



AFRL-RI-RS-TP-2022-008

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 6

TWO SIX LABS

JUNE 2022

TECHNICAL PAPER

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

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

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

AFRL-RI-RS-TP-2022-008 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

CHAD C. DESTEFANO
Work Unit Manager

/ S /

JAMES S. PERRETTA
Deputy Chief
Information Warfare Division
Information Directorate

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

REPORT DOCUMENTATION PAGE

1. REPORT DATE		2. REPORT TYPE		3. DATES COVERED	
JUNE 2022		TECHNICAL PAPER		START DATE JULY 2021	END DATE DECEMBER 2021
4. TITLE AND SUBTITLE Edge of the Art in Vulnerability Research Version 6					
5a. CONTRACT NUMBER FA8750-19-C-0009		5b. GRANT NUMBER N/A		5c. PROGRAM ELEMENT NUMBER 62303E	
5d. PROJECT NUMBER		5e. TASK NUMBER		5f. WORK UNIT NUMBER R2PB	
6. AUTHOR(S) Will Huiras, Irwin Ong, and Jared Ziegler					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Two Six Labs 901 N Stuart Street, Suite 1000 Arlington VA 22203				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA 525 Brooks Road Rome NY 13441-4505			10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RI-RS-TP-2022-008
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 36030 Date Cleared: Mar 28, 2022					
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	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U		SAR	
				18. NUMBER OF PAGES 94	
19a. NAME OF RESPONSIBLE PERSON CHAD DESTEFANO				19b. PHONE NUMBER (Include area code) 315-330-4286	

Contents

1	Introduction	1
1.1	Scope	1
2	Techniques and Workflows	3
2.1	An Observational Investigation of Reverse Engineers' Processes	4
3	Static Analysis	6
3.1	Ghidraal	7
3.2	GhiHorn	11
3.3	Manticore UI	16
4	Dynamic Analysis	19
4.1	bcov	20
4.2	ColdSnap	28
4.3	Format Fuzzer	31
4.4	FuzzBuilder	34
4.5	Fuzzolic	39
4.6	Go-Fuzz	44
4.7	Gramatron	48
4.8	NyxNet	61
4.9	REVEN	64
4.10	ZAFL	70

5	Appendix	73
5.1	Resources	73
5.2	Tools Criteria	73
5.3	Techniques Criteria	74
5.4	Tool and Technique Categories	74
5.5	Static Analysis Technical Overview	75
5.5.1	Disassembly	75
5.5.2	Decompilation	79
5.5.3	Static Vulnerability Discovery	79
5.6	Dynamic Analysis Technical Overview	80
5.6.1	Debuggers	80
5.6.2	Dynamic Binary Instrumentation (DBI)	81
5.6.3	Dynamic Fuzzing Instrumentation	81
5.6.4	Memory Checking	81
5.6.5	Dynamic Taint Analysis	82
5.6.6	Symbolic and Concolic Execution	82

Introduction

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

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

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

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

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

1.1 Scope

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

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

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

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

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

Techniques and Workflows

Much research in vulnerability research focuses on developing new and better tools. While tools are important, just as important or more is the way an analyst approaches a problem. This is one reason why expertise is so crucial. In order to meet the world's software analysis demands, it is critical to discover ways to make doing VR more efficient. A growing number of researchers are seeking to develop improved techniques to make individual analysts more effective or to find workflows for enabling teams of analysts (including analysts of varied skill levels) to work together effectively. Topics include VR methodologies, organization, collaboration, human-machine teaming, tool evaluation, and workflows.

In this edition of the Edge of the Art, we focus on a study of the practices used by successful reverse engineers.

2.1 An Observational Investigation of Reverse Engineers' Processes

In contrast to the more traditional task of program comprehension, there is limited theoretical understanding of the reverse engineering (RE) process itself, which could enable the creation of better RE tools. To address this shortcoming, a recent paper by Votipka et al. [1] reports on an observational interview study involving 16 expert reverse engineers (REs). The authors include a narrative description of the interviews with many short excerpts from the interviewees. Based on this study, the authors develop two main results: a three-phase model for the RE process and a set of guidelines for RE tool design (see the next two sections).

The expert RE subjects of the study were asked to re-enact the process of reverse engineering some program they had past experience with, describe each step, and answer questions posed by an interviewer. In each case, the goal of RE was either vulnerability discovery or malware analysis, but subjects with either goal in mind tended to follow the same process.

This work extends Bryant [2], which developed a sense-making model for reverse engineering, where REs generate hypotheses from prior experience and cyclically attempt to (in)validate these hypotheses, generating new hypotheses in the process. Also in previous work, Votipka et al. interviewed white-hat hackers and testers to determine their vulnerability discovery process [3]. That work identified RE as an important part of vulnerability discovery, leading to the topic of this paper.

The following two sections outline the two main results from the study: a theoretical model of the RE process and guidelines for RE tool design.

A model for the RE process

The model comprises three phases: (1) overview, (2) sub-component scanning, and (3) focused experimentation.

