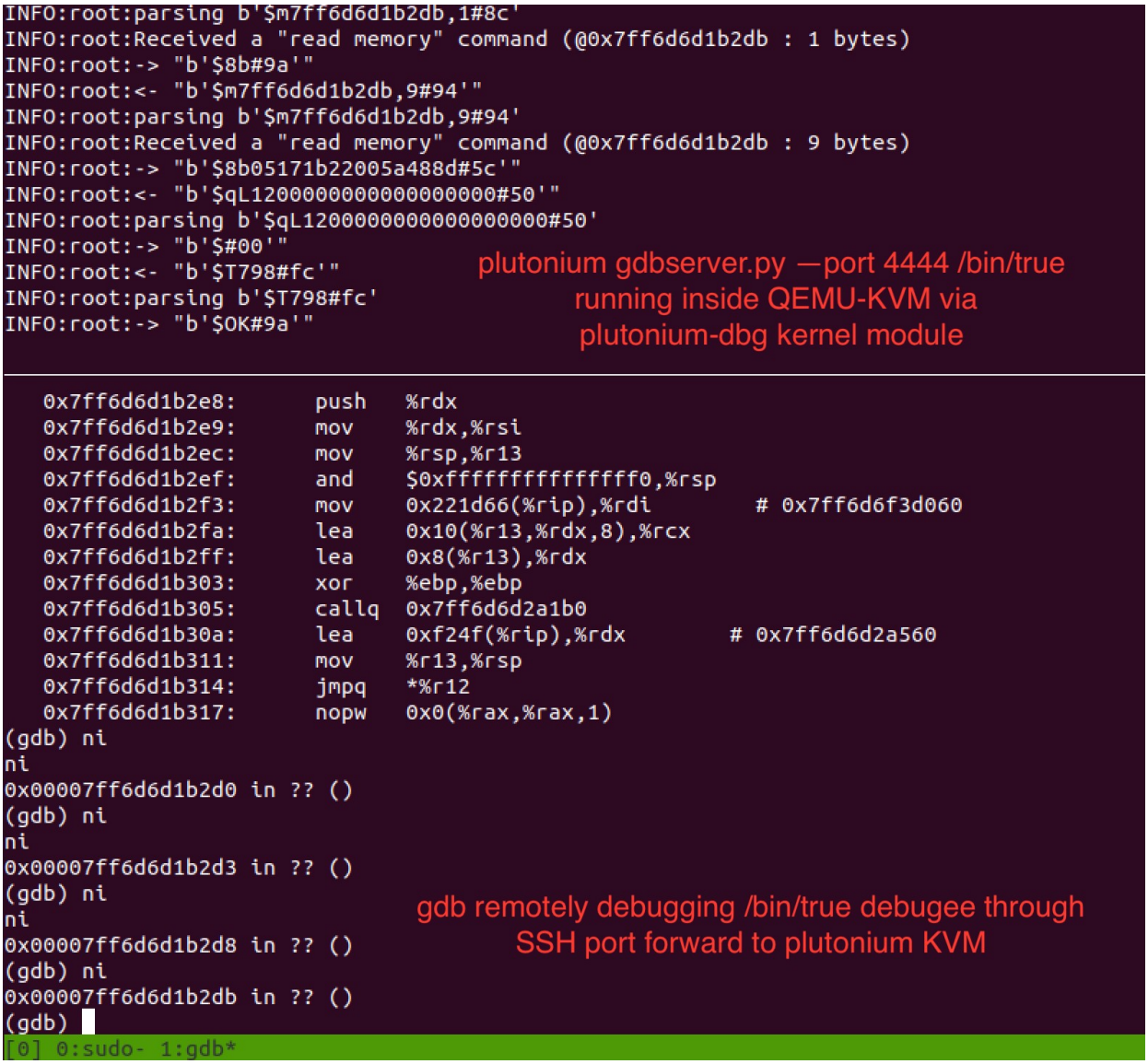


Figure 25: Using plutonium-dbg

4.3 Coverage Analysis

4.3.1 Lighthouse - Version 9.0 Release

Reference Link	https://github.com/gaasedelen/lighthouse https://blog.ret2.io/2020/04/29/lighthouse-v0.9/
Target Type	**Plugin:** IDA Pro; Binary Ninja **Coverage Data:** DynamoRIO; Intel Pin; Frida
Host Operating System	**IDA Pro Plugin:** Windows; Linux; macOS **Binary Ninja Plugin:** Windows; Linux
Host Architecture	*X86 (64)*
Initial Release	March 8, 2017
License Type	Open-Source
Maintenance	Maintained by Ret2 Systems

4.3.1.1 Updates

Lighthouse was covered previously in the December 2019 EOTA report. On April 29, 2020 a major update was released, which included full Python 2/3 compatibility, new coverage file formats, cross referencing coverage, user-defined themes, and official support for Binary Ninja 2.0 [78].

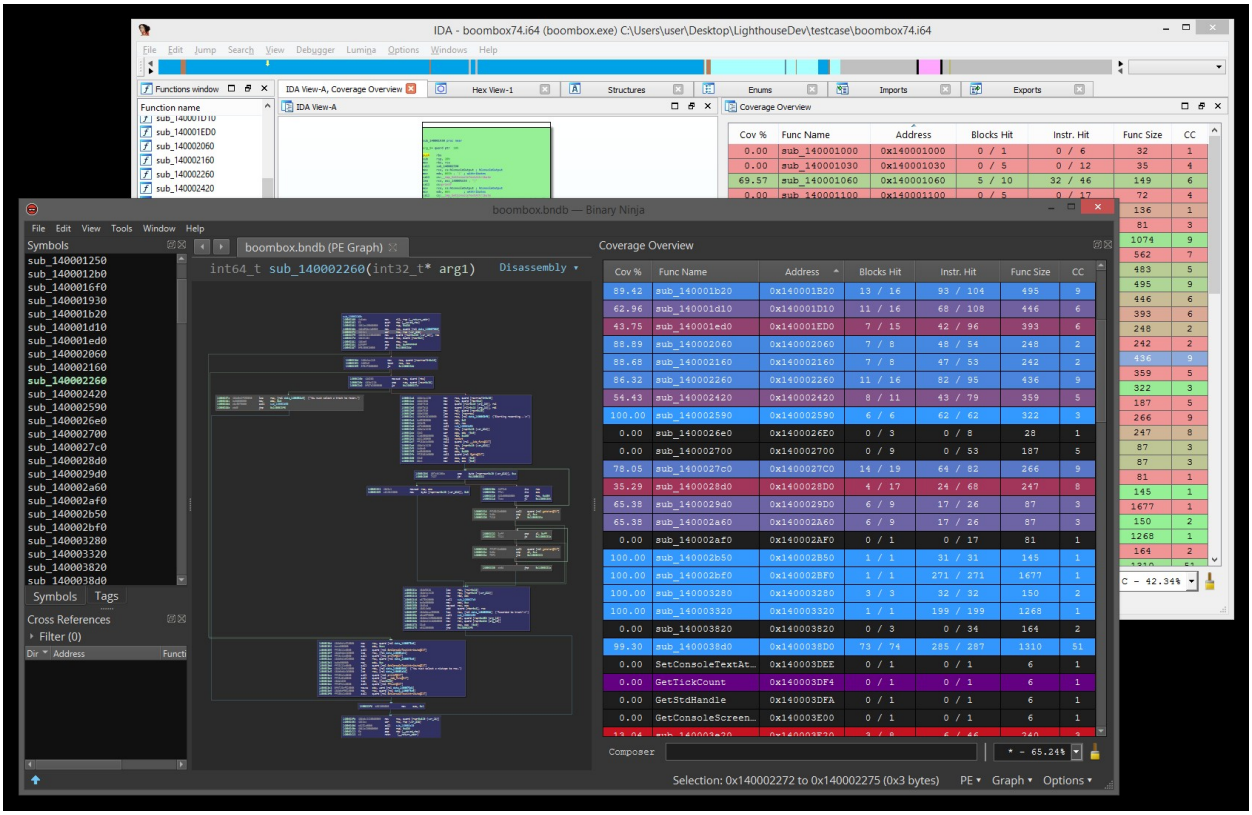


Figure 26 New theming support in Binary Ninja and IDA Pro

New Coverage Formats
Prior versions of Lighthouse relied on the DynamoRIO coverage format (drcov), which had the unintended consequence of forcing supporting tools to standardize around this format [78]. Lighthouse now supports simple formats that are easier to generate from custom tooling in `module name + offset` format (modoff) and absolute address formats. The advantages are that these are simple to implement, human readable, and portable, with each line of the trace representing either an instruction address or a basic block address.

```
0x14000419c
0x1400041a0
0x1400045dc
0x1400045e1
```

Figure 27 Absolute address format coverage file.

```
boombox.exe+0x3a06
boombox.exe+0x3a09
boombox.exe+0x3a0f
boombox.exe+0x3a15
```

Figure 28 Modoff format coverage file.

Coverage Cross-references (xref)
The new version of lighthouse now adds a context menu entry, "Coverage Xref." This will show all currently loaded coverage files that executed the selected code and allow quick switching to that coverage set.

4.3.2 bcov

Reference Link	https://github.com/abenkhadra/bcov https://blog.formallyapplied.com/2020/05/function-identification/ https://arxiv.org/pdf/2004.14191.pdf
Target Type	Binary
Host Operating System	Linux; macOS; Windows
Target Operating System	Linux; ELF Binaries
Host Architecture	x86 (64);
Target Architecture	x86 (64)
Initial Release	June 2020
License Type	Open-Source
Maintenance	Maintained by Formally Applied

Overview

bcov is a tool for efficient binary-level coverage analysis. It statically instruments x86-64 binaries to provide coverage reporting via patching. The data output for coverage will include the address, instruction count, whether it was executed, and if the basic block is a fall through or a branch. For a given function bcov is also able to dump the CFG, the predominator tree, the postdominator tree, and the superblock dominator graph [79]. The primary innovations in bcov are in techniques to improve the efficiency of coverage analysis. This includes porting Agrawal's probe pruning technique to binaries, a jump table CFG analysis technique called *sliced microexecution*, and optimized detour patching [80].

Design and Implementation

For an executable with symbols, identifying functions relies on reading the symbol table. For a stripped binary, various heuristics are used to identify functions. Many tools rely on control flow graph (CFG) based identification techniques because they provide better results than searching for common function prologs and epilogs. The basic mechanisms of CFG-based techniques include collecting the targets of direct calls to identify function entries, traversing a function's CFG to determine its end address, and tail-call analysis to look at jumps outside the current function to identify additional functions. There are some cases where CFG analysis techniques struggle, such as adjacent functions that have been merged into a single function if a tail-call from one function goes into an adjacent function. Another case is functions that do not return but end in some other way, such as a call to exit or abort. There is also undefined compiler behavior that may generate branching code where a branch is never intended, such as the *llvm_unreachable* assertion [81].

The bcov tool uses trampolines to statically insert probes into a binary in order to track basic block coverage. The probes are implemented as detours which redirect execution to a trampoline, which updates coverage data using a single pc-relative mov instruction. To make this work efficiently, transparently, and without compiler support required addressing the problems of probe pruning, precise CFG analysis, and static instrumentation. [80]

Probe Pruning

Instrumenting all basic blocks is both challenging and inefficient. Instead, bcov adopted Agrawal's probe pruning technique, where basic blocks are grouped into superblocks by using their dominator relationships. Any single basic block executed within a superblock implies that all basic blocks within the superblock have been executed. This drastically reduces instrumentation overhead and coverage data. [80]

Precise CFG Analysis

In order to avoid false positives, bcov employs several techniques to improve the precision of CFG analysis. bcov uses a novel technique called sliced microexecution for jump table analysis. bcov also implements a non-return analysis to eliminate CFG edges that are extraneous. Sliced microexecution works by testing each indirect jmp in a function according to the hypotheses in Figure 29. If the conditions fail, then the analysis is aborted and the jmp is categorized as a tail-call. Sliced microexecution works by backwards slicing to determine the boundary conditions, and then testing index values to determine if the jmp is bounded, and thus likely to be using a jump table, as invalid values should hit a default case within a jump table. [80]

	Hypothesis	Action
(1)	Depends on constant base address?	if yes test (2) else abort
(2)	Is constrained by a bound condition?	if yes test (3) else assume (4)
(3)	Bound condition dominates jump table?	if yes do recovery else assume (4)
(4)	Assume jump table is data-bounded	do recovery and try to falsify

Table 2: Hypotheses tested, or falsified, to analyze a jump table. Backward slicing answers #1 to #3. Microexecution is used to falsify hypotheses and recover the jump table.

Figure 29 Sliced microexecution hypotheses [80]

Static Instrumentation

For efficiency and ease of instrumentation, bcov will patch a single basic block within a super block. It make this decision based on the expected overhead of restoring control flow. Where possible, bcov will make use of padding bytes to insert detours, instrument jump table entries, and use larger neighboring basic blocks to host the detour of a smaller basic block [80].

bcov's design is optimized to provide program transparency, performance, and flexibility. It does not aggressively alter the program state, such as the stack, heap, and general purpose registers. The use of a single mov instruction to track coverage also means it has low overhead [80].

Use Cases and Limitations

bcov can be used as a stand-alone binary for some functionality, but it's primary functionality of dumping coverage data requires linking against the bcov-rt runtime or using LD_PRELOAD to inject it in at runtime. Coverage data is output on process exit or when the application receives a user generated signal. bcov is a viable and efficient tool to statically instrument binaries for code coverage, with less overhead than a DBI based tool like DynamoRIO. All instrumented binaries tested by the bcov authors were able to pass the test suites associated with those binaries, with no noticeable regressions. Additionally, it took about 30 seconds to instrument a 25MB binary. In testing, bcov had an average performance overhead of 14%, memory overhead of 22%, and file size overhead of 16%. This is a significant advantage over DBI based coverage tools like Pin and DynamoRIO, that can have a performance overhead of more than 10x, while bcov has roughly equal coverage tracking. [80]

The bcov paper also compared the effectiveness of its jump table analysis, and noted that BAP does not have built-in support for jump table analysis, while angr (v8.18.10) crashed on opencv

and llc binaries, and otherwise was less successful at identifying jump tables than bcov and IDA Pro. Test results indicated bcov is equal to or better than IDA Pro in jump table recovery. [80]

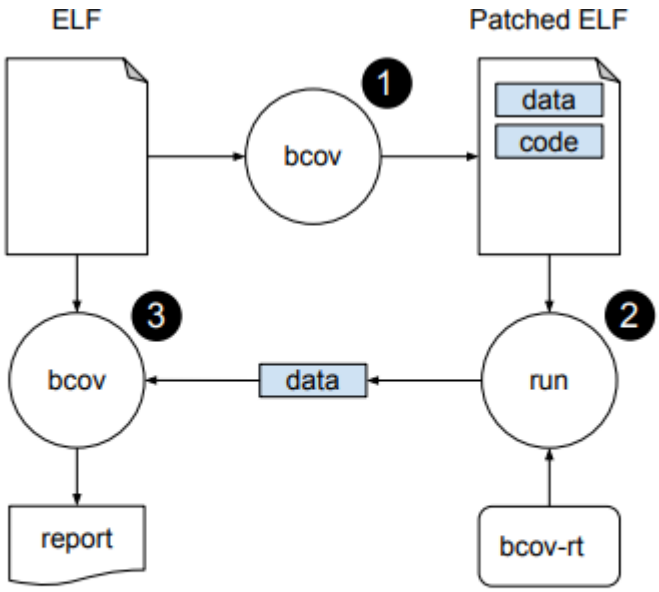


Figure 1: The general workflow of bcov. A binary is patched with extra code segment (trampolines) and data segment (coverage data). Our bcov-rt library dumps the data segment at run-time. In our prototype, reporting coverage requires re-analyzing the binary.

Figure 30 bcov workflow [80]

	Level	Coverage goal	Compiler independence	Performance overhead	Flexibility	Usability
gcov	source	complete	✗	✗	✗	✓
llvm-cov	source	complete	✗	✓	✗	✓
sancov	IR	heuristic	✗	n/a	✗	✓
Intel PT	binary	heuristic	✓	✓	✗	✗
drcov	binary	both	✓	✗	✓	✗
bcov	binary	both	✓	✓	✓	✓

Table 1: A comparison with representative coverage analysis tools. Compiler-dependent tools require modifying the build system and recompilation which limits flexibility. The usability of binary-level tools in the testing workflow is limited. In contrast, bcov only requires replacing a binary with an instrumented version.

Figure 31 bcov compared to other coverage tools [80]

bcov does have some limitations. It is not always successful at patching binaries without modifying the build. The bcov paper noted that a small linker change was sufficient to accommodate the programs that otherwise could not be patched. Additionally, bcov only supports x86-64, relies on invasive ELF file modification, and does not support other executable file formats. It is likely that bcov could be implemented for other architectures and file formats without much issue. The bcov jump table reconstruction is also not perfect, and neither is the non-return analysis, which can cause it to miss some coverage. Lastly, bcov does not work with self-modifying code. [80]

4.4 Frida 12.9 (Stalker update)

Reference Link	**https://frida.re/docs/stalker/**
Target Type	Binary, Java bytecode
Host/Target Operating System	Linux, Android, macOS, iOS, Windows, BlackBerry QNX
Host/Target Architecture	x86 (32, 64), ARM (32, 64)
Initial Release	September 2012
License Type	Open-Source
Maintenance	Maintained by @oleavr and NowSecure

Overview

Frida is a toolkit for dynamic instrumentation of binaries using code injection and hooking. Instrumentation commands are typically written in JavaScript and C when high performance injection is needed [82]. Most users of Frida will make use of the Interceptor API to do inline hooking of functions and methods to analyze function arguments and return values. A lesser known component of Friday is its Stalker engine, which enables researchers to answer questions about what actions a function is taking. For example, Stalker can be used to answer the common question "what other APIs are called for a given input?" Stalker can also be used to support fuzzing, measuring code coverage, and watching when execution diverges for specific inputs [83]. While Stalker was created years ago and did not get heavy use until Frida 10.5 (released late 2017). In December 2019, Stalker was given additional capabilities to follow native function calls, Objective-C methods, and Android Java methods [84]. The May 2020 release added support for ARM32 binaries, opening Stalker for use on more mobile and embedded platforms [85].

Design

The Frida Stalker engine was designed to be a code tracer that provides high granularity while minimizing its impact to the traced process. Stalker dynamically makes a copy of the original machine code and adds instrumentation between instructions.

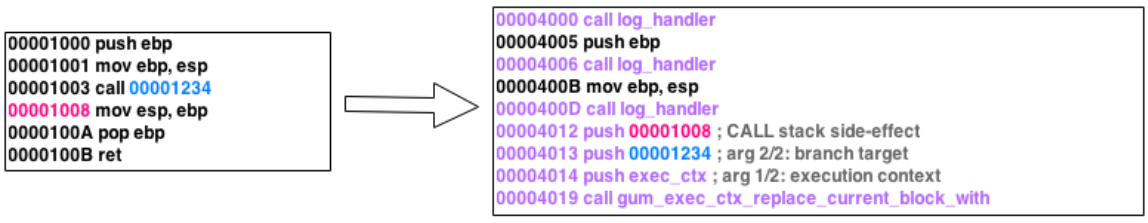


Figure 32 Stalker Instrumentation [86]

The researcher can specify whether they are interested in calls to a specific function, return values, or other highly granular specifications. The recompilation of modified code is done lazily to minimize the impact to the process being examined [83]. Doing the recompilation as necessary also allows Stalker to only instrument the instructions that the user is interested in, supporting Stalker's goal of minimizing impact to the traced process. When making a copy of the code, instructions are copied in batches up to the next branch instruction to limit how much code is modified at a time. [86]

Stalker was also written to trace programs containing anti-debugging features. The author, Ole André Vadla Ravnås, argues that software and hardware breakpoints can both be easily defeated by applications with anti-debugging code in them. Stalker leaves the original instructions unmodified, helping to defeat many anti-debugger techniques. Another Stalker feature for dealing with anti-debugger techniques is the ability to set a trust-threshold parameter on a per application basis. This parameter is used to decide whether a block needs to be recompiled and patched each time it is executed or if it can be permanently branched without causing problems to self-modifying code, a common feature seen in programs that implement anti-debugging techniques. [86]

Below is an example of using Stalker to gather a listing of all calls being made by a thread of interest. It shows how little JavaScript needs to be written to begin using Stalker. onCallSummary still requires printing and display code to show the user the enrichedSummary.

```
ThreadTracer.prototype.follow = function (thread) {
    return new Promise(function (resolve) {
        let threadId = thread.id;
        Stalker.follow(threadId, {
            events: {
                call: true,
                ret: false,
                exec: false
            },
            onCallSummary: function (summary) {
                const enrichedSummary = {};
                for (let address in summary) {
                    if (summary.hasOwnProperty(address)) {
                        enrichedSummary[address] = {
                            symbol: this._moduleMap.symbol(address),
                            count: summary[address]
                        };
                    }
                }
```

Figure 33 Stalker Example [87]

Use Cases and Limitations

Use cases mentioned by the Frida developers include in process fuzzing, code coverage analysis, hooking inline system calls, fault injection, and examining private APIs [84]. In process fuzzer capabilities are already provided by the frida-fuzzer.

There appear to be few specific limitations. Stalker has been tested on all of the supported platforms and has been programmed to deal with platform-specific problems, such as Pointer Authentication Codes (PAC) on iOS [88]. The large amount of API documentation [89], instructions [88], detailed blog write-ups [86], and regular updates suggest that Frida, and in particular Frida Stalker, is a mature tool that can help researchers try to quickly examine the inner workings of a thread of interest.

4.5 Trends

Dynamic analysis has seen a lot of innovation and maturity in recent years. The tools are becoming more powerful, transparent, and efficient. At the same time, they are becoming better integrated with platforms, such as Binary Ninja, and are optimizing for interoperability. There have been major updates to existing tools, such as Windbg, Lighthouse, and Frida, as well as the introduction of entirely new tools, such as the Binary Ninja Debugger Plugin, Plutonium-dbg, and bcov. Given the power of dynamic analysis, there is likely to be continued rapid improvements in this area.

5 Fuzzing

5.1 Technical Overview

Fuzzing is a form of software testing which feeds semi-random inputs into a computer program to trigger a crash or other unintended behavior. 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. Although simple in concept, an entire field of research has formed around finding more effective and faster methods to mutate or generate input.

Fuzzers and fuzzing techniques have evolved in the past three decades, and many 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 has proven to be an extremely effective method of finding flaws that consistently produces verified vulnerabilities (e.g., CVEs).

There are a variety of ways to fuzz program inputs, but these methods largely fall into two categories: mutation-based and generation-based. Mutation-based fuzzing, as the name suggests, mutates input based on coverage metrics from previous inputs or other information. Generation- based fuzzers generate inputs using a grammar, or some other generator.

White-Box, Grey-Box, and Black-box Fuzzing

A black-box fuzzer is one which lacks any knowledge of program internals. These fuzzers are easy to construct but are not the most effective at bug-finding. The most common type of fuzzer discussed in this report is called grey-box. It is driven by instrumentation, whether compiled in from source code or instrumented into the binary. A white-box approach is one which leverages knowledge of the program derived from its source code and uses program analysis techniques, such as symbolic execution and constraint solving.

Mutation-based Fuzzing

Relevant EotA Tools: TortoiseFuzz, IJON, Frizzer, Redqueen, Eclipser, honggfuzz, Angora, AFLGo, AFLFast

Mutation-based fuzzing does not require input to be formatted based on some predetermined grammar. Instead, it takes example inputs as seed values, and mutates them to create input for the program. In a black-box scenario mutation is done blindly. In a grey-box scenario the program binary is instrumented to glean coverage and other information, which the fuzzer then uses to mutate the seeds, often to find new paths. White-box fuzzing also works this way, but can also take advantage of other program analysis techniques.

One well-known mutation-based fuzzer is AFL [90]. AFL's operations are instructive for understanding mutation-based fuzzers more generally. Its instrumentation consists of inserting instructions which record branch decisions and allow AFL to collect path coverage information. These are used to watch for interesting behavior such as new paths traversed. AFL is seeded with a corpus of possible inputs that are then mutated based on the coverage information gained from the instrumentation. Inputs which perform 'interestingly' are then saved to be used as new seeds. Seeds are mutated in several ways, including bit-flipping, inserting 'interesting' integers such as INT_MAX and adding and subtracting small integers from the seed. AFL's overall workflow can be seen in Figure 34.

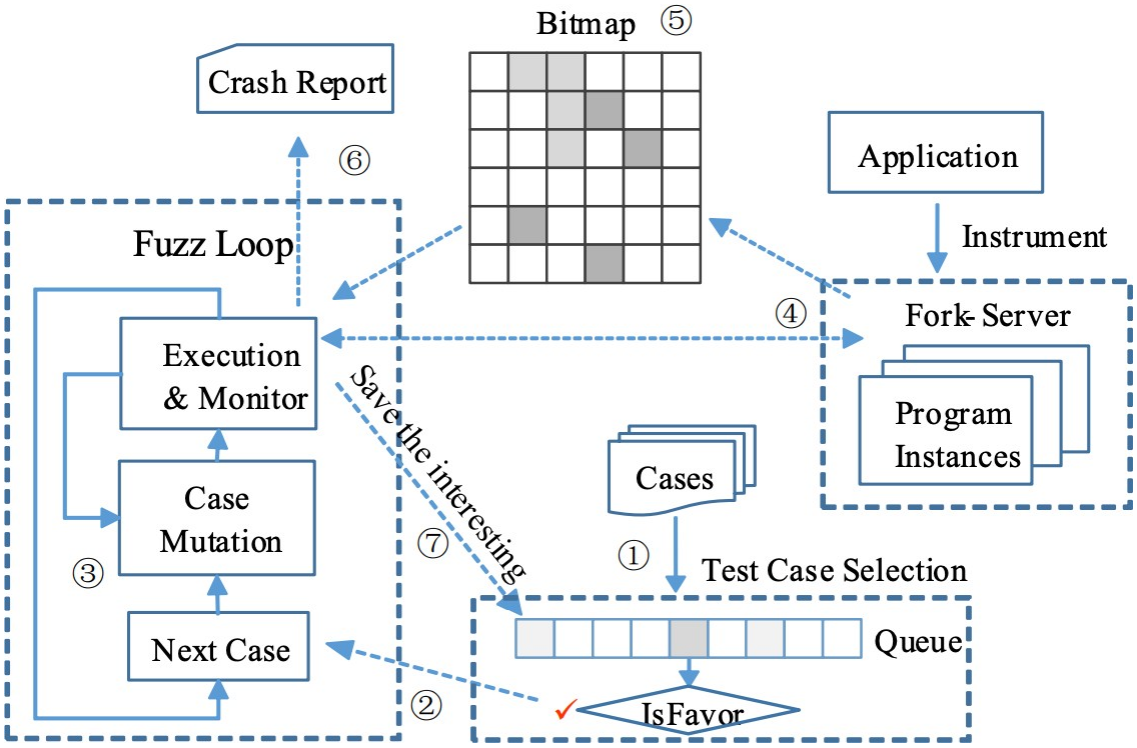


Figure 34: Diagram of AFL workflow [91, p. 2]

Several mutation-driven fuzzers are included in this section, including TortoiseFuzz [91], which is a variant of AFL.

Generation-based Fuzzing

Relevant EotA Tools: Nautilus, F1 Fuzzer, Fuzzilli, CodeAlchemist

Generation-based fuzzing uses a generator, often in the form of a grammar, to generate fuzzer inputs. Generation-based fuzzers are useful for fuzzing programs with complex input formats. In the case of grammar-based fuzzers, a grammar is designed to model and randomly generate syntactically correct inputs. An example grammar is shown in Figure 35.

⟨*START*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*term*⟩
| ⟨*term*⟩ '+' ⟨*expr*⟩
| ⟨*term*⟩ '−' ⟨*expr*⟩

⟨*term*⟩ ::= ⟨*factor*⟩
| ⟨*factor*⟩ '*' ⟨*term*⟩
| ⟨*factor*⟩ '/' ⟨*term*⟩

⟨*factor*⟩ ::= ⟨*integer*⟩
| ⟨*integer*⟩ '.' ⟨*integer*⟩
| '+' ⟨*factor*⟩
| '−' ⟨*factor*⟩
| '(' ⟨*expr*⟩ ')'

⟨*integer*⟩ ::= ⟨*digit*⟩ ⟨*integer*⟩
| ⟨*digit*⟩

⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Figure 35: Example of a grammar used with a grammar-based fuzzer [94]

Generation-based fuzzers are not simply limited to using grammars. The first version of this report highlighted two JavaScript Just-In-Time (JIT) Engine fuzzers that adopted different techniques towards generating semantically and syntactically correct JavaScript inputs [28], [95]. Fuzzilli [28] is one such fuzzer that is built on a custom IR (i.e., FuzzIL) that can be lifted to JavaScript.

Generation-based fuzzing differs from mutation-based in both operation and intent. Whereas mutation-based fuzzers are intended to find crashes caused by any input, regardless of whether that input is formatted correctly for the program, generation-based fuzzers look for errors caused by inputs that are formed correctly. Generation-based fuzzers do not receive any coverage-based feedback, or other instrumentation information. Their advantage is that they do not have to search for correctly formatted inputs, and they can more easily target programs with complicated protocols or input formats.

As discussed further in Trends, grammar-based fuzzers can be incredibly slow, which limits their effectiveness. Generation-based fuzzers also lack the code coverage benefits of mutation-based fuzzer. Nautilus [96] is a fuzzer discussed in this section that attempts to bridge that divide.

Hybrid Fuzzing

Relevant EotA Tools: QSYM

Hybrid fuzzing leverages the benefits of fuzzing (less time and memory complexity) with those of symbolic execution (knowledge of the program, such as path constraints). This is a form of white- box fuzzing, in which the fuzzer is assisted by symbolic execution. This method has been implemented in a variety of fuzzers, many of which competed in the DARPA Cyber Grand

Challenge (CGC) [98]. However, hybrid fuzzing often suffers from the same performance issues as symbolic execution. A recent, and arguably successful, implementation of the technique can be seen in the QSYM fuzzer [69], which was discussed in the first version of this report.

Platform-Specific Fuzzing

Often, the greatest barrier to fuzzing is not the fuzzer itself but instrumenting the target program. Certain targets, such as kernels and hypervisors, present hurdles to fuzzing due to their program structure and runtime behavior. Two recent attempts at kernel fuzzing are kAFL [99] (discussed in the second version of this report) and HFL [100] (discussed in this section). This section also covers the recent hypervisor fuzzer, Hypercube [101]. Other targets with non-x86 architectures present issues due to their lack of standardization. This is something addressed by AFL's QEMU mode and fuzzers like AFLUnicorn (discussed in the first version of this report).

Measuring Fuzzer Performance

How to best measure the efficacy of a fuzzer has been a frequent discussion among fuzzing researchers, especially with respect to mutation-based fuzzers. The ability to effectively determine the performance of a fuzzer relative to others has been limited by the variance and lack of consensus on fuzzer metrics. Many fuzzing papers will test on different software, for different amounts of time, and count different kinds of results as a success. The 2018 paper, *Evaluating Fuzz Testing* [102], examined how a spate of recent fuzzing research was evaluated, and recommended certain testing standards. Many subsequent fuzzing papers have followed this paper's advice, including tools discussed in this report. However, more work is required to determine how best to measure the performance of fuzzers, and there is far from a consensus in the community.

5.2 Mutation-based Fuzzing

5.2.1 TortoiseFuzz

Reference Link	**https://github.com/TortoiseFuzz/TortoiseFuzz**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	Tool not yet released; Paper released Jan 2020
License Type	N/A - Not yet released
Maintenance	N/A - Not yet released

Overview

TortoiseFuzz is a grey-box fuzzer designed to find memory corruption vulnerabilities by prioritizing coverage with security sensitive operations. This fuzzer is based on the NDSS 2020 paper, *Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization* [91].

This research addresses the fact that current coverage-based fuzzers treat all paths equally, regardless of whether they have exploitable characteristics. TortoiseFuzz proposes a method of *coverage accounting* that privileges inputs that generate coverage with sensitive memory operations.

TortoiseFuzz is a simple modification to AFL that culls the queue based on their coverage accounting method. The author's eschewed memory intensive techniques like taint analysis and symbolic execution, to focus on several coverage accounting heuristics that evaluate which paths are more vulnerable than others. In evaluating TortoiseFuzz, the authors found it had comparable performance to state-of-the-art fuzzers, including QSYM. However, TortoiseFuzz's scope is specifically limited to memory corruption vulnerabilities, which introduces certain biases that more general fuzzers lack.

Design and Implementation

TortoiseFuzz is designed to direct fuzzing towards memory sensitive regions by adding queue culling AFL. TortoiseFuzz uses three metrics to determine what inputs to cull and how to prioritize the ones that are left:

1. **Function Calls:** Using the Common Vulnerabilities and Exposures (CVEs) information for disclosed vulnerabilities over the last four years. The top 10 functions included the glibc functions *memcpy()*, *malloc()*, *free()*, *memmove()*, and *memset()*. Edges are measured by how many vulnerability-involved functions are in the edge's destination basic block.

2. **Loops:** Loops are identified as being closely associated with memory vulnerability due to their propensity for overflows, among other errors. The authors use CFG-level instrumentation to identify loops by searching for back edges. Once the loops have been identified, inputs that encounter them are prioritized.

3. **Basic Blocks:** Edges are also evaluated by the number of instructions with memory operations in their destination basic block.

TortoiseFuzz modifies AFL in two ways. Instrumentation allows the fuzzer to evaluate edges on the metrics previously discussed. The queue culling then happens at runtime. Figure 36 shows the AFL algorithm with the functions modified by TortoiseFuzz in grey.

Algorithm 1 Fuzzing algorithm with coverage accounting

```
1:  function FUZZING(Program, Seeds)
2:      P ← INSTRUMENT(Program, CovFb, AccountingFb)        ▷ Instr. Phase
3:      // AccountingFb is FunCallMap, LoopMap, or InsMap

4:      INITIALIZE(Queue, CrashSet, Seeds)
5:      INITIALIZE(CovFb, accCov, TopCov)
6:      INITIALIZE(AccountingFb, accAccounting, TopAccounting)
7:      // accAccounting is MaxFunCallMap, MaxLoopMap, or MaxInsMap
8:      repeat                                              ▷ Fuzzing Loop Phase
9:          input ← NEXTSEED(Queue)
10:         NumChildren ← MUTATEENERGY(input)
11:         for i = 0 → NumChildren do
12:             child ← MUTATE(input)
13:             IsCrash, CovFb, AccountingFb ← RUN(P, child)
14:             if IsCrash then
15:                 CrashSet ← CrashSet ∪ child
16:             else if SAVE_IF_INTERESTING(CovFb, accCov) then
17:                 TopCov, TopAccounting ←
18:                     UPDATE(child, CovFb, AccountingFb, accAccounting)
19:                 Queue ← Queue ∪ child
20:             end if
21:         end for
22:         CULL_QUEUE(Queue, TopCov, TopAccounting)
23:     until time out
24: end function
```

Figure 36: Algorithm of AFL with TortoiseFuzz modifications in gray [91, p. 6]

Use Cases and Limitations

TortoiseFuzz was evaluated in an exhaustive process that tested it against 6 others fuzzers, including grey-box fuzzers like AFLFast and AFL, and hybrid fuzzers like QSYM and Angora. They ran each fuzzer on 30 different applications for 140 hours and repeated all experiments 10 times. This is a total of 42,000 hours per fuzzer, or almost 300,000 hours total. The thoroughness of this testing reflects positively on the validity of their results. From the evaluations, the authors found that TortoiseFuzz outperformed the grey-box fuzzers and achieved a comparable result to QSYM.

TortoiseFuzz has a narrower scope and relies solely on heuristics, unlike the general fuzzers it was evaluated against. The identification of vulnerable functions is purely empirical and done with inexact methods like scraping CVE descriptions and then parsing them for call-stacks for vulnerable functions. The limitation of this approach is that many CVEs do not have call-stacks in their descriptions. The authors chose not to investigate the efficacy of each of their metrics independently, which makes it difficult to identify which were more useful. However, they do recognize the limitations of their approach and state that their plan is to perform coverage accounting in a more systematic and quantifiable way.