In the *overview* phase, the RE seeks a broad overview of the program's functionality, perhaps by reviewing the list of strings, APIs used, or the program metadata. The RE might run the program and observe its behavior and perhaps use static analysis tools. This phase is included in the RE process, but is not usually considered part of the more general program comprehension process. During the normal program comprehension process, the (non-reverse-)engineer would typically already understand the basic functionality of the program and have access to resources such as documentation unavailable to the RE. During this phase, the RE establishes initial hypotheses and questions which focus investigation on certain sub-components of the program, leading to the second phase of the RE process.

In the *sub-component scanning* phase, the RE performs a more-focused review on program subcomponents, which produces more refined hypotheses and questions. The RE continues to scan APIs, strings, and UI elements for any suspicious or notable signs based

on experience—e.g., a call to the Win32 API function `GetProcAddress` because it is commonly used for obfuscation. The RE might focus on specific data-flow or control-flow paths and identify specific questions that require concrete information to answer.

In the third phase, *focused experimentation*, REs attempt to answer questions raised in the previous phases through execution or in-depth static analysis. These analysis results are then fed back into the second phase for further investigation, which produces more questions, leading to more focused experimentation, forming a loop. In this way, questions and hypotheses are iteratively refined and investigated until the RE achieves her overall goals. Techniques used during this third phase might include fuzzing, comparison against a reference (e.g., cryptographic) implementation, line-by-line reading of the source code, and symbolic execution. Whereas the first two phases tend to focus on static analysis, the third phase uses more dynamic analysis.

Guidelines for RE tool design

Based on their study, the authors developed several guidelines for designing RE tools:

- Match interaction with analysis phases. Design tools with the analysis phases discussed above (overview, sub-component scanning, focused experimentation) in mind.
- Present input and output in the context of code. Tightly-couple analysis tools with the disassembler or decompiler code view. “In almost all cases, the tools REs choose to use provide a simple method to connect results back to specific lines of code” [1].
- Allow data transfer between static and dynamic contexts. REs regularly switch between static and dynamic representations, so tools should enable this. E.g., allow stepping through decompiled code.
- Allow selection of analysis methods. REs choose to use many different analysis methods based on their experience of the trade-offs, so tools should enable this. E.g., the HexRays decompiler allows users to toggle between a potentially imprecise, but easier to read, decompiled program view and the more complex disassembled view [4].
- Support readability improvements. Fundamentally, the purpose of RE is to recover human meaning, so tools should enable that by allowing users to add notes or change naming to encode semantic information as it is discovered. The main purpose of most of the tools used by REs was to improve code readability (e.g., decompilers, IDA’s Lumina server). “Additionally, most REs performed several manual steps specifically to improve readability, such as renaming variables, taking notes, and reconstructing data structures” [1].
- Also, RE tool designers should consider the exploratory visual analysis (EVA) literature. “EVA considers situations where analyses search large datasets visually to summarize their main characteristics” [5, 6].

Static Analysis

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

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

3.1 Ghidraal

Reference Link	
Target Type	Binary
Host Operating System	Linux; MacOS; Windows
Target Operating System	Linux; MacOS; Windows
Host Architecture	GraalVM Java Virtual Machine
Target Architecture	Many (any supported by Ghidra)
Initial Release	06/15/2020
License Type	Open Source (DFARS 252.227-7013 and 252.227-7014)
Maintenance	Maintained by Jason P. Leasure

Overview

Ghidraal [7] is an extension for Ghidra that enables a variety of programming languages to be used to write scripts in Ghidra. These include: Python 3, Ruby, R, and Javascript. By default, Ghidra supports only Java and a limited Python 2 implementation through Jython. By building upon the GraalVM [8] project, Ghidraal unlocks the full power of these additional languages, making Ghidra scripting potentially much more accessible and capable than in the past.

Design and Implementation

Ghidraal is built on top of GraalVM. Oracle's GraalVM project is a Java Development Kit (JDK) "designed to accelerate the execution of applications written in Java and other JVM

languages while also providing runtimes for JavaScript, Ruby, Python, and a number of other popular languages.” In practice, this means that these languages can run together in the same application.

Ghidra itself is, of course, written in Java. In order to support GraalVM (and, by extension, Ghidraal), it is necessary to run Ghidra under the GraalVM runtime. The Ghidraal README guides the user through this.

The installation was relatively straightforward, though there were a few hiccups. Ghidraal ships with an environment script, `env.sh`, that automatically installs GraalVM (if necessary) and sets up the environment to use GraalVM as the default Java runtime. This works as advertised, but when we attempted to build the Ghidraal project from Github (which requires Gradle 5+), we were unsuccessful. Rather than chase these down the errors, we settled on a release version. The most version available claimed to support Ghidra version 9.2.3, but we proceeded using our installed version, 10.0.1. We executed Ghidra from within the GraalVM environment, which worked just fine. When we attempted to import the extension package zip file, we received a wrong version message. However, unzipping the package, changing the version field in the `extension.properties` file, re-zipping, and importing installed the extension with no apparent issues.