5.2.2 IJON

Reference Link	**https://github.com/RUB-SysSec/ijon**
Target Type	Source; Binary
Host Operating System	Linux
Target Operating System	--
Host Architecture	X86 (32, 64)
Target Architecture	x86 (32, 64); ARM (32, 64); MIPS (32, 64); MIPS (32, 64)
Initial Release	2020
License Type	Open-Source
Maintenance	Ruhr University Bochum – Systems Security

Overview

IJON is an extension for AFL-based fuzzers to facilitate exploration of difficult to reach code through hints provided by an analyst. With the insertion of one to two annotations in the source code of a target program, hints influence a fuzzer's feedback loop to navigate program states more effectively [103]. New program states are realized as additional code coverage, rewarding a fuzzer's advancement of each new state in addition to branch traversal code coverage. Analysts can leverage IJON's feature set to overcome difficult to reach code by specifying the importance of specific values of the internal state of a program within a new AFL instance. Once the hard to reach code is found, the new AFL-IJON instance communicates the additional code coverage and inputs used to reach it to the main AFL session.

Design and Implementation

IJON is designed to be used as a "human in the loop" fuzzer extension. To date, there are IJON implementations for AFL, AFLFAST, LAF-INTEL, QSYM, and ANGORA. The modifications are to support two human interactions: 1) observe a stuck fuzzing job 2) and then intervene appropriately. The workflow for the analyst is to identify relevant portions of the code, store additional and relevant values in AFL's bitmap, modify the fuzzer's feedback function with annotations, and minimize space exploration through goal creation [103].

```
IJON_DISABLE();
//...
if(x<0)
    IJON_ENABLE();
```

Figure 37: Example annotations of IJON_DISABLE and IJON_ENABLE

For an analyst to guide the fuzzer, annotations are directly inserted into the source code, as demonstrated in Figure 37. Here, two annotations are used to direct the feedback loop to focus on coverage to a specific code region. With the *IJON_ENABLE()* annotation inside the *if* condition, fuzzing input is restricted to only values that satisfy the condition, reducing the search space of uninteresting paths [104]. If a value to a reference variable in the program is determine to be interesting when changed, then the *IJON_INC()* annotation can be used to track and reward the fuzzer's feedback loop. Likewise, the *IJON_SET()* annotation rewards the fuzzer for new coordinates achieved in a maze game as shown in Figure 38 [103]. This maze contains 4 different branches that are easily exhausted by control flow graph code coverage, but that encompass many states needed to solve for the correct path. Moreover, the *IJON_STATE()* annotation is used to create a virtual state for exploring combinations of variables, and the *IJON_MAX()* allows for rewarding the fuzzer to maximize one or more values.

```
while(true) {
    ox=x; oy=y;
    IJON_SET(hash_int(x,y));
    switch (input[i]) {
        case 'w': y--; break;
        case 's': y++; break;
        case 'a': x--; break;
        case 'd': x++; break;
    }

    if (maze[y][x] == '#'){ Bug(); }

    //If target is blocked, do not advance
    if (maze[y][x] != ' ') { x = ox; y = oy; }
}
```

Figure 38: IJON annotate maze game

Use Cases and Limitations

For targets with source code, IJON enabled fuzzers perform better than the corresponding unmodified versions [103]. In Figure 39, the Trusted Platform Module (TPM) library called LIBTPMS is fuzzed with both AFL and AFL-IJON. The figure illustrates how much more exploration coverage IJON is able to achieve with up to 32x improvement using two annotations
[104] [103]. An advantage of IJON is that there is no loss of features over the unmodified AFL fuzzer. IJON can be used as a supplement for the difficult to overcome roadblocks that fuzzing often encounters and introduces a mechanism for analyst to focus on interesting paths.

Figure 39: Exploration comparison of AFL and AFL-IJON on a LIBTPMS (Trusted Platform Module) [104]

5.3 Generation-based Fuzzing

5.3.1 Nautilus

Reference Link	**https://github.com/nautilus-fuzz/nautilus**
Target Type	Source
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2020
License Type	Open-Source
Maintenance	Not Actively Maintained

Overview

Nautilus is a fuzzer that attempts to combine the benefits of tailored inputs from grammar-based fuzzers with the code coverage information of mutation-based fuzzers. The paper *NAUTILUS: Fishing for Deep Bugs with Grammars* [96], describes it as a grammar-based fuzzer that initially generates inputs based on a grammar and then performs mutations based on code coverage and the grammar. Nautilus has won several bug-finding plaudits, including against Microsoft Edge's ChakraCore and Ruby.

Nautilus is an actively maintained project that recently released Version 2.0. It is implemented in Rust and can also be used in combination with AFL, where AFL will import Nautilus inputs as it runs. A grammar can also be defined in Python. [96] [105] [106]

Design and Implementation

Nautilus employs a multistep process that generates inputs from a grammar and then mutates them based on coverage feedback. Figure 40 presents an overview of Nautilus. The numbered steps in the diagram are described below.

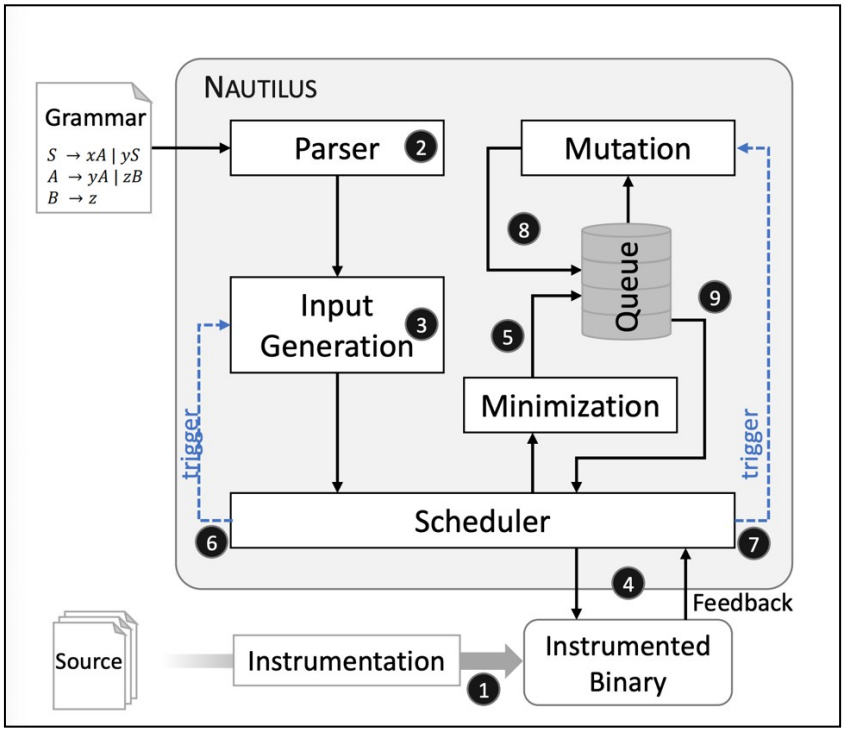


Figure 40: Nautilus Overview [96, p. 4]

0. **Nautilus Inputs:** The source code and the input grammar are input into the fuzzer.
1. **Compile Target Code:** In order to get coverage feedback, the code has to be compiled with Nautilus's compiler.
2. **Grammar Parsing:** Nautilus parses the input grammar given.
3. **Input Generation:** Nautilus generates 1000 initial random inputs and sends them to the scheduler.
4. **Checks Code Coverage:** Nautilus checks if these initial inputs generate new code coverage, using the instrumented binary compiled in step 1.
5. **Minimization:** Any input that covers new paths is then minimized into a new grammar and added to the queue of inputs for mutation.

6. **Scheduler Decision:** The scheduler decides whether the inputs in the queue can continue to be mutated and produce new paths, if it can't it triggers new input generation, and goes back to step 3.
7. **Mutation:** If new inputs can be mutated, it performs mutations based on the grammar.
8. **Mutated Input:** The mutated input is added to the queue.
9. **Fuzzing:** The queue is used to fuzz the instrumented binaries.

Compilation: In step 2, Nautilus uses an AFLPlusPlus to compile the source code.

Grammars: Grammars can be written in Python. An example grammar for generating XML code is shown in Code 2. This can be found in the repository along with examples for JavaScript, Lua, PHP, and Ruby.

```
#ctx.rule(NONTERM: string, RHS: string|bytes) adds a rule NONTERM->RHS.
# We can use {NONTERM} in the RHS to request a recursion.
ctx.rule("START","<document>{XML_CONTENT}</document>")
ctx.rule("XML_CONTENT","{XML}{XML_CONTENT}")
ctx.rule("XML_CONTENT","")

#ctx.script(NONTERM:string, RHS: [string]], func) adds a rule NONTERM->func(*RHS).
# In contrast to normal `rule`, RHS is an array of nonterminals.
# It's up to the function to combine the values returned for the NONTERMINALS with any fixed content used.
ctx.script("XML",["TAG","ATTR","XML_CONTENT"], lambda tag,attr,body: b"<%s %s>%s</%s>"%(tag,attr,body,tag) )
ctx.rule("ATTR","foo=bar")
ctx.rule("TAG","some_tag")
ctx.rule("TAG","other_tag")
ctx.regex("TAG","[a-z]+")
```

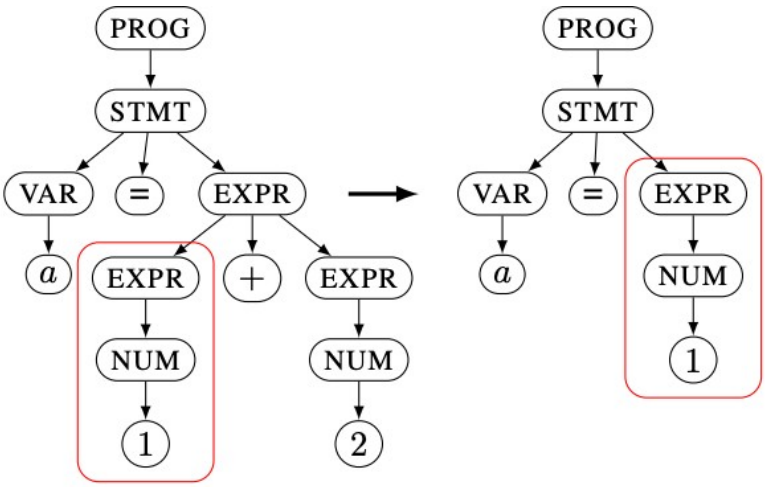
Code 2: Example grammar for XML, written in Python [106, p. README.md]

Generation can be done either through naïve generation or uniform generation. In the former, rules are picked randomly. For example, if there are two rules at a branch each will be picked approximately 50% of the time. In the latter, all possible trees of a certain depth are constructed and used for generation. Uniform generation allows CFG bias to be mitigated.

Minimization: After a newly generated input is shown to find new paths (Steps 1- 4) it is minimized in Step 5. There are two minimization strategies used:

- **Subtree Minimization:** Subtree minimization happens by sequentially visiting each node in the input's Abstract Syntax Tree (AST) and replacing its subtree with the smallest possible subtree. If an input generated from this still causes the new paths to be taken, it is kept. Otherwise, it is reverted to the original subtree.

- **Recursive Minimization:** This is performed after Subtree minimization and works to reduce the number of recursions that happen in an input. An example of recursive minimization given in the paper can be seen in Figure 41.

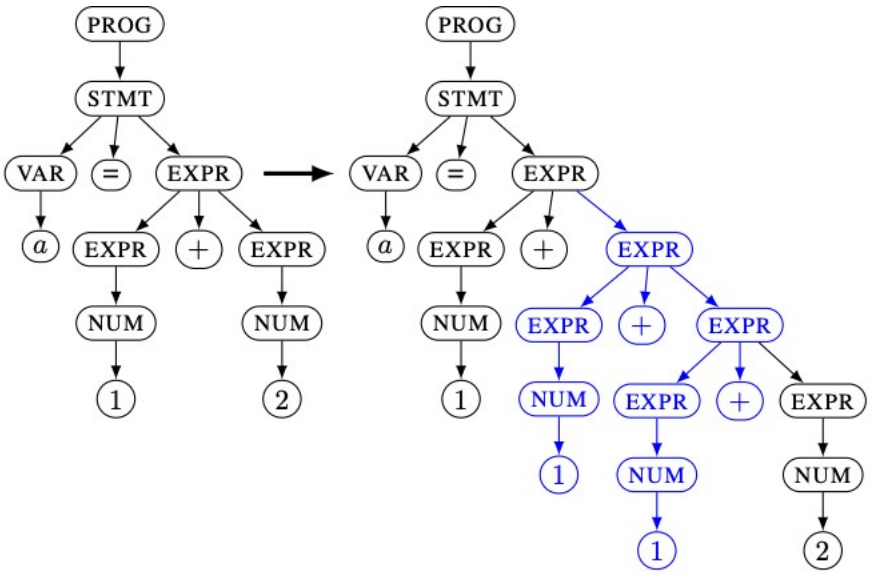


Example IV.3. Recursive Minimization is executed on `a = 1 + 2`, which contains a recursive `EXPR`: both the whole right-hand side, as well as the individual numbers, are derivable from `EXPR`. Using either of the two numbers instead of the addition yields a valid minimized tree.

Figure 41: Example of Recursive Minimization [96, p. 5]

Mutation: There are several kinds of mutation that Nautilus performs at Step 8:

- **Random Mutation:** Randomly identifies a node in the AST of the input and generates a new subtree at that node.
- **Rules Mutation: "**Sequentially replaces each node of the input tree with one subtree generated by all other possible rules."
- **Random Recursive Mutation:** Randomly identifies a recursion of an input's AST and repeats that recursion 2^{nn} times. An example of this kind of mutation can be seen in Figure 42.
- **Splicing Mutation:** This mutation "combines inputs that found different paths by taking a subtree from one interesting input and placing it in another input."
- **AFL Mutation:** Performs AFL mutations.



Example IV.4. This tree contains a recursion (an `EXPR` node has `EXPR` child nodes). Random Recursive Mutation randomly repeats this subtree recursion (two times in the example) and inserts the result in the already existing recursion. This turns the simple `a = 1 + 2` into the more complex `a = 1 + (1 + (1 + (1 + 2)))`.

Figure 42: Example of Random Recursive Mutation [96, p. 5]

Fuzzing: Fuzzing is performed on the instrumented binary. Figure 43 show the workflow of the fuzzing phase.

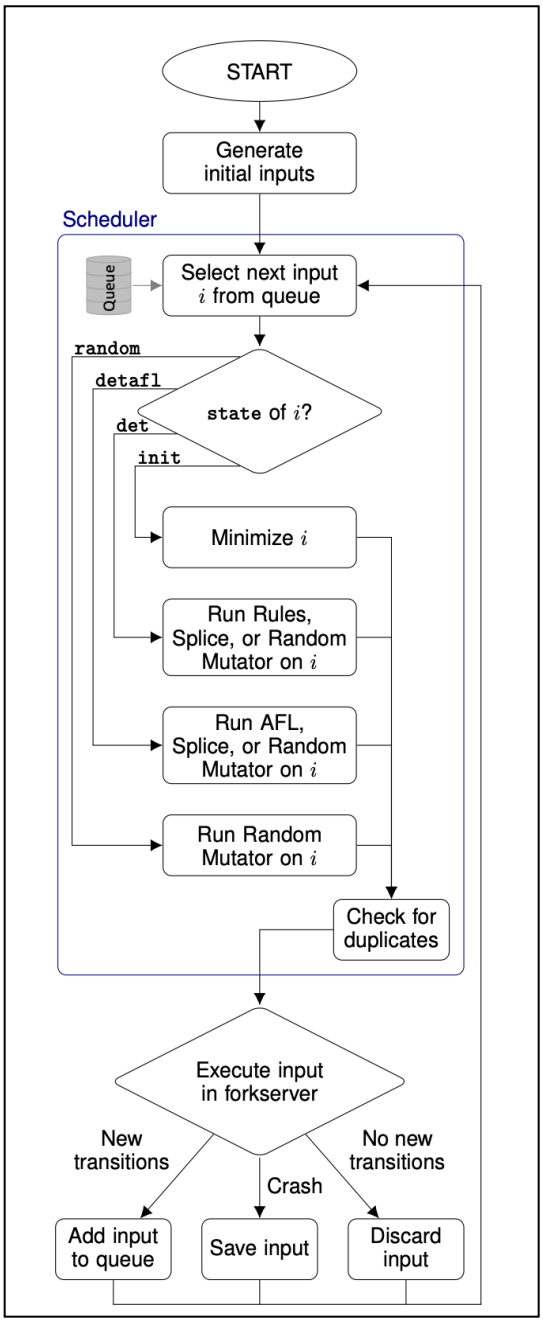


Figure 43: Nautilus Fuzzer [96, p. 7]

Implementation: Nautilus is written in Rust and requires Cargo and the Rust nightly build to run properly. It also requires AFLPlusPlus. When run, Nautilus has a similar terminal output as AFL, an example of which is shown in Figure 44.

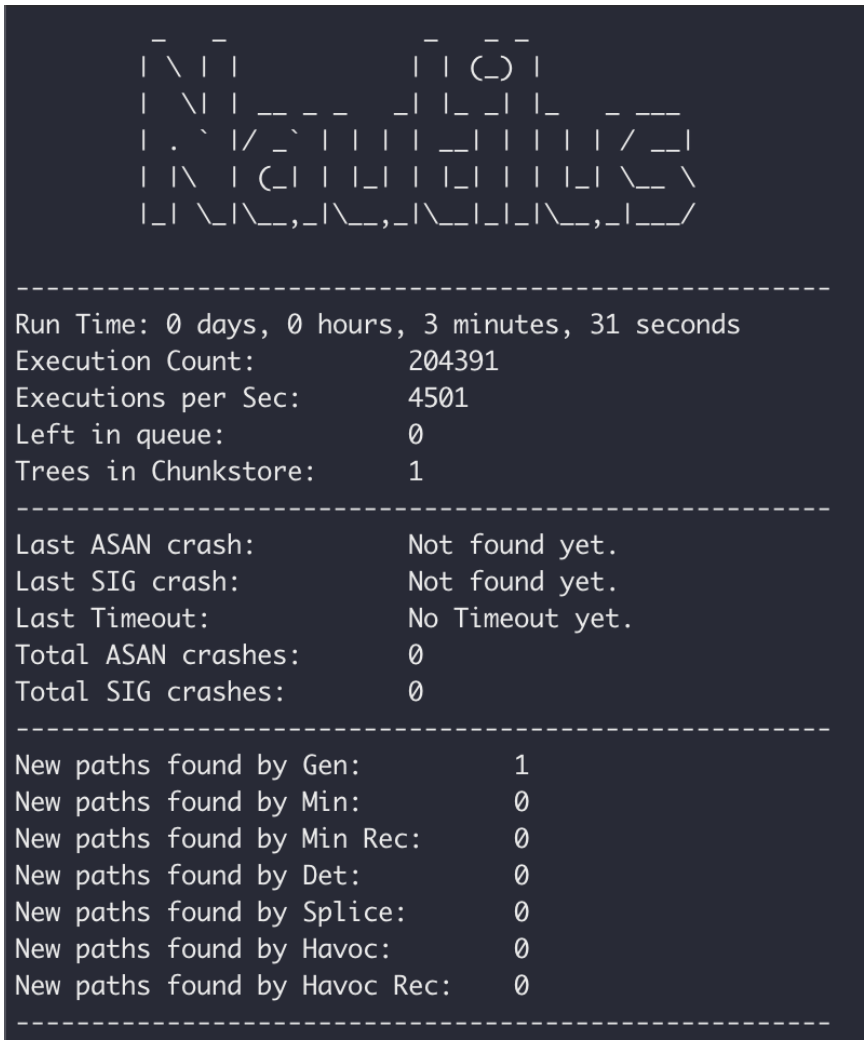


Figure 44: Nautilus Terminal Stats Screen

Use Cases and Limitations

Nautilus has already been used to find bugs in several major projects like the JavaScript JIT engine ChakraCore, which is part of Microsoft's Edge browser. Generation-based fuzzers are particularly applicable to Just-In-Time Compilers and other parsing, compilation, and rendering engines. The ability to specify a grammar is useful to fuzz for bugs that are not the result of malformed inputs.

The Python-based grammar specification and the integration with AFL are nice usability features of this tool. However, Nautilus requires relatively new features of Rust which may not be available in the current stable Rust release, a fact that is absent from Nautilus's build instructions. Compiling and running on a newly installed Ubuntu 18.04 distribution required several adjustments to the given instructions (including running with the Rust nightly release).

Input generation is time-intensive and slower that just random input mutation. However, the tight integration with AFL will somewhat mitigate this issue. If the user needs only inputs generated from a grammar, and not mutated ones, they should use a different grammar based fuzzer. The F1 fuzzer (or the Rust version, the FZero fuzzer) are grammar-based fuzzers that produce very fast inputs but do not include the specific mutation strategies used in Nautilus.

5.4 Platform Specific Fuzzing

5.4.1 HFL

Reference Link	**https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing- on-the-linux-kernel/**
Target Type	Source (Linux Kernel)
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	Tool not yet released; Paper released Jan 2020

Overview

Hybrid Fuzzing on the Linux Kernel (HFL) implements novel methods to bring hybrid (concolic) fuzzing to the Linux kernel. Described in the NDSS paper *HFL: Hybrid Fuzzing on the Linux Kernel,* seeks to take a technique that has been successfully applied to user space, to the far more complicated kernel space. Here, concolic fuzzing describes an approach that combines symbolic execution with fuzzing. Concolic analyses have been applied to user space programs, but thus far, have been limited to subsections of the Linux kernel. This due to three kernel specific characteristics: indirect control flow resulting from system calls and module interfaces, system state maintained by dependencies between system calls, and nested structures passed as arguments to the system calls. HFL introduces novel techniques that unravel indirect control flow, as well as maintain proper kernel state and argument structures during execution. It leverages modified versions of Syzkaller and QEMU as the concrete execution engine, S2E as the symbolic execution engine, GCC for compiler instrumentation, and LLVM Linux for static analysis on LLVM bitcode. [100]

Design and Usage

HFL must mitigate the above characteristics to increase concolic analysis efficacy. Each of the characteristics and their mitigations are described below.

Indirect Control Flow: The Linux kernel uses function pointers to modularize system calls and hardware modules. This indirect control flow proves difficult for concolic analysis because the concrete engine cannot efficiently select the correct function pointer, while the symbolic engine cannot efficiently constrain any specific pointer. HFL addresses this by translating indirect control flow to direct control flow by converting table look ups into branching conditionals. The translator "keeps track of how syscall parameters are propagated by performing inter-procedural dataflow analysis [100, p. 1]," subsequently inserting conditionals based on feasible function pointer values. An example of changes to kernel source code is in Code 3.

```
// before translation
ret = ucma_table[hdr.cmd](...);

// after translation
if (hdr.cmd == RDMA_CREATE_ID)
  ret = ucma_create_id (...);
else if (hdr.cmd == RDMA_DESTROY_ID)
  ret = ucma_destroy_id (...);
...
```

Code 3: Changes to kernel source code before and after translation [100, p. 6]

System State Maintenance: System calls transition the kernel through various states that may be order dependent. Inappropriate transitions between states prevent efficient exploration of the kernel code space. HFL infers the order of system calls through a two-step process. The first is inter-procedural analysis to identify *candidate dependency pairs* that operate on the same data (e.g., reading and writing to the same memory). The second is concolic analysis of these candidate dependency pairs to generate sequence rule sets. HFL "symbolically executes [the] instructions of a candidate dependency pair, [then] checks if these access the same address [100, p. 8]." If so, HFL marks these as *true dependency pairs* and infers an order. This approach has the added benefit that true dependency pairs operate on a common object in different contexts, allowing for greater constraint and relationship definition between the object and its members. Figure 45 describesthe system call dependency inference system.

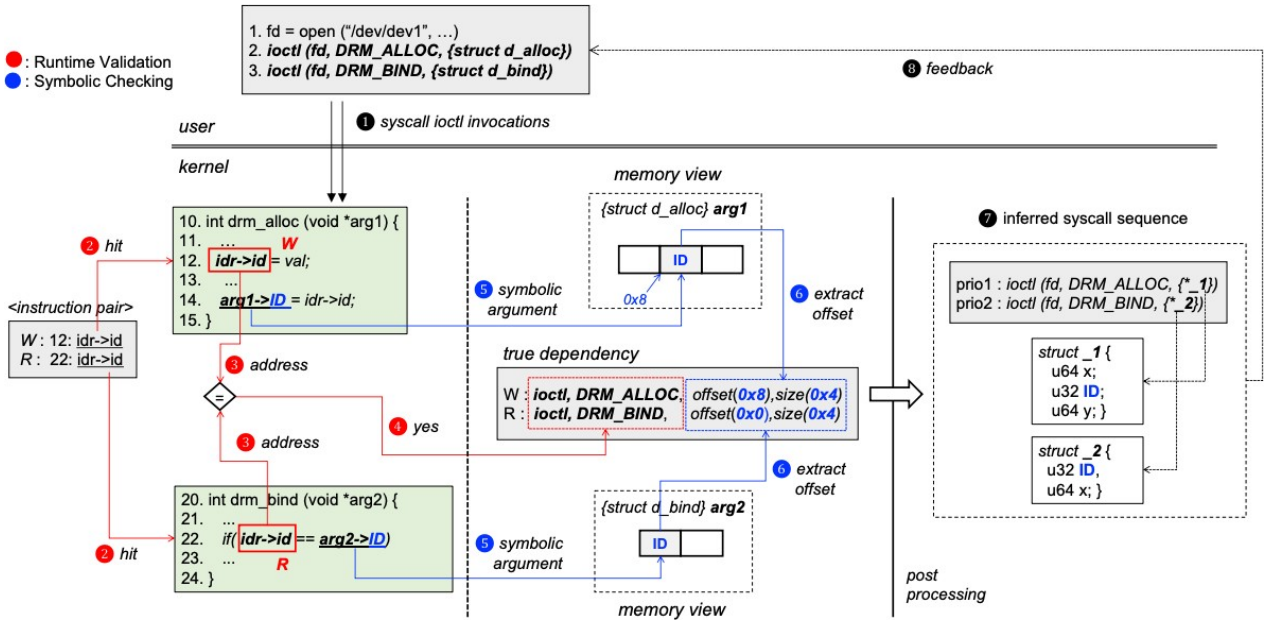


Figure 45: HFL system call dependency interference system [100, p. 9]

Algorithm for Figure 45:

1. Concolically analyze system call invocation
2. Identify which instructions pair to the same memory addresses
3. If they share the same address, mark this as a true dependency pair
4. Use the symbolized memory to identify dependencies on memory regions
5. Collect more constraints and relationships between structure and members
6. Define a dependency rule set
7. Infer order from the rule set

Nested Structures in Arguments: System calls that operate on user data may transfer information using dynamic structures and variables. Further, there may be relationships between these argument structures and variables - a dynamically allocated buffer may also define a length attribute. The kernel handles these cases by looping through the arguments and individually copying each from user to kernel space. Concrete and symbolic fuzzing engines struggle to generate complex nested structures.

HFL addresses nested structures by monitoring data transfer functions between kernel and user space. The key pieces of information are "memory locations [that connect] nested … structures and the length of [those] arguments [100, p. 8] ." Figure 46 and the following algorithm describe how system call arguments are retrieved in the case of ioctls.

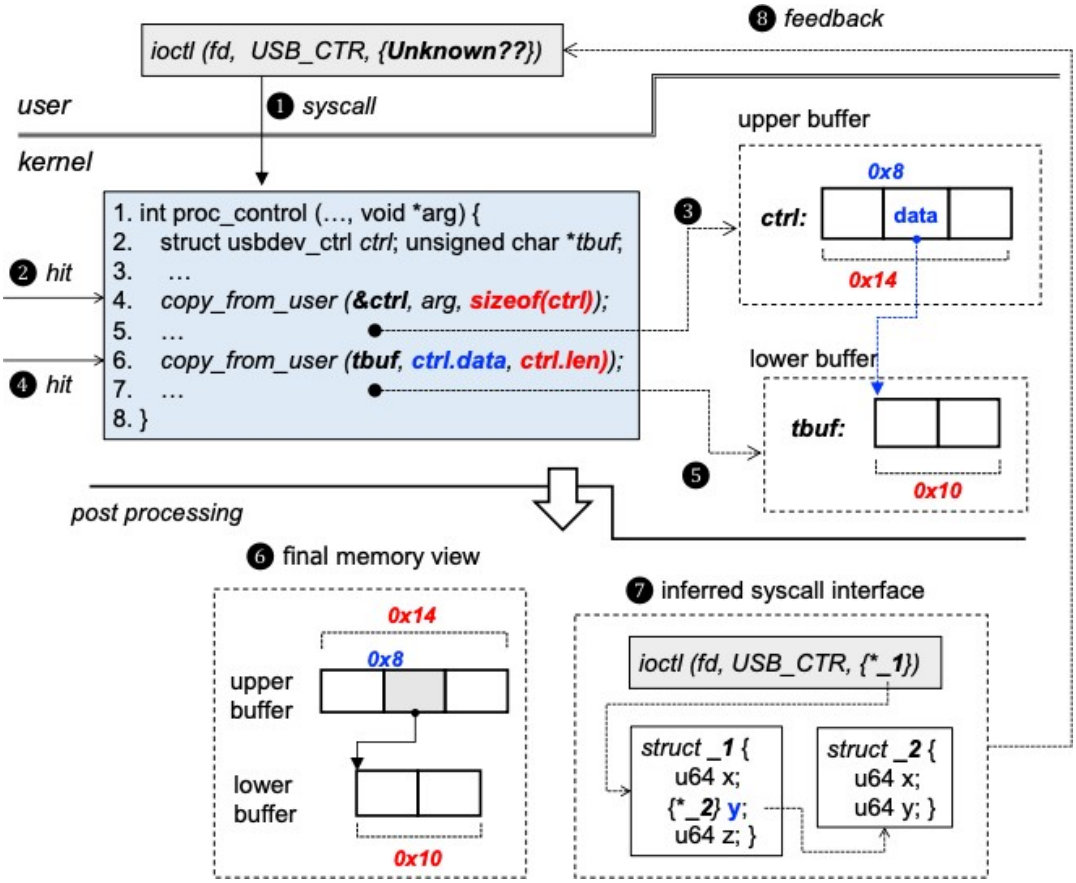


Figure 46: "Workflow of nested syscall argument retrieval [100, p. 9]"

Algorithm for Figure 46:

1. Monitor execution at the data transfer routine
2. Copy the ctrl data structure into symbolic memory, keeping buffer and size constraints
3. Copy the buffer containing the data, keeping symbolic relationship between ctrl and tbuff
4. Construct a final memory view describing the structure
5. Define new argument rules for the invoked system call

Use Cases and Limitations

HFL is designed to fuzz the Linux kernel. It leverages Linux specific patterns for transitioning system call arguments from user to kernel space. Its analyses are limited to the x86-64 architecture (at the time of writing) because of its dependencies on Syzkaller and S2E. No public implementation currently exists, and the available information is limited to performance metrics.

Performance Metrics: HFL outperformed Syzkaller and other kernel fuzzing tools - "the coverage improvement of HFL over Moonshine and Syzkaller is 15% and 26% respectively, [and four times improvement over] kAFL, TriforceAFL and S2E [100, p. 12]". It found "24 vulnerabilities previously unknown [of which] 17 were confirmed [100, p. 12]", the majority being

"integer overflow and memory access violation, followed by uninitialized variable access [100,p. 12]." HFL found the same bugs as Syzkaller in 15 hours, compared to the latter's 50 hours, and exceed many other tools' coverages.

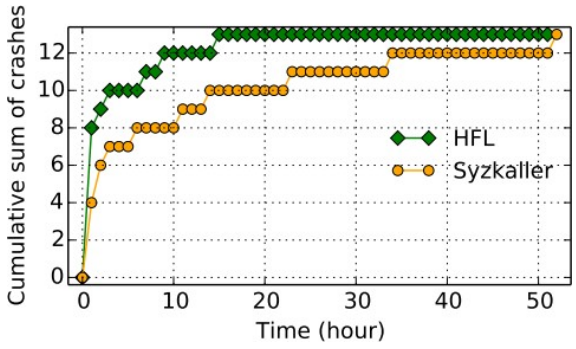


Figure 47: "Comparison of bug-finding time for 13 known crashes [100, p. 10] "

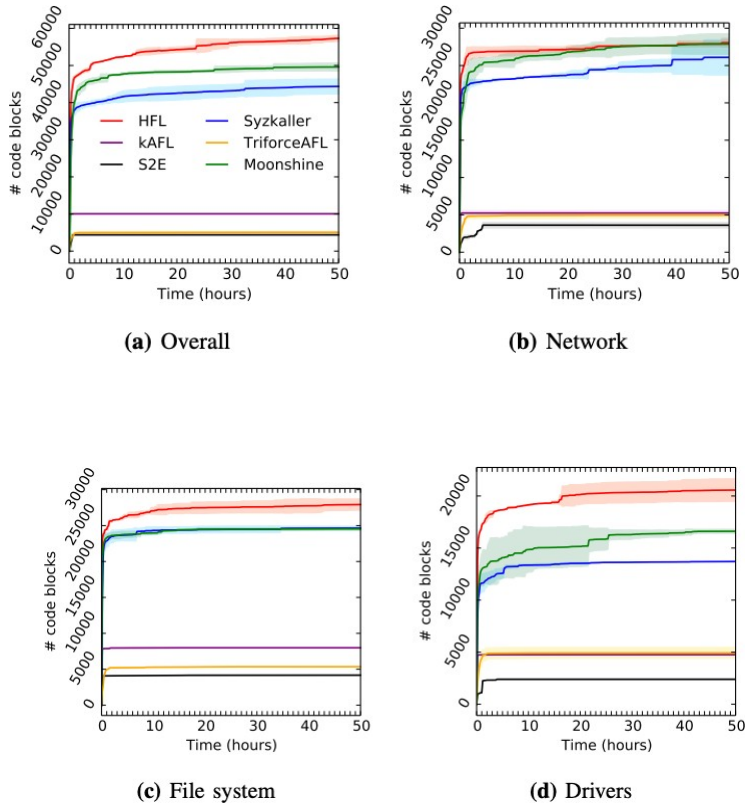


Figure 48: "Coverage results during a [50-hour] run [100, p. 12] "

The overall coverage efficiency compared to a variety of other tools is below, "the line indicates averages while the shaded area represents 95% confidence interval over three runs [of 50 hours each] [100, p. 12]."

The results demonstrate that hybrid fuzzing is most useful, followed by system call inference and argument retrieval, then resolution of indirect system calls to direct. However, the authors "emphasize that all of HFL's features, rather than exclusive application of each feature, are essential for maximizing coverage [100, p. 13]"

5.4.2 Hypercube

Reference Link	**https://www.ndss-symposium.org/wp-content/uploads/2020/02/23096.pdf**
Target Type	Binary (Hypervisor)
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	Tool not yet released; Paper released Jan 2020
License Type	Not yet available, license that will be used not stated
Maintenance	Tool not yet released

Overview

HYPER-CUBE is a hypervisor fuzzer implemented as a custom OS that boots inside one of several hypervisors/virtual machine monitors and attempts to crash it. HYPER-CUBE was presented in the 2020 NDSS paper *HYPER-CUBE: High-Dimensional Hypervisor Fuzzing* [101]. HYPER- CUBE is not guided by past execution traces or instrumentation. Instead it takes a fast and "dumb" brute-force approach. Despite the lack of coverage-guidance, HYPER-CUBE states it has found the same bugs as other, more complicated approaches, in much less time. As of two months after the conference, the source code has not yet been released despite it being linked to in the paper and the authors criticizing its nearest competitor VDF [107] for not releasing their tool. [101]

Design and Implementation

The goal of HYPER-CUBE is to expand on "two-dimensional fuzzing [108]" where two different interfaces were fuzzed in sequence. The authors describe their *high-dimensional* approach as fuzzing an "an arbitrary number of interfaces in any order [101]". In HYPER-CUBE the interfaces are primarily memory areas accessed by emulated devices to perform privileged actions that may not be properly handled by the hypervisor, such as Memory Mapped IO (MMIO). Figure 49 shows the architecture of HYPER-CUBE.

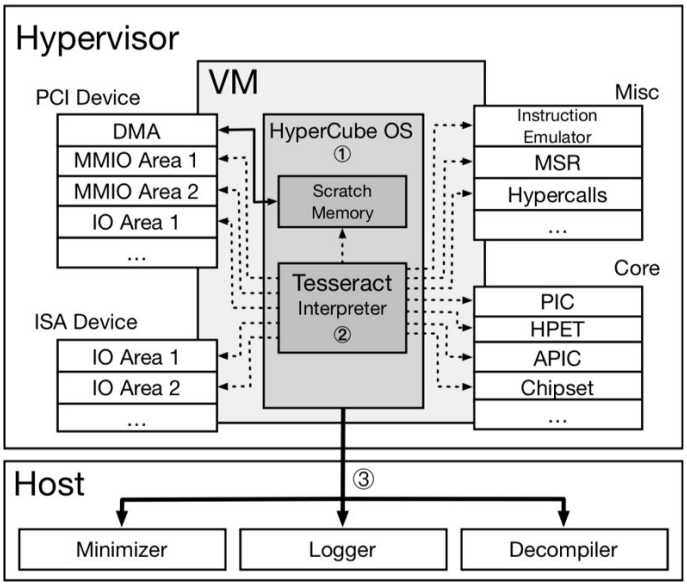


Figure 49: HYPER-CUBE Architecture from paper

HYPER-CUBE is composed of three parts:

1) A small OS that boots in 10% quicker than the Linux kernel. HYPER-CUBE OS enumerates the hardware interfaces and helps avoid costly reboots due to crashes.