Ghidraal includes several simple example scripts in each of the four scripting languages for getting started. (See figure 3.1.) Each executed successfully. The full Ghidra scripting API is available for each language. It’s also possible to install packages for the supported languages. The Python 3 package “panda” installed successfully using a command from within the GraalVM environment. That stands in contrast with Ghidra’s Jython implementation, which does not support external packages.

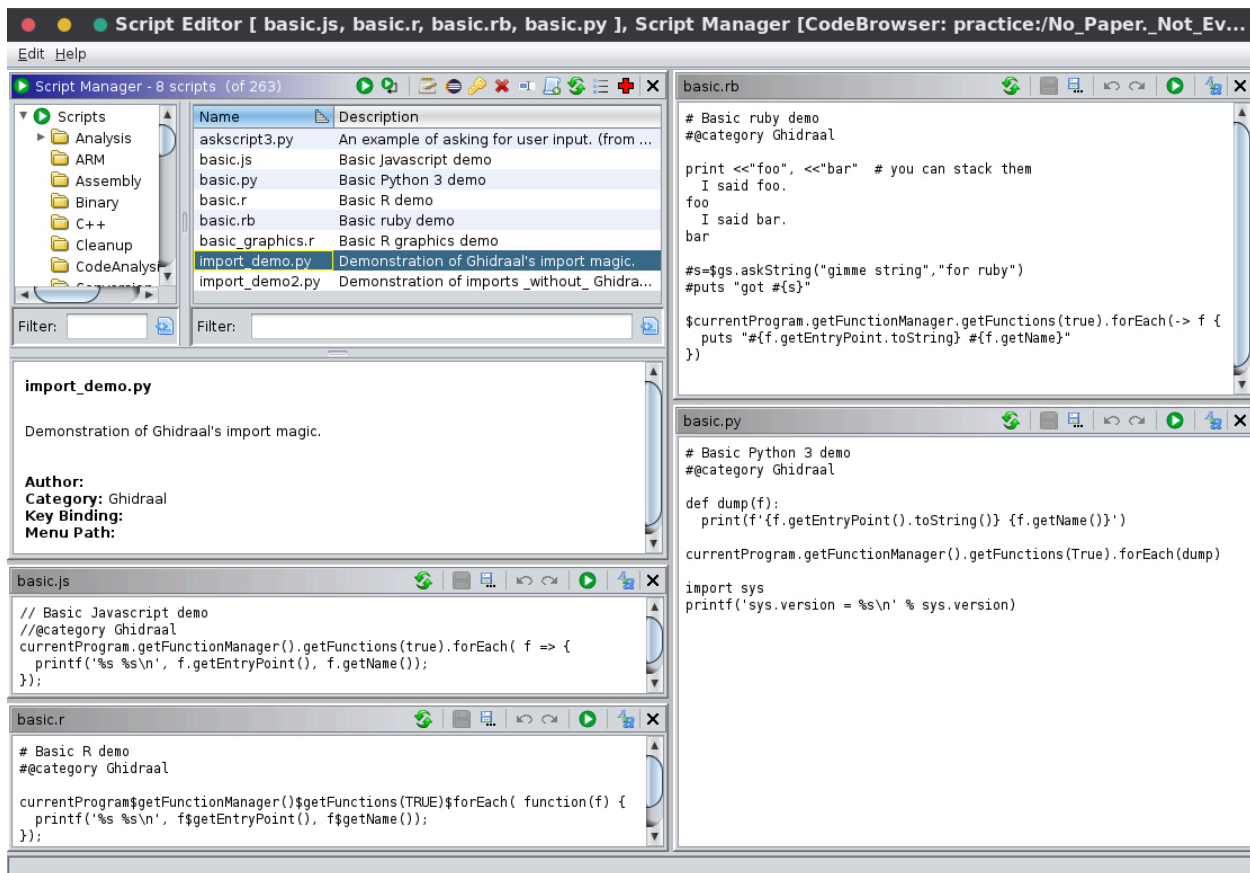


Figure 3.1: Some simple scripts that come with Ghidraal.

Ghidraal also provides an interactive interpreter window for each of the languages, as shown in figure 3.2. The interpreters can be run simultaneously and include convenience features such as auto-completion, allowing for interactive discovery of Ghidra functionality without having to constantly reference the Ghidra API documentation.