2) "TESSERACT" which is a bytecode interpreter that executes instructions manufactured from a PRNG.

3) Various supporting scaffolding programs to manage tasks such as monitoring the VM's behavior and feeding the TESSERACT interpreter byte-strings.

Use Cases and Limitations

HYPER-CUBE can find bugs in hypervisors at a high-rate due to the speed at which it brute forces memory regions and instructions. The authors found 54 bugs, yielding 43 CVEs. One example cited was a bug in BSD's *bhyve* hypervisor when executing a `rep movs` instruction targeting the Advanced Programmable Interrupt Controller (APIC) MMIO region. The memory accesses were incorrectly emulated, leading to a crash of the hypervisor.

HYPER-CUBE's authors contrast it favorably with VDF [107] due to HYPER-CUBE's far higher instruction throughput. HYPER-CUBE purports to make up for intelligent path selection with raw scale. The authors state that they found almost all the same crashes as VDF in less time. However, it is not clear from the paper how HYPER-CUBE manages to avoid the complexity of device initialization that VDF took pains to handle through extensively tracing concrete boots of the kernel. HYPER-CUBE could be missing crashes that require a more complicated initialized device state.

HYPER-CUBE appears straightforward and flexible enough to handle many hypervisors, but it has two primary platform limitations. First, HYPER-CUBE cannot fuzz Hyper-V because it did not implement a 64-bit UEFI bootloader. Second, HYPER-CUBE does not support para- virtualized devices and so does not support Xen or VMware ESXi.

5.4.3 Syzkaller

Reference Link	**https://github.com/google/syzkaller**
Target Type	Binary (Kernel)
Host Operating System	Linux
Target Operating System	Akaros, Android, Darwin/XNU, FreeBSD, Fuchsia, NetBSD, OpenBSD, Windows, gVisor
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64), ARM (32, 64)
Initial Release	2015
License Type	Open-Source
Maintenance	Maintained by Google

Overview

syzkaller is an open-source multi-platform kernel fuzzer that uses a combination of generation- based fuzzing and mutation to discover a wide range of kernel bugs. As an unsupervised fuzzer, syzkaller leverages code coverage to guide and maximize code paths for achieving a significant amount of bug discoveries. To date, thousands of bugs have been found using syzkaller, spanning 7 different operating systems and bug classes, to include memory corruption, General Protection Fault (GPF), deadlocks, and kernel "warnings" that sometimes result in memory leaks [109]. Written in Golang, syzkaller has been actively maintained and publicly available for many years. Designed as a "structure-aware" fuzzer, kernel system calls are targeted by leveraging source code to extract system call descriptions and then micro programs are generated for execution within a closely monitored QEMU virtual machine instance [110].

Design and Implementation

Like other fuzzers, syzkaller can parallelize and run several jobs at once using multiple VMs. It achieves this through a configurable component call syz-manager (shown in Figure 45). The syz- manager acts as a dispatcher and orchestrator for each VM. It will reboot QEMU images as necessary, start processes on the VM, and monitor for new code coverage while maintaining a catalog of crashes and storing the corpus of micro programs viewable from the web GUI's main page (Figure 52).

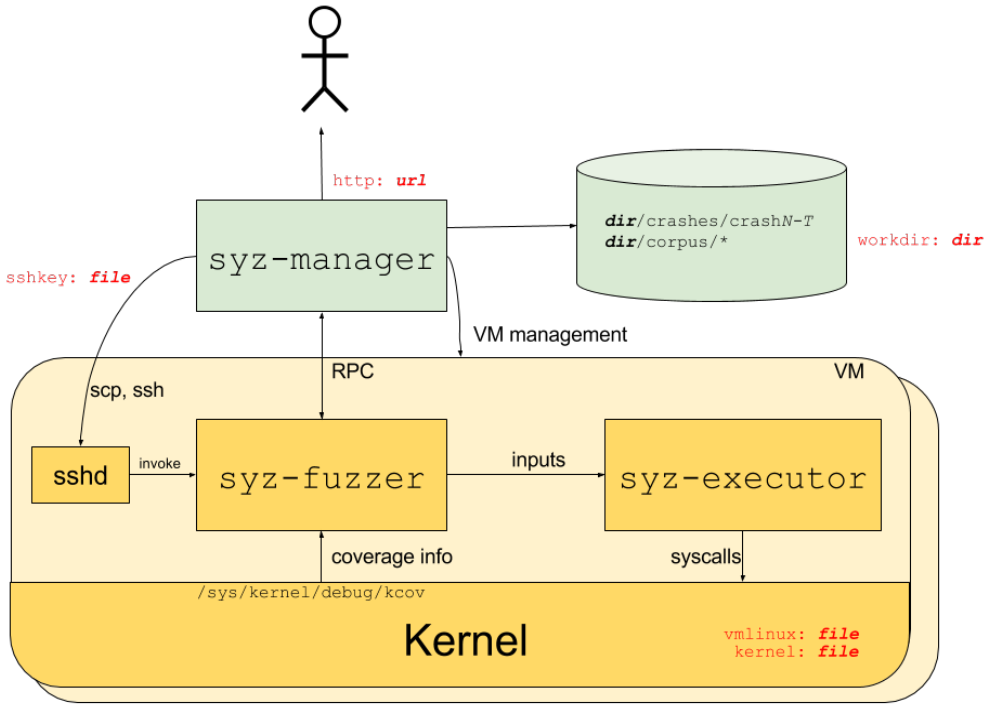


Figure 50: syzkaller System Architecture

The syz-fuzzer component is responsible for fuzzing system calls, specifically the generation and mutation of micro programs consisting of "syscall descriptions." Syscall descriptions are created from the kernel's source code using regular expressions and are made into programs by stringing a random set of system calls in a sequence while also verifying all call dependencies are satisfied. Figure 51 shows an example of a generated program with arguments mutated.

```
pipe2(&(0x7f0000000000)={0xffffffffffffffff, <r0=>0xffffffffffffffff}, 0x0)
open_by_handle_at(r0, &(0x7f0000000100)={0x8}, 0x0)
syz_genetlink_get_family_id$nl80211(&(0x7f0000000200)='nl80211\x00')
accept4$inet(0xffffffffffffffff, 0x0, 0x0, 0x0)
setxattr$security_ima(&(0x7f0000000400)='./file0\x00', 0x0, 0x0, 0x0, 0x0)
syz_genetlink_get_family_id$ipvs(0x0)
syz_genetlink_get_family_id$smc(&(0x7f0000000640)='SMC_PNETID\x00')
creat(&(0x7f0000000800)='./file0\x00', 0x59)
```

Figure 51 Example syscall description program created by syzkaller

Once a program is generated, syz-fuzzer will begin to manipulate various parts of the program using 5 different types of functions.

- **squashAny:** Preforms AFL mutations by picking a random complex pointer and squashes the arguments into an ANY, randomizing blobs if they exist.
- **splice:** Joins another program from the corpus of programs at a random system call index.
- **insertCall:** Inserts an additional system call at a random point, however weighted for inserting calls to the end of the program.
- **mutateArg:** Mutates the argument(s) of a random system call.
- **removeCall:** Removes a randomly selected system call.

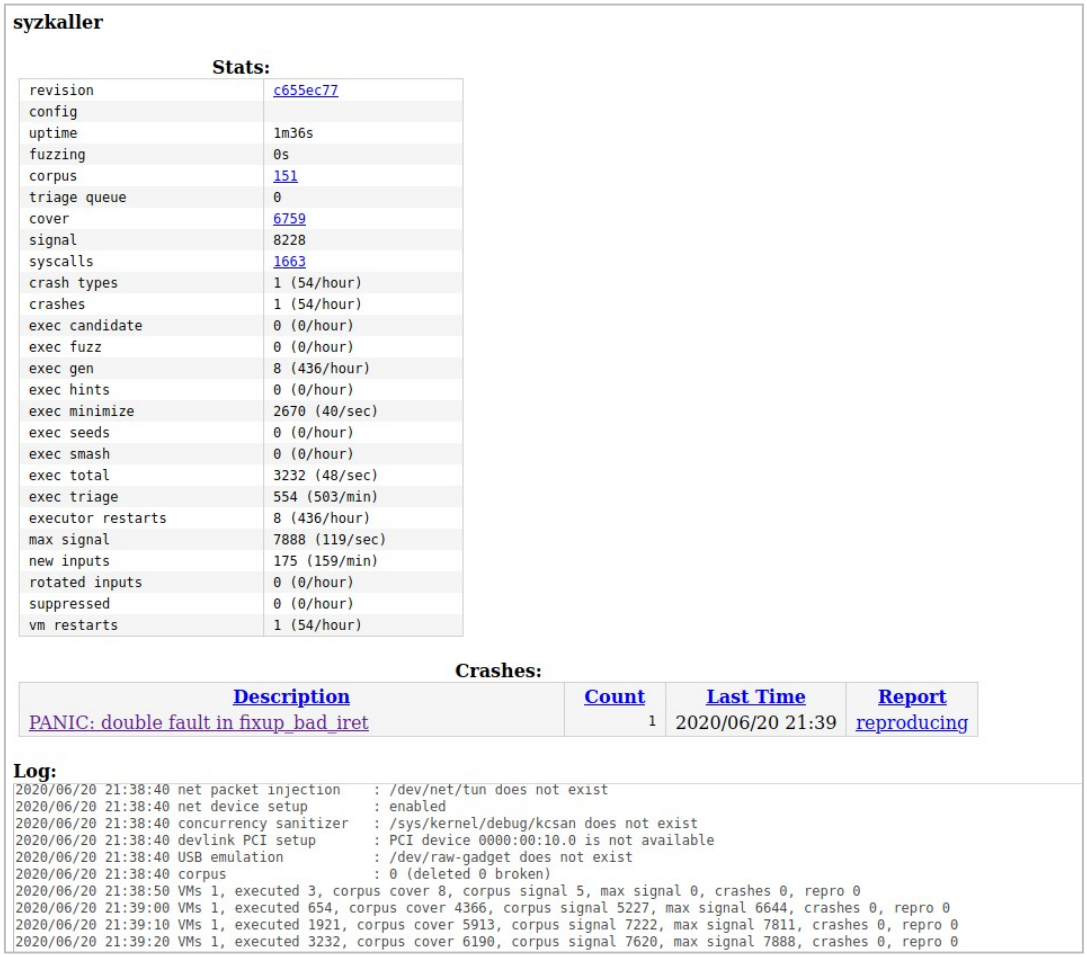


Figure 52 Syzkaller web GUI

To increase code coverage reliably, syzkaller uses KCOV to track runtime executions per thread [112]. When a program is run, the coverage is recorded for that particular program and then executed again to identify possible "code flakes" – parts of code that show up in one KCOV coverage map but fail to be reported within the same program in a sequential run. This is an important feature since the kernel is a complex ecosystem of multiple processes, background threads, and scheduling variabilities that can result in indeterministic execution, thwarting accurate code coverage. Syzkaller's additional executions remove false code coverage flakes and only add verified coverages to the program corpus.

Use Cases and Limitations

Originally designed for targeting Linux kernels, syzkaller has grown to be a highly extensible fuzzing platform supporting numerous operating systems through additional syscall descriptions. In a recent security conference, Windows syscall descriptions were successfully added to a modified version of syzkaller using leaked source code from past Windows operating systems and verified using IDA and WinDbg. Although several system call description extractors are packaged with syzkaller, they are not all encompassing. Missing syscall descriptions attribute to the main limitation of syzkaller, as syscall descriptions are the foundational layer used for supporting program generation through the system call grammar. Without a complete system call description, syzkaller is unable to generate syscall descriptions for every part of the kernel, a problem magnified for closed sourced operating systems or undocumented APIs.

5.5 Trends

The primary trends with fuzzers as of late have been maturation and improvements upon existing ideas, combined with trying to fuzz targets that are traditionally hard to fuzz. TortoiseFuzz, IJON, and Nautilus build upon AFL, trying to create strategies that improve fuzzing performance in a quantitative and measurable manner. Additionally, tools are being created to specifically focus on kernel and hypervisor fuzzing. In general, the field of fuzzing is becoming more mature, and research is focusing fuzzer weaknesses.

6 Exploitation

6.1 Technical Overview

Program bugs come in many forms, but ones which could potentially be exploited to generate unintended effects are considered vulnerabilities. NIST SP 800-30 defines a vulnerability as "a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source [113]." In vulnerability research, once a vulnerability has been identified, the next step is to effectively leverage that vulnerability with an exploit (i.e., to cause unintended behavior in the target). This is often a more challenging task than finding the original vulnerability. Almost all the tools discussed in the first two sections of this report are useful for both vulnerability discovery and exploitation. For example, a researcher might use a debugger to determine exactly how many 'A' characters to pad their payload with so that their exploit is correctly situated. By contrast, this section focuses on tools and techniques explicitly intended for exploitation. For example, shadow is a debugger designed specifically for exploiting heap vulnerabilities in programs that use the jemalloc allocator. It is common for exploitation tools or techniques to be designed for a specific kind of target.

Mitigations

Over the years, numerous defensive techniques have been developed to mitigate the danger of vulnerabilities. Techniques are tailored for different classes of vulnerability, and some have proven more successful than others. At the edge of the art, offensive techniques are often developed to circumvent these mitigations. Common exploit mitigations for binary exploitation include:

- **Stack Canaries:** Values placed on the stack to prevent accessing memory past the end of the buffer.
- **DEP/NX:** This marks regions of memory (like the stack) as not executable, to prevent malicious code from being written and then executed.
- **RELRO:** This prevents the Process Linkage Table (PLT) and the Global Offset Table (GOT) from being abused by making either part or all of it Read-Only.
- **Position Independent Executable (PIE):** This constructs the code portion of an executable with relative addressing to make it position independent. This is used in conjunction with ASLR.
- **Address Space Layout Randomization (ASLR):** This randomizes the position of areas within the virtual address space of the program, including the positions of the executable code (if PIE is enabled), the stack, heap, and various libraries.
- **Control Flow Integrity:** This kind of mitigation attempts to ensure that the intended control flow of the program is followed, to prevent attacks like ROP.

Other mitigations and security practices have also proved useful, such as include privilege separation, privilege revocation, and proper sanitization of data input into a program. The following discusses types of exploit tools and techniques that are actively being researched and developed.

Shellcode

Related EotA Tools: SynesthesiaYS

Shellcode is a set of binary instructions that will perform an exploitative task, such as spawning a shell. However, while the name is derived from that example, shellcode can generate any number of effects. Shellcode often must be constrained to a certain input format consistent with the target program, such as using only alphanumeric bytes.

Control Flow Attacks

Related EotA Tools: DOP, BOPC, KEPLER

Code reuse attacks like return-oriented programming (ROP) are often used to circumvent mitigations like non-executable stacks (DEP/NX). ROP is a technique in which existing code is reused for malicious purposes. In its most basic form, small snippets of instructions that end in a branching instruction can be chained together by pushing their addresses onto the stack. With the right chain of instructions, an attacker can gain control by spawning a shell or some other task. These snippets of instructions are known as "gadgets." There has been significant research into static analysis techniques to identify them within program binaries and libraries. ROP can be thought of as a technique to build a weird machine [114] in which valid parts of the program are manipulated to produce unintended behavior.

Mitigations have emerged to stymie the effectiveness of ROP. Address Space Layout Randomization (ASLR) can make it difficult to determine the location of ROP gadgets at runtime. Control Flow Integrity (CFI) was specifically intended to prevent control flow attacks like ROP by ensuring that the control flow of the program is not circumvented. However, techniques and exploits which bypass CFI have emerged.

In addition to basic ROP, there are variations on the technique that have emerged over the years, to include:

- Blind ROP (BROP): BROP is a technique for attacking remote proprietary services for which the binary is not available [115].

- Sigreturn-oriented programming (SROP): SROP manipulates signal handling in Unix systems to construct a ROP-like weird machine [116].

- JIT-ROP: JIT-ROP is a technique to use ROP with a Just-In-Time (JIT) compiler for JavaScript in browsers. JIT-ROP makes the JIT compiler produce gadgets that can then be used for ROP [117].

- Data-oriented programming (DOP): A technique which creates a ROP-like weird machine only with non-control data in memory. This avoids CFI mitigations [118].

- Block-oriented programming (BOP): This technique builds on DOP [119].

- Jump-oriented programming (JOP): Creates a ROP-like weird machine with jump instructions rather than return instructions [120].

- Counterfeit object-oriented programming (COOP): A technique developed to avoid CFI mitigations by chaining together existing C++ virtual functions [121].

Data-Oriented/Data-Only Attacks: A variant of control flow or code reuse attacks, these attempt to circumvent control flow protections like CFI by manipulating a program's memory and register values, rather than its control flow. The end goal is to use instructions already within a valid control flow that perform reads, writes, and other useful operations on corrupted memory to create a 'weird machine.' The tools DOP [122] and BOPC [123], discussed in this section, address these kinds of attacks.

Heap Exploitation

Related EotA Tools: Gollum, SEIVE and SHRIKE, HeapHopper, SLAKE, shadow

Bugs related to dynamic memory make the heap a target for exploitation. There are a variety of techniques and vulnerabilities for heap exploitation. These include:

- Use After Free (UAF): When free is called on a heap object but the remaining pointer to it is still in use.

- Heap Overflow: Like a stack overflow, this is enabled by a vulnerability that allows an attacker to write past the end of the intended buffer on the heap and overwrite other heap buffers.

- Heap Spraying: This is a technique in which a payload is "sprayed" onto the heap to increase the likelihood that it is in the correct place for the exploit to work.

- Double-free: This is a vulnerability in which a heap object is freed twice. An attacker can then possibly control a pointer to that area of memory in a malicious way.