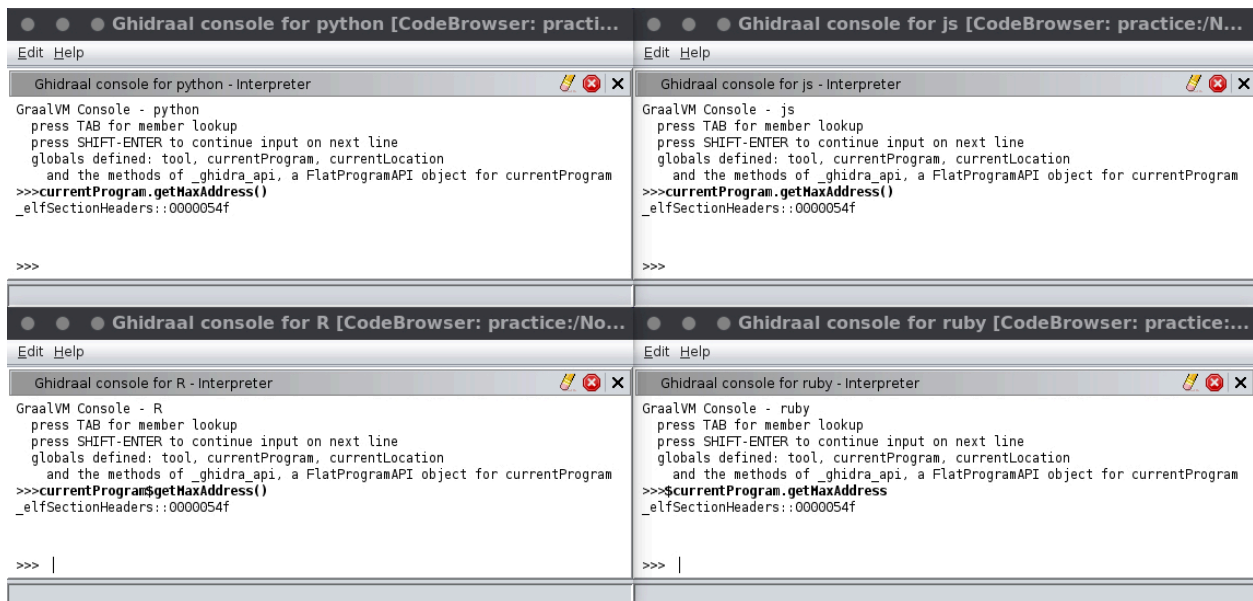


Figure 3.2: Ghidraal includes interactive interpreters for each supported language.

Use Cases and Limitations

Ghidraal is an impressive demonstration of the power of GraalVM and provides an apparently seamless integration of multiple modern scripting languages into Ghidra. The Python 3 support alone makes it worth looking into, since Python 2 has been deprecated at this point and this enables the use of modern Python libraries from within Ghidra.

The biggest limitation is the need to run Ghidra from within GraalVM. Pinning to the GraalVM JDK, which is not commonly installed, means that Ghidraal won't be useful in all environments. Moreover, scripts written in these languages will not be portable to vanilla Ghidra users. Also, the library of Python scripts that ship with Ghidra won't work with Ghidraal's Python 3 (and vice versa.) That said, for individual users and teams, Ghidraal opens the door to more rapid, more powerful Ghidra script development and a choice of scripting languages.

3.2 GhiHorn

Reference Link	https://github.com/CERTCC/kaiju
Target Type	Binary
Host Operating System	Linux; MacOS; Windows
Target Operating System	N/A
Host Architecture	x86, x86_64
Target Architecture	Ghidra P-Code compatible code
Initial Release	07/12/2021
License Type	Open Source (BSD)
Maintenance	Maintained by CERTCC

Overview

GhiHorn is a Ghidra extension maintained by CERTCC that supports SMT solver integration [9]. For this report, we'll be focusing on the path analysis aspect of GhiHorn. GhiHorn's path analyser aims to determine whether a path is "feasible" between two virtual addresses (i.e. there is a valid set of states that would lead code execution down said path). This type of path analysis fills a void left in modern binary path analysis by fuzzers, whose coverage data (which is used to determine path feasibility) is often dictated by pseudo-random input mutations. While GhiHorn looks promising, it still appears to be in a very experimental state, and is not likely to be useful for analysing real-world binaries without further development work.

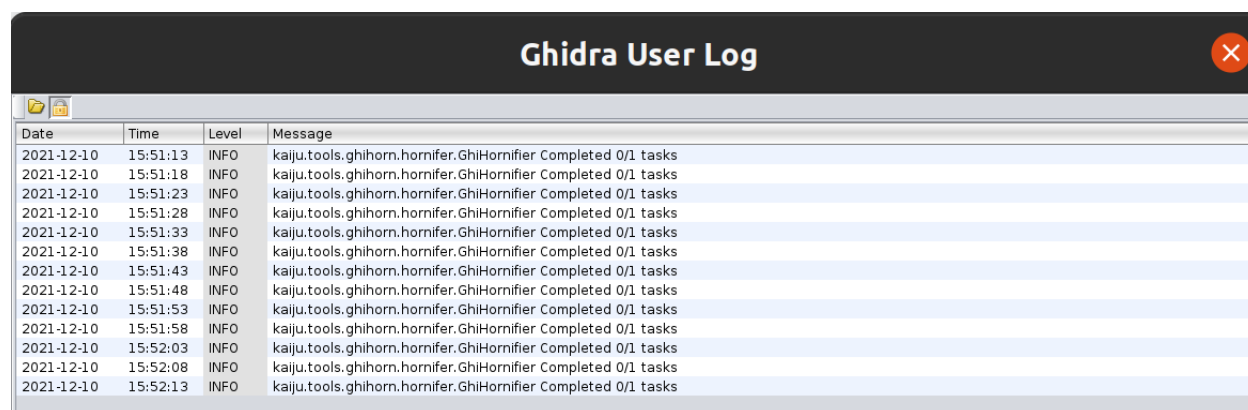
Design and Implementation

GhiHorn works by converting Ghidra's P-Code and Varnode representations into Horn clauses, structured to meet the SMT-LIB specification. By conforming to this specification, the tool's statements may then be solved by third-party solvers such as Z3.

GhiHorn operates on Ghidra code at the basic-block level. For each basic block, state transitions within each block are encoded into Z3 statements. These statements are then merged together into a single Z3 statement, representing all state changes that occur within the basic block. The resultant collection of Z3 statements are then merged into a set, using the control flow graph provided by Ghidra as a linkage map. The antecedent of each Z3 statement is set to the statement(s) corresponding to the basic block(s) that feed into the basic block from which the initial Z3 statement was derived. A similar procedure is followed to assign consequents, except using the basic blocks into which the subject basic blocks feed. The result is a set of Z3 statements that are linked in a way that mimics the linkage described in the program's control flow graph.

Installation of GhiHorn is fairly straightforward. Z3 and its Java bindings must be installed on your host and in a location that Ghidra can access. Otherwise, GhiHorn itself is available as a Ghidra extension and can be installed like any other Ghidra extension.

Using the tool through Ghidra's UI is simple: enable the extension and open the GhiHorn window. Using the tool, however, was not so simple. The Windows calculator application was our first target for GhiHorn's path analyzer. However, every path query we attempted ended up causing the GhiHorn extension to hang. The final query was left to run over 24 hours, but the task assigned to that query failed to complete in that time (figure 3.3).



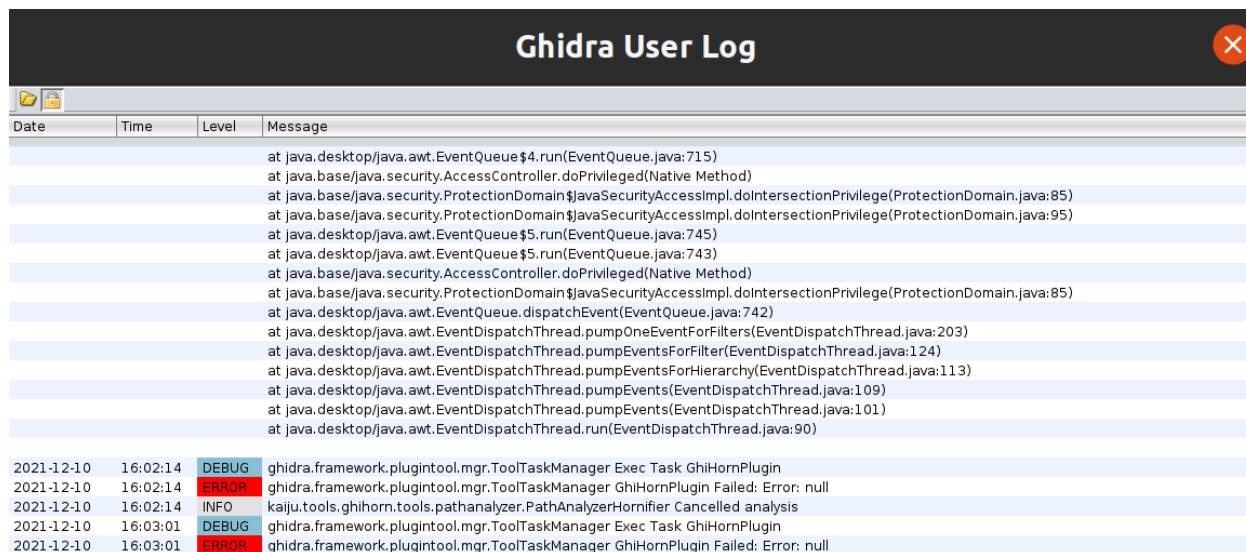
Date	Time	Level	Message
2021-12-10	15:51:13	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:18	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:23	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:28	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:33	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:38	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:43	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:48	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:53	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:51:58	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:52:03	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:52:08	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks
2021-12-10	15:52:13	INFO	kaiju.tools.ghihorn.hornifer.GhiHornifier Completed 0/1 tasks

Figure 3.3: GhiHorn hanging on calc.exe from a Windows 7 host.

The next attempt at using GhiHorn involved compiling a simple C program with Clang and analyzing the resultant binary on the same Linux host that Ghidra was running. However, attempting to use GhiHorn against that binary resulted in an error (figure 3.4). Around the time of writing this report, others were reporting similar issues.