There are many other ways to attack the heap. These kinds of exploitation techniques are often more complicated than stack-based attacks and can require knowledge of the internals of a specific allocator.

The tools HeapHopper, and SHRIKE and SEIVE, which were discussed in the second version of this report, are Automatic Exploit Generation (AEG) tools for heap-based attacks. The tool Gollum (discussed in this section) builds on the research for SHRIKE and SEIVE to construct a full heap- based AEG. The tool SLAKE, also discussed in this section, targets the Linux kernel space dynamic memory allocator, the SLAB/SLUB.

Automatic Exploit Generation (AEG)

Related EotA Tools: BOPC, KEPLER, SLAKE, Gollum, SEIVE and SHRIKE, Revery, angr

AEG techniques are an area of research which has existed for a decade but gained notoriety in 2016 during the DARPA Cyber Grand Challenge (CGC) [98]. As the name suggest, AEG involves automatically writing an exploit for a vulnerability. Sometimes the AEG tool is given a known vulnerability, but an end-to-end AEG machine is designed to automatically discover the vulnerability and then exploit it. Often, this can be conceived as a two-step process. First, discover an input that triggers the vulnerability, and then generate a 'payload' to exploit that vulnerability [124, p. 11]. Many of the tools and techniques discussed elsewhere in this report are applicable to AEG, especially symbolic execution and fuzzing.

This section focuses on the second step, automatic exploitation after vulnerability discovery. AEG techniques have been applied to the stack, heap, kernel, etc. The tools HeapHopper, SHRIKE, and SEIVE are AEG tools for heap-based attacks. Gollum builds on the research for SHRIKE and SEIVE. The first version of this report discussed FUZE [125], which is an AEG tool designed to exploit the Linux kernel. Two additional AEG tools for the Linux kernel are discussed in this section, KEPLER and SLAKE [126]. BOPC [123] automates Data-Oriented Programming (DOP) attacks.

6.2 Data-Oriented Attacks

6.2.1 DOP

Reference Link	**https://huhong-nus.github.io/advanced-DOP/**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2016
License Type	Open-Source
Maintenance	Maintained by

Overview

DOP is a tool for data-oriented programming, an exploitation method which manipulates a program's memory (rather than its control flow) to achieve some level of control of the program, ideally turning it into a Turing machine. Presented in the paper *Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks* [122], DOP is described as a way "to construct expressive non-control data exploits for arbitrary x86 programs." The significance of these methods is that they bypass Control Flow Integrity protections because they manipulate the data flow instead of the control flow. Although less powerful, and more difficult to construct than a control flow attack, the authors show that programs often have the building blocks for these attacks, and given a well-placed vulnerability, these attacks can be effective. The authors demonstrate this on two real world programs, constructing a Turing machine in both examples. [122] [127] [128]

Design and Implementation

DOP formalizes a language, called MINDOP, of data-oriented "gadgets" which can be chained together using a data-oriented dispatcher. A gadget takes the form of an instruction or set of instructions that performs some operation (i.e., arithmetic, assignment, load, store, jump or conditional jump) on an attacker-controlled value. A dispatcher is typically some form of loop which allows a gadget to be called arbitrarily by the attacker. Figure 53 shows the semantics of MINDOP, which the authors show to be Turing complete.

MINDOP language. To simulate (conditional) jump, data-oriented gadget changes the virtual input pointer (vpc) accordingly.

Semantics	Instructions in C	Data-Oriented Gadgets in DOP
arithmetic / logical	`a op b`	`*p op *q`
assignment	`a = b`	`*p = *q`
load	`a = *b`	`*p = **q`
store	`*a = b`	`**p = *q`
jump	`goto L`	`vpc = &input`
conditional jump	`if a goto L`	`vpc = &input if *p`

p – &a; q – &b; op – arithmetic / logical operation

Figure 53- MINDOP Language [122, p. 4]

DOP includes an LLVM-based tool which can identify gadgets in a program. Once gadgets have been discovered, they are semi-manually chained together. It is in this way that this tool is not a fully automatic generator, but a method to automate parts of the process. One might consider it analogous to tools such as ROPGadget which automatically find gadgets for ROP and other control flow-oriented attacks. Figure 54 shows the interaction between gadgets and dispatchers.

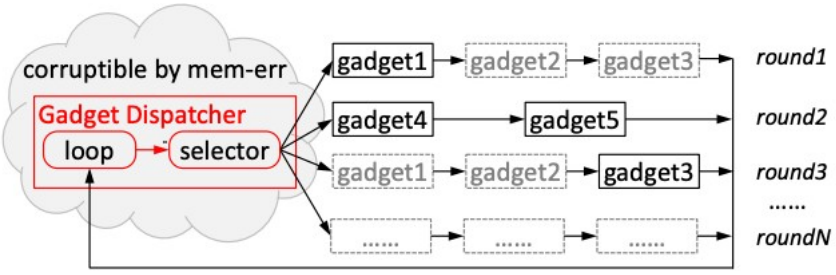


Fig. 2. The design model of DOP in MINDOP. The gadget dispatcher includes a loop and a selector. The loop keeps the program passing by the selector and various data-oriented gadgets. For each round, the selector is controlled by the memory error to activate particular data-oriented gadgets.

Figure 54 - Diagram of gadget and dispatcher interaction [122, p. 5]

Example: Consider the "micro-operations" described in Figure 55.

```
1  struct server{ int *cur_max, total, typ;} *srv;
2  int connect_limit = MAXCONN; int *size, *type;
3  char buf[MAXLEN];
4  size = &buf[8]; type = &buf[12];
5  ...
6  while(connect_limit--) {
7    readData(sockfd, buf);       // stack bof
8    if(*type == NONE  )  break;
9    if(*type == STREAM)          // condition
10      *size = *(srv->cur_max); // dereference
11   else {
12      srv->typ = *type;         // assignment
13      srv->total += *size;      // addition
14   } ... (following code skipped) ...
15 }
```

Code 2. Vulnerable FTP server with data-oriented gadgets.

Figure 55 - Example of gadgets in code [122, p. 3]

"The line 12 is an assignment operation on memory locations pointed by two local variables (srv and type), which are under the influence of the memory error. Line 10 has a dereference operation, the source pointer (srv) for which is corruptible. Similarly, Line 13 has a controllable addition operation. We can think of each of these micro-operations in the program as data-oriented gadgets. If we can execute these gadgets on attacker-controlled inputs, and chain their execution in a sequence, then an expressive computation can be executed."

Using the *while* loop, the behavior described in the code in Figure 56 can be simulated by manipulating the inputs into the code in Figure 55.

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  void updateList(struct Obj *list, int addend) {
3     for(; list != NULL; list = list->next)
4        list->prop += addend;
5  }
```

Code 3. A function increments the integer field of a linked list by a given value. It can be simulated by chaining data-oriented gadgets in Code 2.

Figure 56 - Behavior simulated by code in Figure 55 [122, p. 3]

"Notice that the loop in line 6 to 15 allows chaining and dispatching gadgets in an infinite sequence, since the loop condition is a variable (i.e., connect_limit) that is under the memory error's influence. We call such loops gadget dispatchers. A sequence of data-oriented gadgets in [Figure 41] would allow the remote adversary to simulate the function shown in [Figure 56], which maintains a linked list of integers in memory and increments each integer by a desired value."

Use Cases and Limitations

The authors describe DOP's usability as follows,

"We identified 7518 data-oriented gadgets from 9 programs. 8 programs provide x86 data-oriented gadgets to simulate all MINDOP operations. In fact, there are multiple gadgets for each operation. These gadgets provides [sic] the possibility for attackers to enable arbitrary calculations in program memory… This result implies that real-world applications do embody MINDOP operations and are fairly rich in DOP expressiveness…Our programs contain 5052 number of gadget dispatchers in total, such that each program has more that one dispatcher … 1443 of these dispatchers contain x86 gadgets of our interest …This means that the dispatchers are abundant in real-world programs to simulate MINDOP operations."

However, their definitions of gadgets and dispatchers is such that these numbers account for many basic operations or loops in a program, things which one would expect to be ubiquitous. What matters is how many gadgets they were able to identify that are within the scope of avulnerability and act on the memory controllable by the user due to the vulnerability. That number is far lower. However, the authors were able to use their technique to create Turing machines for two of the vulnerabilities they looked at, in one case leaking the private key of a server. This is the kind of attack for which DOP is most useful.

6.2.2 BOPC

Reference Link	https://github.com/HexHive/BOPC
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	November 2019
License Type	Open-Source
Maintenance	Maintained by HexHive

Overview

The Block Oriented Programing Compiler (BOPC) is a tool designed to automate Block-Oriented Programming (BOP) attacks on vulnerable binaries. Described in the paper *Block Oriented Programming: Automating Data-Only Attacks* [129] by Ispoglou et al., BOPC, and their technique, BOP, are closely associated with the related work on Data-Oriented Programming (DOP) [122] and the tool DOP (also discussed in this section). BOPC builds on the idea of DOP and data-only attacks to avoid control-flow mitigations like CFI and shadow stacks. Rather than leveraging any sequence of instructions, as DOP does, BOPC leverages basic blocks, hence the name. Assuming an attacker already has the ability to write anywhere in memory, BOPC takes as input a sequence of operations (i.e. payload) that the attacker wishes to execute but cannot due to control flow protections, written in their SPloit Language (SPL). BOPC then "finds execution traces in the binary that execute the desired payload while adhering to the binary's Control Flow Graph (CFG)". The authors call their tool "code reuse under CFI." The tool is written in Python and is accompanied by a well-documented README. It is, however, a paper artifact so its usability has not thoroughly tested or accounted for. [129] [123]

Design and Implementation

The core assumption of BOPC is that the attacker already has a write-anywhere primitive. In the case of a control flow attack, the attacker might use this to overwrite a function pointer. In the presence of CFI, an attack such as this might become more difficult. A data-only attack, however, removes the need to work around CFI, since it bypasses it entirely.

BOPC takes in three inputs:

- A target binary with this write-anywhere primitive.
- The desired exploit expressed as a sequence of operations to be carried out via a data- only attack. This 'payload' is written in the *SPloit Language (SPL),* a high-level, Turing complete language the authors developed to be like C. These could be anything from initializing registers with arbitrary values to spawning a shell. An example of an SPL payload can be found in Code 4.
- An 'entry point,' which is the first instruction in the binary at which the payload can be executed. This allows the attacker to first complete the part of the exploit that allows

them the write-anywhere primitive, as well as any other operation that needs to be done before the payload is executed. Multiple entry points can be given.

```
1  //
2  // BOPC Evaluation
3  //
4  // execve('/bin/sh') payload
5  //
6  void payload()
7  {
8      string prog = "/bin/sh\0";
9      int argv    = {&prog, 0x0};
10
11     __r0 = &prog;
12     __r1 = &argv;
13     __r2 = 0;
14
15     execve(__r0, __r1, __r2);
16
17     // return ?
18 }
```

Code 4: Example of an SPL payload for execve('/bin/sh') [123, p. BOPC/payloads/execve.spl]

Once an SPL payload has been constructed and input along with entry points and the target binary, BOPC can begin to automatically find basic blocks in the target binary which implement individual SPL statements within the payload. These are called *functional blocks*. To do this, BOPC generates a *Block Constraint Summary* for each basic block, which records the changes in registers and memory which occur when this block is executed, as well as any "any potential system, library call, or conditional jump at the end of the block [129, p. 6]." These summaries are generated using symbolic execution and for this the authors use the symbolic execution tool angr [130].

Functional blocks are then identified by iteratively identifying and then reducing a set of candidate basic blocks using an algorithm which creates a graph that maps the 'virtual' registers and variables in the payload to hardware registers and memory addresses, respectively, and then finding a maximum bipartite match in these graphs, such that the necessary memory addresses and registers can be properly reserved for the SPL execution. If this mapping is found, a set of functional blocks can be selected.

Once functional blocks have been determined, they are chained together using *dispatcher blocks* which are basic blocks that can be used to chain together these functional blocks in order to execute the payload. These dispatcher blocks must adhere to CFI and other control flow protections, as well as not clobber the register and memory state needed to execute the payload (this is called the *SPL State).* This is done using a method like the *k-shortest path* algorithm [129, p. 8].

A functional block, chained with a dispatcher block that allows it to reach the next functional block in the SPL execution, is called a *BOP Gadget.* A diagram of this is shown in Figure 57.

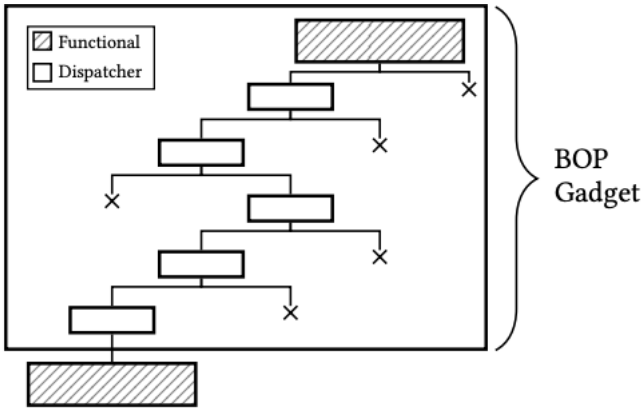


Figure 2: BOP gadget structure. The *functional* part consists of a single basic block that executes an SPL statement. Two functional blocks are chained together through a series of *dispatcher* blocks, without clobbering the execution of the previous functional blocks.

Figure 57: Diagram of a BOP Gadget [129, p. 4]

Once BOP gadgets have been constructed, they can be stitched together and used to execute the payload. A high-level overview of the BOPC process can be seen in Figure 58.

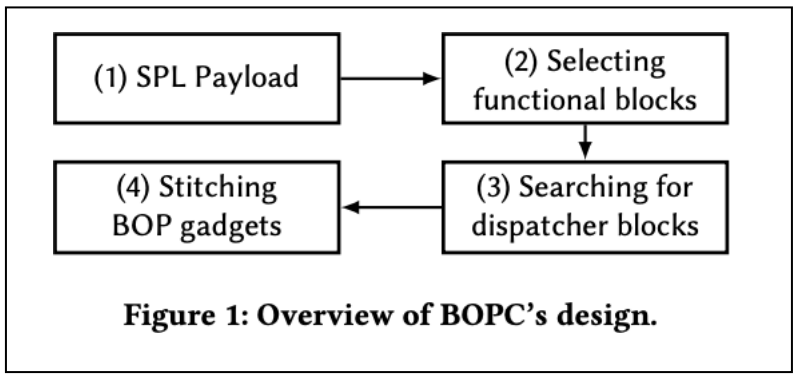


Figure 1: Overview of BOPC's design.

Figure 58: High-level overview of BOPC [129, p. 3]

The BOPC implementation provides the user with a number of useful features, like the optimizer (-O option) which can make it easier to identify gadgets for SPL payloads by rearranging the order execution of the payload with the *out of order execution* option (-O ooo), or by rewriting the payload with the *statement rewriting* option (-O rewrite). In the rewrite case, an execve() in a payload might be replaced by an execv() if the former was not called by the target binary but the latter was.

Use Cases and Limitations

This tool has several requirements which do constrain its usage. It requires that the attacker already have an arbitrary write primitive, which in turn requires a vulnerability for which a partial exploit, which can write anywhere in memory, has been developed. However, this requirement makes this tool perhaps more useful than DOP, which focuses on memory immediately controllable by a vulnerability. While it is true that the generalizations provided by DOP can be abstracted for BOPC's attack model, which is exactly what makes BOPC useful. It has built on the work of DOP in an effective way by applying that research to an arbitrary-write primitive and automated it. BOPC is also more robust and better documented and maintained than DOP.

Like MinDOP, BOPC is designed for avoiding control flow mitigations like Control Flow Integrity (CFI) and shadow stacks. Its authors refer to its capability as "code reuse under CFI." BOPC's SPL language is Turing complete, and like MinDOP, the authors here have shown that many programs have gadgets that can be coupled with a vulnerability to simulate a Turing machine. However, this is less powerful than remote code execution as the Turing machine's 'tape' is limited to this program's virtual memory space. Also, although this tool can be used "automatically," in practice it would at best be a helpful tool during manual data-only exploitation.

This paper's implementation is higher quality and better documented than most paper artifacts, however the authors urge caution: "It's not a product, so do not expect it to work perfectly under all scenarios. It works nicely for the provided test cases, but beyond that we cannot guarantee that will work as expected [123, p. README.md]."

6.3 Heap

6.3.1 Gollum

Reference Link	**http://www.cs.ox.ac.uk/tom.melham/pub/Heelan-2019-GMG.pdf**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	October 2019
License Type	Open-Source
Maintenance	Paper artifact with little to no maintenance

Overview

Gollum is an automatic heap exploitation tool based on the paper *Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters* by Heelan et al. Building on the paper *Automatic Heap Layout Manipulation* by Heelan et al., Gollum incorporates the tools SEIVE and SHRIKE into an end-to-end automatic exploit generation tool for heap overflows in interpreters. Gollum is directed at the PHP and Python interpreter and works by breaking heap generation into several distinct components and then chaining together modular generation tools for each (with layout manipulation being one of these components). [131]

Design and Implementation

Gollum is implemented in six stages:

1. Importing the Vulnerability Trigger
2. Injecting the Vulnerability Trigger into Tests
3. Exploring Heap Layouts
4. Determining Input-Output Relationships
5. Generating an Exploit Modulo a Heap Layout
6. Solving the Heap Layout Problem

Figure 60 is a full workflow diagram.

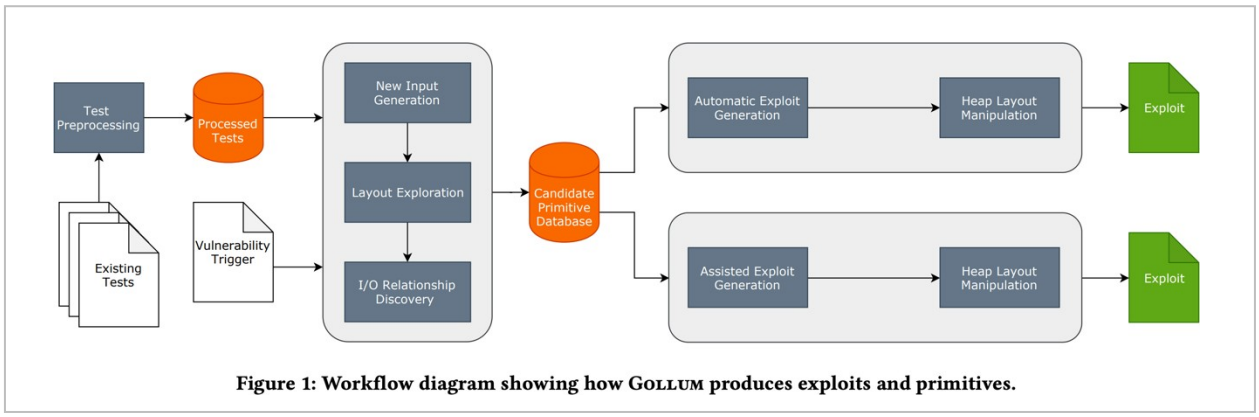


Figure 1: Workflow diagram showing how GOLLUM produces exploits and primitives.

Figure 59 - Workflow diagram for Gollum [131, p. 3]

Gollum takes as input a vulnerability trigger for a heap overflow and a set of test cases (e.g., regression test suite for the interpreter). As output, it returns full exploits, and primitives that can be used in manual exploits. Rather than one single tool, Gollum is made up of several modular tool components which are chained together to create an exploitation tool. Shapeshifter is a custom memory allocator which allows the user to specify a heap layout to determine if that layout would facilitate exploitation. SEIVE is augmented by a genetic algorithm. Then there is the component which searches for exploit primitive, and the component which constructs the exploit.

In steps 1 and 2, the vulnerability trigger is injected into the testcases. Step 3 executes those test cases with Shapshifter to explore what heap objects are allocated at that point. Step 4uses fuzzing to determine what level of control the user has over the heap at the point of vulnerability. Step 5 uses Shapeshifter to determine if an exploit is possible for some layout of the heap. Steps 3-5 result in primitive discovery, which works as follows.

1. Discover a data structure to target and corrupt.
2. Ensure instance of structure adjacent to source of overflow.
3. Ensure post-overflow corrupted data is used by the attacker.

Step 6 uses SEIVE, however this time augmented with a genetic algorithm rather than a random search, to find inputs which would result in this exploitable layout.

Figure 61 shows example code for various points throughout Gollum's process.