To bypass this issue, we compiled the same program but compiled with Clang on an OSX

host. When attempting to use GhiHorn to analyze this binary, GhiHorn threw an error after attempting to analyze the P-Code for a function from an external library (printf() in this case). However, after further reviewing of some GhiHorn reference material, handling external functions may be the responsibility of the user.



Date	Time	Level	Message
			at java.desktop/java.awt.EventQueue\$4.run(EventQueue.java:715)
			at java.base/java.security.AccessController.doPrivileged(Native Method)
			at java.base/java.security.ProtectionDomain\$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:85)
			at java.base/java.security.ProtectionDomain\$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:95)
			at java.desktop/java.awt.EventQueue\$5.run(EventQueue.java:745)
			at java.desktop/java.awt.EventQueue\$5.run(EventQueue.java:743)
			at java.base/java.security.AccessController.doPrivileged(Native Method)
			at java.base/java.security.ProtectionDomain\$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:85)
			at java.desktop/java.awt.EventQueue.dispatchEvent(EventQueue.java:742)
			at java.desktop/java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:203)
			at java.desktop/java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.java:124)
			at java.desktop/java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:113)
			at java.desktop/java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:109)
			at java.desktop/java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:101)
			at java.desktop/java.awt.EventDispatchThread.run(EventDispatchThread.java:90)
2021-12-10	16:02:14	DEBUG	ghidra.framework.pluginool.mgr.ToolTaskManager Exec Task GhiHornPlugin
2021-12-10	16:02:14	ERROR	ghidra.framework.pluginool.mgr.ToolTaskManager GhiHornPlugin Failed: Error: null
2021-12-10	16:02:14	INFO	kaiju.tools.ghihorn.tools.pathanalyzer.PathAnalyzerHornifier Cancelled analysis
2021-12-10	16:03:01	DEBUG	ghidra.framework.pluginool.mgr.ToolTaskManager Exec Task GhiHornPlugin
2021-12-10	16:03:01	ERROR	ghidra.framework.pluginool.mgr.ToolTaskManager GhiHornPlugin Failed: Error: null

Figure 3.4: GhiHorn failing to analyze a binary file that calls external functions.

Finally, we compiled a version of the same code but with all external function calls removed. With all of the external function calls removed, GhiHorn was able to successfully execute our desired path queries. For this binary,, GhiHorn was able to identify feasible paths (or lack thereof) between two virtual addresses, as well as provide concrete values that direct code execution down those paths (figure 3.5).

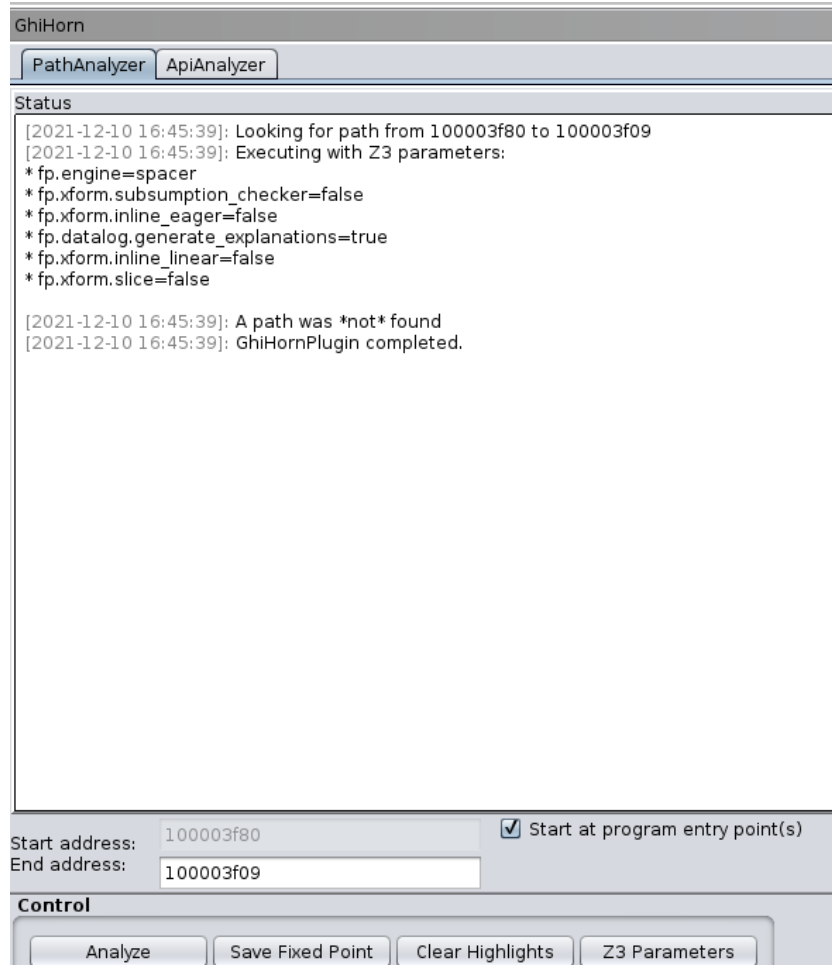


Figure 3.5: Running a successful GhiHorn query that determines there is no feasible path between two virtual addresses (even though there is a path between both addresses in Ghidra’s CFG).