```
# --- Original vulnerability trigger --- #
import socket
r, w = socket.socketpair()
w.send(b'X' * 1024)
r.recvfrom_into(bytearray(), 1024)

# --- Test for XML Parsing --- #
import unittest
from xml.parsers import expat
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()

# --- Program combining test and trigger --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"X" * 1024)
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with one-gadget payload --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with Heap Manipulation --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        self.v0 = bytearray('A'*935)
        self.v1 = bytearray('A'*935)
        self.v2 = bytearray('A'*935)
        self.v1 = 0
        ...
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)
```

Listing 1: The various Python programs involved in the exploitation process for the motivating example. The code under each comment would be a separate program but are presented in a single figure to save space. Imports are only shown for the first two code snippets.

Figure 60 - Example code for stages of Gollum [131, p. 4]

Use Cases and Limitations

The most immediate use of this tool is on vulnerabilities for the Python and PHP interpreter, but the most interesting aspect of this tool is its applicability to the problem of automatic exploit generation (AEG) for heap exploits. Whereas there has been significant research into AEG for stack-based vulnerabilities and exploitation methods like ROP, heap exploits present a much more

challenging problem. Gollum, and Heelan et al.'s previous work on SEIVE and SHRIKE are promising forays into this area, however these tools themselves remain significantly constrained in their abilities. One interesting contribution here is the modularity of this tool, which seems to be an effective way to combine AEG components into an end to end exploit generator.

6.4 Kernel Exploitation

Reference Link	**https://github.com/ww9210/kepler-cfhp**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2019
License Type	Open-Source
Maintenance	Not Actively Maintained

Overview

KEPLER is a suite of tools for exploiting the Linux kernel that focuses primarily on gadget identification and gadget chain identification. It is described in the paper *Kepler: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities* [132] which was presented at USENIX 2019. This can essentially be thought of as analogous to a ROP gadget tool for Linux kernels, and fits within both the control flow hijacking attacks and automatic exploit generation in kernels research areas. This tool uses symbolic execution to identify gadgets and evaluate them. It is designed to generate a kernel ROP chain bootstrapped to the exploit primitive that the user input. [132] [133]

Design and Implementation

KEPLER is a ROP chain gadget tool for Linux kernels, which will take an exploit primitive and theoretically return a full exploit using symbolic execution to find ROP gadgets and string them together in a control flow which will allow the attacker to exploit the program using the primitive. Figure 62 shows KEPLER's overall design.

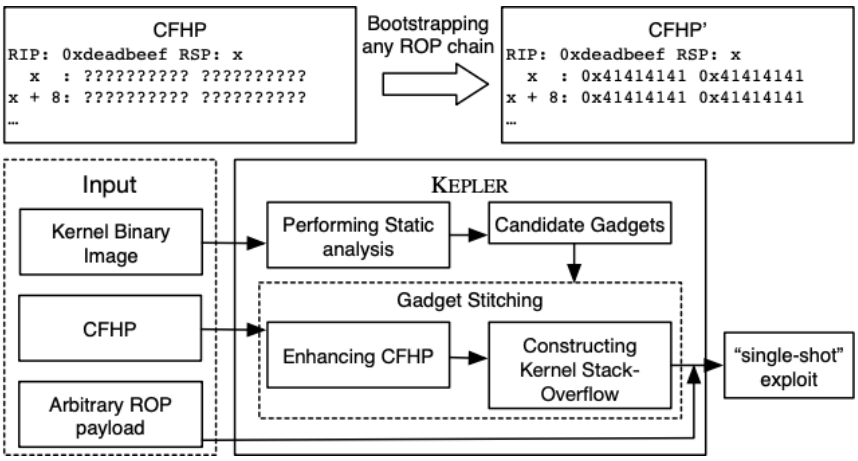


Figure 61: KEPLER's design [132, p. 1193]

KEPLER takes in 3 inputs:

1. A kernel binary image
2. A control flow hijacking primitive (CHFP). This is a "is a machine state that potentially deviates from the legal control-flow graph. In the context of symbolic analysis, a control- flow hijacking primitive is usually identified by applying a heuristic which queries the backend constraint solver to check whether the number of possible control flow jump target is beyond a threshold when the control flow jump target contains symbolic bytes."
3. An arbitrary return-oriented programming (ROP) payload. Code 6 below shows an example of this payload.

```
1  // copy more payloads to the kernel stack
2  // prepare auguments for copy_from_user()
3  p[i++]=POPRAX; // pop rax ; ret
4  p[i++]=POPRSI; // pop rsi ; ret
5  p[i++]=0xffffffff81254a99; // mov rdi, rsp ;
       call rax
6  p[i++]=POPRAX;
7  p[i++]=0x1000;
8  p[i++]=0xffffffff81a04201; // sub rdi, rax ;
       mov rax, rdi ; ret
9  p[i++]=POPRSI;
10 p[i++]=STAGE_TWO_ROP_PAYLOAD;
11 p[i++]=POPRDX; // pop rds ; ret
12 p[i++]=0x1040;
13 p[i++]=COPY_FROM_USER; // copy_from_user()
14
15 // substract rsp to the first gadget
16 p[i++]=POPRAX;
17 p[i++]=0x1040;
18 p[i++]=0xffffffff81a04201; // sub rdi, rax ;
       mov rax, rdi ; ret
19 p[i++]=POPR12; // pop r12 ; ret
20 p[i++]=0xffffffff810001cc; // ret
21 p[i++]=0xffffffff81c01688; // mov rsp, rax ;
       push r12 ; ret
```

Code 5: Example kernel ROP payload [132, p. 1204]

KEPLER uses a technique called ‘single-shot’ exploitation. They describe this as follows:

> “We present a code reuse exploit technique which converts a single ill-suited control-flow hijacking primitive into arbitrary ROP payload execution under various constraints posed by modern Linux kernel mitigations and the primitive itself.…Our approach to calculate exploitation chain is automatable because the gadget stitching problem could be cast as a search problem over a search space of reasonable size. In addition, the "single-shot" nature of this technique makes it suitable for the vulnerabilities prone to unexpected termination because it avoids stressing a control-flow hijacking primitive for multiple times. [132, p. 1188]”

KEPLER uses symbolic execution to identify potential exploit paths and implements this using the symbolic execution tool angr. First KEPLER performs a depth first search of the binary to identify gadgets, and then uses symbolic execution to identify the memory accesses surrounding the gadget and performing constraint solving. Figure 63 displays a kernel exploit primitive which could be fully exploited by KEPLER.

```
1  struct ip_mc_socklist {
2    struct ip_mc_socklist *next_rcu;
3    struct ip_mreqn multi;
4    unsigned int sfmode;
5    struct ip_sf_socklist *sflist;
6    struct rcu_head rcu;
7  };
```

(a) Definition of struct `ip_mc_socklist`. Its first member `next_rcu` is unmanageable because the PoC uses a heap spray technique which does not allow us to control first `QWORD` of struct `ip_mc_socklist`.