Use Cases and Limitations

Determining reachability is an important aspect of vulnerability research. Unreachable bugs are much less valuable than those that can be reached by an attacker. Determining reachability is one of the major strengths of fuzzing. The fact that a fuzzer is able to trigger a vulnerability in a way to produce a measurable side effect (e.g. a program crash) almost certainly means that the vulnerability is reachable. Additionally, if a fuzzer is configured to emit coverage data as well, then the coverage map created by aggregating all of the coverage data will show what code paths the fuzzer explored. If a vulnerability exists in one of those code paths, then it’s reasonable to suspect that the vulnerability is reachable. However, fuzzers do have some limitations when it comes to calculating reachability. The fact that a fuzzer can’t reach a particular code path doesn’t mean that said code path is unreachable.

GhiHorn’s usefulness can be found in situations in which a vulnerability has been identified in compiled code, but it is unknown how to reach that vulnerability. A vulnerability

researcher could use GhiHorn to query whether there is a reachable path from an identified program entry point to a virtual address corresponding to a vulnerability (e.g. the address of a function, a particular basic block, a function call, etc). Whereas much of this process is typically done manually, GhiHorn provides the option of using an SMT solver (Z3 in this case) to perform most of the heavy lifting in calculating values that meet the constraints of the identified code path.

The largest limitation with GhiHorn, in its current state, is its usability. Of the experiments done for this review, the only successful application of the Ghidra extension was on highly contrived sample code that was written with GhiHorn specifically in mind. While this level of usability may be sufficient for the purposes of testing and further developing the tool, it is not sufficient for using the tool against production code. Additionally, the requirement that users provide their own simulations of code not statically compiled into the target binary significantly increases the level of effort required to use GhiHorn against binaries that utilize external libraries for functionality.

3.3 Manticore UI

Reference Link	https://github.com/trailofbits/MUI
Target Type	Binary
Host Operating System	MacOS, Linux
Target Operating System	Linux, EVM
Host Architecture	N/A
Target Architecture	x86, x86_64, aarch64, ARMv7, EVM
Initial Release	7 July 2021
License Type	Open Source (No license specified)
Maintenance	Trail of Bits, last commit 15 Sep 2021

Overview

Manticore User Interface (MUI) is a Binary Ninja plugin that gives users an intuitive, visual interface for working with symbolic execution[10][11]. The plugin supports basic symbolic execution workflows for both Linux and Ethereum Virtual Machine (EVM) targets.

Design and Implementation

MUI offers a context menu and extra actions for interacting with the Manticore backend, as well as UI widgets for displaying state information. The common symbolic execution workflow of “try to reach this address while avoiding these addresses” is enabled via MUI actions reachable either by right-click or control palette. In addition, actions can be set to hotkeys (Binary Ninja allows keybindings to be registered for any registered plugin). The

“solve with Manticore” action opens a dialog that gives users supplementary options such as specifying symbolic input sources.

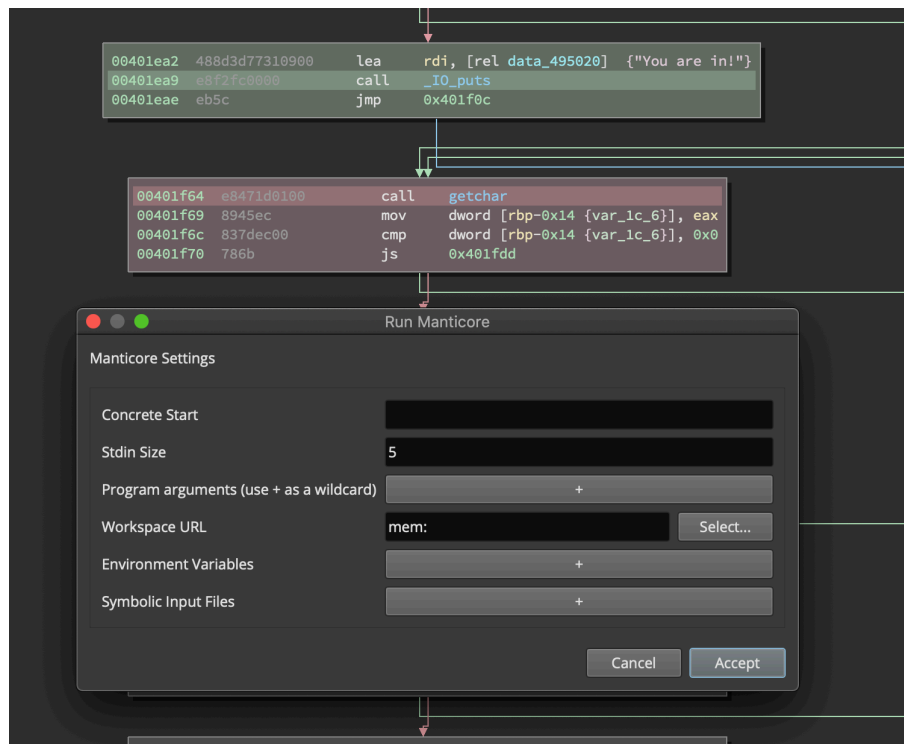


Figure 3.6: Specifying a custom length and other options via the “Solve with Manticore” dialog