```
1  void ip_mc_drop_socket(struct sock *sk){
2    struct inet_sock *inet = inet_sk(sk);
3    struct ip_mc_socklist *iml;
4    // inet->mc_list is a dangling pointer
5    while ((iml = inet->mc_list) != NULL) {
6      // iml is alias of the dangling pointer
7      inet->mc_list = iml->next_rcu;
8      // queuing a rcu_head for execution in
           the future
9      kfree_rcu(iml, rcu);
10   }
11 }
12 void rcu_reclaim(struct rcu_head *head){
13   head->func(head); // control-flow hijack
14 }
15 void rcu_do_batch(...){
16   struct rcu_head *next, *list;
17   while (list) {
18     next = list->next; // next rcu is
           unmanageable
19     rcu_reclaim(list);
20     list = next;
21   }
22 }
```

(b) Tailored source code pertaining to the `CFHP.` function `ip_mc_drop_socket` repeat invoking `kfree_rcu()` which queues a `rcu` task for asynchronous execution until `inet->mc_list` is `NULL`. The site pertaining to the `CFHP` is in function `rcu_reclaim`.

Table 1: A control-flow hijacking primitive in kernel UAF vulnerability CVE-2017-8890.

Figure 62: Example of a kernel exploit that could be exploited by KEPLER [132, p. 1191]

Use Cases and Limitations

KEPLER is useful for Linux kernel hijacking attacks. As the authors stated in their paper,

"Our experiment utilizes KEPLER to explore the aforementioned kernel image with the vulnerabilities inserted. In this process, we exhaustively search gadget chains useful for exploitation and mitigation circumvention… As we can observe, KEPLER could automatically pinpoint tens of thousands of unique kernel gadget chains to perform exploitation without triggering kernel protections. Since we implement KEPLER to perform gadget chain exploration in parallel, we also discover that these gadget chains could typically be identified within 50 hours. These observations together imply that KEPLER could diversify the ways of performing kernel exploitation in an efficient fashion. Given that some commercial security products pinpoint kernel exploitation by using the patterns of exploits, the ability to diversify exploitation has the potential to assist an adversary to bypass the detection of commercial security products. [132, pp. 1199- 1200]"

However, as is the case with many of these tools, the efficacy of it is relegated to a more academic and theoretical context. This requires very specific primitives, as well as the time and desire to work with not well documented paper artifacts, as opposed to a well-documented tool like ROPGadget. However, given that there are few ROP gadget identifiers for the Linux kernel, that may make it worth the required time and effort.

6.4.1 SLAKE

Reference Link	**https://github.com/chenyueqi/SLAKE**
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2019
License Type	Open-Source
Maintenance	Not Actively Maintained

Overview

SLAKE is an Automatic Exploit Generation (AEG) tool presented in the paper *SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel.* A *slab* is a memory structure analogous to the Linux heap used in the Linux kernel. This research builds on userland heap AEG research, such as SEIVE, SHRIKE, and Gollum. They first "use static and dynamic analysis techniques to explore the kernel objects, and the corresponding system calls useful for exploitation." Then they "model commonly-adopted exploitation methods and develop a technical approach to facilitate the slab layout adjustment." The authors assert that their paper not only generated exploits for 27 SLAB vulnerabilities, but also found novel ways to do so.

Design and Implementation

SLAKE is made up of a range of tools, including a fuzzer, tracer and static analyzer.

Their technique uses both static and dynamic analysis. In the former case, they use LLVM to compile the Linux kernel using the tool KINT, which reconstructs the Linux kernel call graph. In the latter case they use the kernel fuzzer Syzkaller integrated with the tool Moonshine to fuzz the kernel. SLAKE first uses static analysis to model the kernel as a call graph and reconstruct the layout of the SLAB and then by fuzzing the SLAB via Syzcaller.

SLAKE relies on several assumptions:

- A kernel vulnerability with a working POC that results in a kernel panic
- Controlling PC will result in exploitation

Figure 64 is an illustration of three of the four kinds of kernel exploitation approaches that SLAKE attempts to achieve.

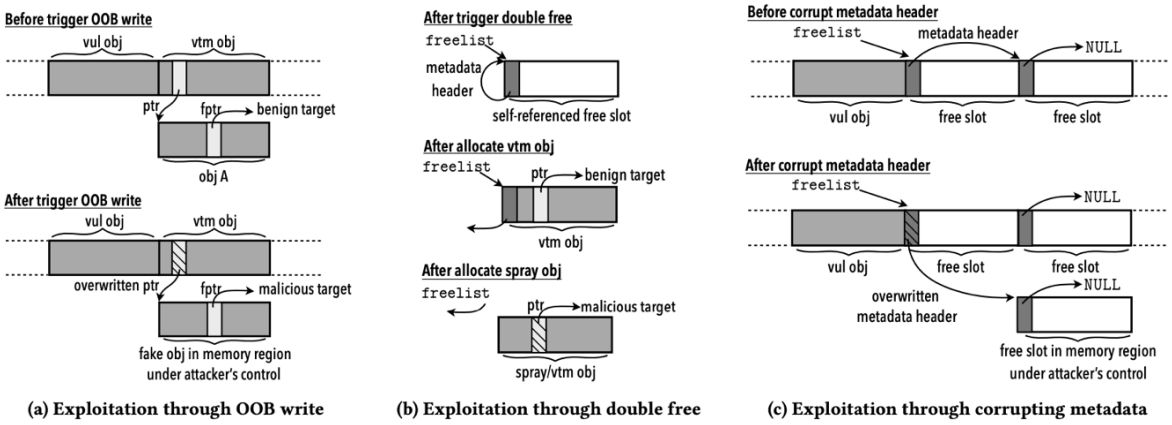


Figure 63: Example kernel exploitation approaches given in [126]

In (a), an out-of-bounds (OOB) write is an exploit in which there is an overflow vulnerability allows the attacker to overwrite some object adjacent to where the overflow occurs. If objects are aligned correctly (in userland heap exploitation this is commonly known as 'heap grooming' or 'heap feng-shui'), the attacker may be able to overwrite a powerful object such as a function pointer or data pointer. Even overwriting heap metadata just by one null-byte can be powerful (called a null-byte overflow).

In (b), an attacker is able to free an object twice, which, if done with the right heap manipulation, will be added to some free list twice, and then when reallocated once, will persist on that freelist meaning that there are multiple avenues for exploitation.

In (c), an attacker is able to do the same thing as in (a) but specific to heap metadata.

Not mentioned in Figure 64 are Use-After-Free (UAF) vulnerabilities, which occur when an object is freed (and thus added to a freelist) and subsequently 'used' by dereferencing a pointer to the heap chunk which is now sitting in a bin. This allows an attacker to allocate another object to that chunk, and then use the first reference which was erroneously not nullified to manipulate the second, valid object's data.

Static Analysis: SLAKE first performs static analysis to identify objects and syscalls, then it triages these for objects and sites of interest. SLAKE's static analysis is implemented on top of LLVM IR using the tool KINT.

There are two types of interesting kernel objects:

- **Victim Objects:** "We deem an object as a victim object if, through its enclosed object pointers, we could perform multi-step dereference and eventually identify a function pointer dereference (e.g. *objÁ→ objB->objC->fptr*) [126]"
- **Spray Objects:** "The usages of a spray object include (1) taking over the slot of a freed object - still referenced by a dangling pointer - as well as (2) overwriting the content of that freed object. As a result, a spray object does not have to contain a function or object pointer but provides an adversary with the ability to copy arbitrary data from userland to kernel slab [126]."
-

There are also two kinds of sites of interest:

- **Allocation Sites: "**We examine the def-use chain of the return value for each allocation function and deem the site of an allocation call as a site of allocation if and only if that return value is cast to the type identified [126]."
- **Deallocation sites:** "For the kernel functions associated with deallocation (e.g., kfree()), they typically take as input the pointer of an object. Similar to the way to pinpoint allocation sites, we can, therefore, track down the deallocation sites tied to victim object by looking at the type of the object pointer passed to the deallocation function [126]."

Dynamic Analysis: After the above have been identified, SLAKE then uses fuzzing to identify systems calls of interest. This is done via the kernel fuzzer syzkaller, in conjunction with the kernel tool Moonshine, QEMU and GDB.

There are three types of system calls of interest:

- **Syscalls for allocation:** "To identify the system calls tied to critical object allocation, we perform fuzz testing against a Linux kernel. When a test case triggers an anchor site tied to object allocation, we profile each of the system calls by recording the kernel objects that every individual system call (de)allocates on the slab [126]."
- **Syscalls for deallocation:** "When a deallocation site is triggered by a test case generated by kernel fuzzer, we check whether the address of that deallocation object matches the address logged previously, and preserve the corresponding system calls only if a match is identified [126]."

- **Syscalls for dereference:** "To identify the system calls that could dereference a function pointer through a target victim object, we first allocate that target victim object through the system calls identified. Under that context, we then perform kernel fuzzing and explore the system calls that can reach to the corresponding dummy dereference site [126]."

After all of these have been identified, SLAKE then performs SLAB manipulation to properly utilize the objects it has selected. For each exploitation technique, a different way of manipulating the SLAB is utilized. An example of this can be seen in Figure 65.

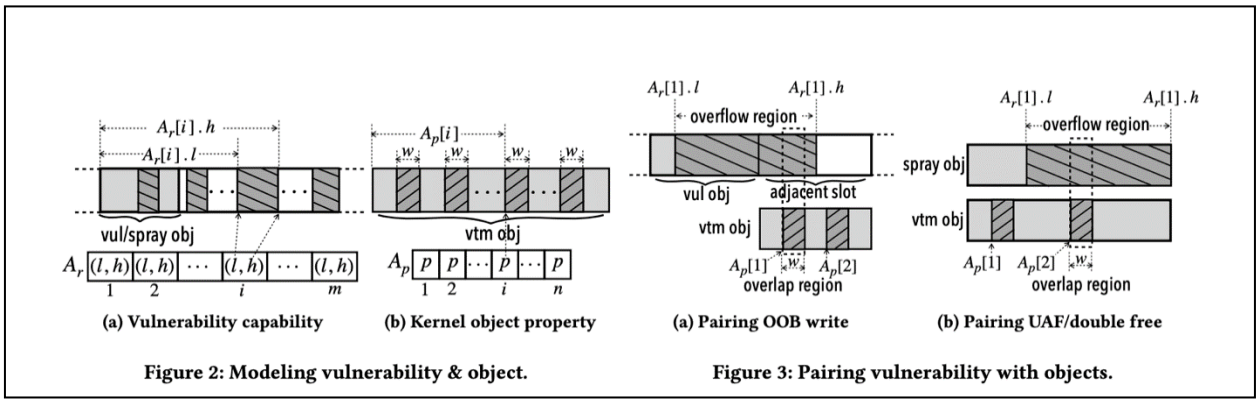


Figure 64: Example of SLAKE manipulating the SLAB [126]

Once exploitable layouts have been identified, SLAKE reorganizes the SLAB's occupied slots, as shown in Figure 66.

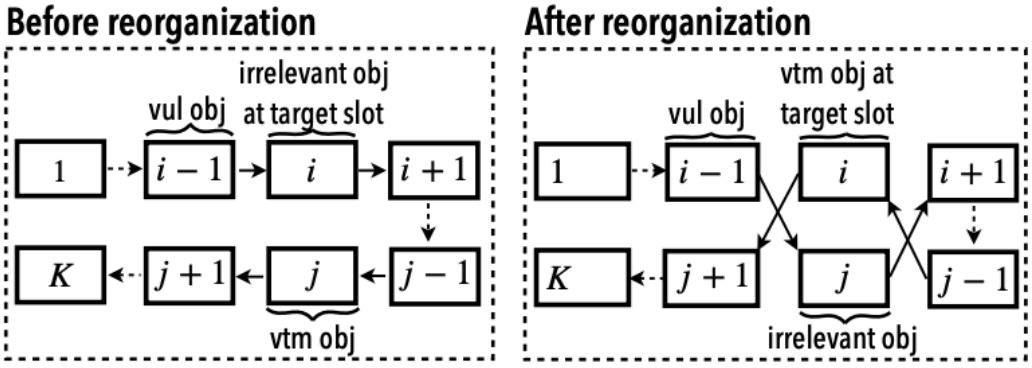


Figure 65: SLAKE reorganization algorithm

After performing reorganization, SLAKE outputs the payload needed to achieve the exploit it has identified. However, this is only the case if SLAKE has been successful.

Use Cases and Limitations

Linux kernel exploitation techniques like this one are not only useful on Linux systems. They can also be applied to other operating systems with similar kernels, such as Android. SLAKE builds on existing work both in the area of heap manipulation and kernel exploitation. Given the novelty of the research, it provides an interesting addition to the area and a good paper for future research to build on.

SLAKE, like many academic papers, proposes interesting and novel ideas, but lacks useable artifacts. The released code is entirely undocumented, making it difficult to use. As a generalized method for conceptualizing heap and SLAB manipulation, and kernel exploitation, it could prove practically useful given difficult kernel SLAB vulnerabilities to exploit.

6.5 Trends

Once again, there is a lot of focus on automated or partially automated exploit generation. This is an area that has seen a lot of research since the DARPA Cyber Grand Challenge (CGC) in 2016. Recently, there has been an increased focus on automated heap exploitation, as well as kernel exploitation, two areas that have traditionally been a bit harder to automatically generate exploits.

There has also been significant focus on various Data Oriented Programming (DOP) exploitation techniques. These come in response to exploit mitigations that have reduced the effectiveness of Return Orient Programming (ROP), necessitating new types of exploitations. However, the current automated DOP exploitation tools are still more theoretical than practical and are hard to apply to real vulnerabilities.

7 Conclusion

This Edge of the Art report has presented new and novel tools and techniques developed since the previous versions of this report. It also summarizes common techniques in each tool category to provide the necessary background to understand the nuances and differences of each tool. As with previous reports, the intent is to effectively keep pace with the ever-expanding boundary that is the “edge” of the vulnerability discovery and exploitation discipline.

8 Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] "CMU Cylab, “A formal specification for BIL: BIL Instruction Language,” 02-Oct-2015. [Online]. Available: https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf.," [Online]. [Accessed 10 1 2020].

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

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

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

[29] *Y. Shoshitaishvili et al., “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 138–157, doi: 10.1109/SP.2016.17..*

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

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

[32] *C. Cifuentes, “Reverse Compilation Techniques,” Queensland University of Technology, 1994..*

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

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

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

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

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

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

[39] R. Stortz, "Revisiting 2000 cuts using Binary Ninja’s new decompiler," 17 4 2020. [Online]. Available: https://blog.trailofbits.com/2020/04/17/revisiting-2000-cuts-using-binary-ninjas-new-decompiler/. [Accessed 23 4 2020].

[40] Hex-Rays, "IDA: What's new in 7.5," Hex-rays, 19 May 2020. [Online]. Available: https://www.hex-rays.com/products/ida/news/7_5/. [Accessed 10 June 2020].

[41] Hex-Rays, "IDA: What's new in 7.4," 10 June 2020. [Online]. Available: https://www.hex-rays.com/products/ida/news/7_4/.

[42] NSA, "Ghidra Releases," NSA, February 2020. [Online]. Available: https://ghidra-sre.org/releaseNotes_9.1.2.html. [Accessed June 2020].

[43] River Loop Security, "Hashashin," Github, 2019. [Online]. Available: https://github.com/riverloopsec/hashashin. [Accessed 24 2 2020].

[44] R. O'Connell and R. Speers, "HASHASHIN: USING BINARY HASHING TO PORT ANNOTATIONS," River Loop Security, 2 December 2019. [Online]. Available: https://www.riverloopsecurity.com/blog/2019/12/binary-hashing-hashashin/. [Accessed 24 2 2020].

[45] R. O'Connell, "BINARY HASHING: MOTIVATIONS AND ALGORITHMS," River Loop Security, 26 November 2019. [Online]. Available: https://www.riverloopsecurity.com/blog/2019/11/binary- hashing-intro/. [Accessed 24 02 2020].

[46] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," in Proceedings 2020 Network and Distributed System Security Symposium, San Diego, CA, 2020 .

[47] Zynamics BinDiff, "Zynamics BinDiff," [Online]. Available: https://www.zynamics.com/bindiff.html. [Accessed 11 5 2020].

[48] Y. Duan, X. Li, J. Wang and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," 2020. [Online]. Available: https://www.ndss-symposium.org/wp- content/uploads/24311-slides.pdf. [Accessed 11 5 2020].

[49] Y. Duan, "DeepBinDiff," Github, 2020. [Online]. Available: https://github.com/yueduan/DeepBinDiff. [Accessed 11 5 2020].

[50] Abadi, M. et al. , "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2015. [Online]. Available: https://www.tensorflow.org/. [Accessed 11 5 2020].

[51] OpenBSD Project , "OpenSSH," 1999-2020. [Online]. Available: https://www.openssh.com/. [Accessed 11 5 2020].

[52] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences," in Proceedings of the 28th USENIX Security Symposium, Santa Clara CA, 2019..

[53] Lu, Kangjie, Aditya Pakki, and Qiushi Wu. "Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs." European Symposium on Research in Computer Security. Springer, Cham, 2019..

[54] K. Lu, A. Pakki and Q. Wu, "Detecting Missing-Check Bugs via Semanticand Context-Aware Criticalness and Constraints Inferences," 16 8 2019. [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/sec19_slides_lu.pdf. [Accessed 11 5 2020].

[55] UMNSEC, "Crix: Detecting Missing-Check Bugs in OS Kernels," Github, 8 2019. [Online]. Available: https://github.com/umnsec/crix. [Accessed 11 5 2020].

[56] O. Sandoval, "drgn," [Online]. Available: https://github.com/osandov/drgn. [Accessed 21 1 2020].

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

[58] D. Liew, C. Cadar, A. F. Donaldson and J. R. Stinnett, "Just fuzz it: solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, Tallinn, Estonia, 2019.

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

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

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

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

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

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

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

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

[67] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.,* vol. 51, no. 3, 2018.

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

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

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

[71] M. Brain, F. Schanda and Y. Sun, "Building Better Bit-Blasting for Floating-Point Problems," in *TACAS 2019: Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

[72] A. Luhrs, "New WinDbg available in preview!," Microsoft, 28 8 2017. [Online]. Available: https://blogs.windows.com/windowsdeveloper/2017/08/28/new-windbg-available-preview/. [Accessed 7 5 2020].

[73] J. Pinkerton, "Time Travel Debugging is now available in WinDbg Preview," Microsoft, 27 9 2017. [Online]. Available: https://blogs.windows.com/windowsdeveloper/2017/09/27/time-travel-debugging- now-available-windbg-preview/. [Accessed 13 5 2020].

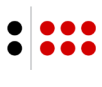
[74] A. Lamoureux, "Debugger Showcase," Vector 35, 06 05 2020. [Online]. Available: https://binary.ninja/2020/05/06/debugger-showcase.html#supported-platforms. [Accessed 11 05 2020].

[75] Vector 35, "Debugger," Github, 2020. [Online]. Available: https://github.com/Vector35/debugger. [Accessed 11 5 2020].

[76] P. K. F. F. J. K. Tobias Holl, "Kernel- Assisted Debugging of Linux Applications.," *In Reversing and Offensive- oriented Trends Symposium (ROOTS '18),* p. 9, 29-30 November 2018.

[77] H. Tobias, "ROOTS - Kernel-Assisted Debugging of Linux Applications," in *ROOTS*, Vienna, Austria, 2018.

[78] M. Gaasedelen, "What's New in Lighthouse v0.9," Ret2 Systems, 29 April 2020. [Online]. Available: https://blog.ret2.io/2020/04/29/lighthouse-v0.9/. [Accessed 10 June 2020].



[79] abenkhadra, "bcov," [Online]. Available: https://github.com/abenkhadra/bcov. [Accessed 15 June 2020].

[80] M. A. a. S. D. a. K. W. Ben Khadra, "ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE'20 (accepted)," in *ACM Press*, Sacramento, CA, USA, 2020.

[81] Formally Applied, "Function identification in stripped binaries revisited," 26 May 2020. [Online]. Available: https://blog.formallyapplied.com/2020/05/function-identification/.

[82] "Frida: A world-class dynamic instrumentation framework," [Online]. Available: https://frida.re/. [Accessed 22 June 2020].

[83] O. A. V. Ravnås, "Frida 10.5 Released," 25 August 2017. [Online]. Available: https://frida.re/news/2017/08/25/frida-10-5-released/.

[84] O. A. V. Ravnås, "Frida 12.8 Released," 18 December 2019. [Online]. Available: https://frida.re/news/2019/12/18/frida-12-8-released/.

[85] O. A. V. Ravnås, "Frida 12.9 Released," 19 May 2020. [Online]. Available: https://frida.re/news/2020/05/19/frida-12-9-released/.

[86] O. A. V. Ravnås, "Anatomy of a code tracer," 23 October 2014. [Online]. Available: https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8.

[87] "thread-tracer.js," 29 September 2014. [Online]. Available: https://github.com/frida/cryptoshark/blob/89af94760d679085129ddc71beba88ccee8654ba/agent/thread- tracer.js.

[88] "Stalker," [Online]. Available: https://frida.re/docs/stalker/. [Accessed 22 June 2020].

[89] "JavaScript API: Stalker," [Online]. Available: https://frida.re/docs/javascript-api/#stalker. [Accessed 22 June 2020].

[90] M. Zalewski, "afl," [Online]. Available: http://lcamtuf.coredump.cx/afl/.

[91] Y. Wang et al., “Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization,” in Proceedings 2020 Network and Distributed System Security Symposium, San Diego, CA, 2020.

[92] Google, "honggfuzz," [Online]. Available: https://google.github.io/honggfuzz/. [Accessed 21 1 2020].

[93] D. Mantz and B. Kauer, "Frizzer," Github, [Online]. Available: https://github.com/demantz/frizzer/. [Accessed 21 2 2020].

[94] R. Gopinath and A. Zeller, “Building Fast Fuzzers,” arXiv:1911.07707 [cs], Nov. 2019.

[95] 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, doi: 10.14722/ndss.2019.23263.

[96] C. Aschermann, P. Jauernig, T. Frassetto, A.-R. Sadeghi, T. Holz, and D. Teuchert, “Fishing for Deep Bugs with Grammars,” presented at the NDSS 2019, San Diego, CA, 2019, p. 15..

[97] P. Godefroid, M. Y. Levin, D. Molnar and Microsoft, "SAGE: Whitebox Fuzzing for Security Testing," *acmqueue,* vol. 10, no. 1, 11 January 2012.

[98] Defense Advanced Research Project Agency (DARPA), "Cyber Grand Challenge (CGC) (Archived)," [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge. [Accessed 22 1 2020].

[99] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kafl: Hardware-assisted feedback fuzzing for OS kernels,” in 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, 2017, pp. 167–182. .

[100] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, “HFL: Hybrid Fuzzing on the Linux Kernel,” in Proceedings 2020 Network and Distributed System Security Symposium, San Diego, CA, 2020.

[101] S. Schumilo, A. A. Cornelius Aschermann, S. Wörner and T. Holz., "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing," in *Symposium on Network and Distributed Systems Security (NDSS), 2020.*, San Diego, 2020.

[102] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating Fuzz Testing,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS ’18, Toronto, Canada, 2018, pp. 2123–2138, doi: 10.1145/3243734.3243804.

[103] "IJON: Exploring Deep State Spaces via Fuzzing," in *41st IEEE Symposium on Security and Privacy*, 2020.

[104] "IJON Space Explorer," [Online]. Available: https://github.com/RUB-SysSec/ijon. [Accessed 20 06 2020].

[105] Chair for Systems Security, RUB-SysSec, "Nautilus," Github, 2019. [Online]. Available: https://github.com/RUB-SysSec/nautilus. [Accessed 16 4 2020].

[106] nautilus-fuzz, "Nautilus 2.0," Github, 4 2020. [Online]. Available: https://github.com/nautilus-fuzz/nautilus. [Accessed 16 4 2020].

[107] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. Vdf: Targeted evolutionary fuzz testing of virtual devices. In Symposium on Recent Advances in Intrusion Detection (RAID), 2017. .

[108] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In IEEE Symposium on Security and Privacy, 2019..

[109] D. Vyukov, "syzkaller: adventures in continuous coverage-guided kernel fuzzing," in *BlueHat*, Tel Aviv, Israel, 2020.

[110] "syzkaller unsupervised coverage-guided kernel fuzzer," [Online]. Available: https://github.com/google/syzkaller. [Accessed 19 6 2020].

[111] "afl," [Online]. Available: http://lcamtuf.coredump.cx/afl/.

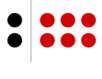
[112] N. Ben-Simon and Y. Alon, "Check Point Research," Check Point Research, [Online]. Available: https://research.checkpoint.com/2020/bugs-on-the-windshield-fuzzing-the-windows-kernel/. [Accessed 19 06 2020].

[113] Joint Task Force Transformation Initiative. September 2012. Guide for Conducting Risk Assessments. In NIST Special Publication 800-30 Revision 1. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD. .

[114] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman and A. Shubina, "Exploit Programming: From Buffer Overflows to “Weird Machines” and Theory of Computation," *;login:,* December 2011 .

[115] 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, doi: 10.1109/SP.2014.22.

[116] E. Bosman and H. Bos, “Framing Signals - A Return to Portable Shellcode,” in 2014 IEEE Symposium on Security and Privacy, 2014, pp. 243–258, doi: 10.1109/SP.2014.23.



[117] K. Z. Snow, F. Monrose, 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.

[118] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks,” in 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, 2016, pp. 969–986, doi: 10.1109/SP.2016.

[119] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block Oriented Programming: Automating Data-Only Attacks,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto Canada, 2018, pp. 1868–1882, doi: 10.114.

[120] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code- reuse attack,” in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’11, Hong Kong, China, 2011, p. 30.

[121] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit Object- oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications,” in 2015 IEEE Symposium on Security and Privacy, San Jose, CA.

[122] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, 2016.

[123] HexHive, "BOPC," Github, 2019. [Online]. Available: https://github.com/HexHive/BOPC. [Accessed 19 03 2020].

[124] S. D’Antoine, “Automatic Exploit Generation,” presented at the CanSec West 2016, 2016. .

[125] 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. .

[126] Y. Chen and X. Xing, “SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel,” in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London United Kingdom, Nov. 2019, pp. 1707–1722.

[127] H. Hu, "Advanced DOP," [Online]. Available: https://huhong-nus.github.io/advanced-DOP/. [Accessed 11 02 2020].

[128] C. Z. Leong, "DOP-StaticAssist," Github, [Online]. Available: https://github.com/melynx/DOP-StaticAssist. [Accessed 11 02 2020].

[129] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block Oriented Programming: Automating Data-Only Attacks,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto Canada, 2018, pp. 1868–1882.

[130] 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.

[131] S. Heelan, T. Melham, and D. Kroening, “Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters,” in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London United Kingdom, 2019.

[132] W. Wu, Y. Chen, X. Xing, and W. Zou, “KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities,” p. 19. .

[133] ww9210, "kepler-cfhp," Github, 2019. [Online]. Available: https://github.com/ww9210/kepler-cfhp. [Accessed 11 5 2020].

[134] S. Heelan, "HeapLayout," Github, [Online]. Available: https://github.com/SeanHeelan/HeapLayout. [Accessed 21 1 2020].

[135] S. Heelan, "Automation in Exploit Generation with Exploit Templates," 5 3 2019. [Online]. Available: https://sean.heelan.io/2019/03/05/automation-in-exploit-generation-with-exploit-templates/. [Accessed 22 1 2020].

[136] K. Zeng, "NDSS 2020 Not All Coverage Measurements Are Equal: Fuzzing for Input Prioritization," 6 4 2020. [Online]. Available: https://www.youtube.com/watch?v=Fud0v0ppCOo. [Accessed 11 5 2020].

[137] A. Zeller. [Online]. Available: https://www.st.cs.uni-saarland.de/dd/. [Accessed 21 1 2020].

[138] M. Ammar.