Once solving starts, two additional MUI widgets become useful: “state list” and “state graph explorer”. The state list is fairly self explanatory, but the State Graph Explorer leverages Binary Ninja’s FlowGraph API to present a graphical representation of information relating to symbolic execution states. This gives users a more intuitive understanding of where states were forked, and how particular states are reached. While this may seem like visual gravy, this sort of display is useful for teaching new users how Manticore works and basic troubleshooting. Integrating tools into reverse engineering frameworks, while not a new concept[12], provides a powerful way to contextualize analysis and enhance usability.

MUI also allows hooking particular locations, which gives users more control over their analyses. This gives the user greater control over Manticore’s internal functionality. This feature behaves as advertised on the basic examples provided.

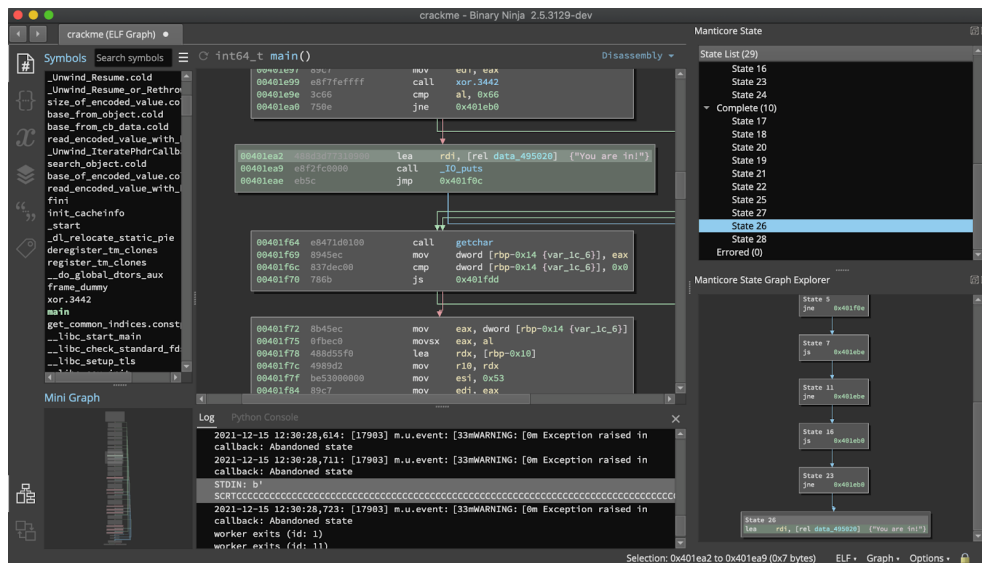


Figure 3.7: Exploring provenance of a successful end state for a simple crackme

MUI also offers a novel feature- it can leverage Manticore support for EVM smart contract analysis with a second plugin, Ethersplay. Ethersplay allows smart contract disassembly to be displayed in Binary Ninja, and MUI provides an interface for exploration and symbolic execution. Manticore already supports reasoning over smart contracts, so MUI primarily helps with understanding state flows and makes certain options more discoverable. This support is experimental, and in our testing Ethersplay was easier to set up and use than the Manticore integration for smart contracts.

Use Cases and Limitations

The primary use case for integration of a tool like Manticore with Binary Ninja is to streamline exploration of a target providing the user with an intuitive interface to Manticore from within their existing reverse engineering framework. MUI performs this function well for basic manual workflows, but it is not without quirks. When things go wrong, it can be difficult for a new user to determine the cause of the problem. While working through the Binary Ninja UI is helpful for small adjustments, it is easier to troubleshoot more complex Manticore problems with a standalone Python script. Another challenge is that, in some cases, Manticore may fail when it encounters an instruction that it does not support. That said, the State List and Graph Explorer are helpful when things go as planned. Just like many symbolic execution tools, there are a number of factors that can cause issues, but when it works the results can feel like magic.

Overall, MUI is a functional integration that can enable powerful interactive workflows, but the learning curve is still steep.

Dynamic Analysis

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

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

4.1 bcov

Reference Link	https://github.com/abenkhadra/bcov
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	N/A
Target Architecture	x86_64
Initial Release	29 April 2020
License Type	Open Source (MIT)
Maintenance	@abenkhadra, last commit 25 Jan 2021

Overview

bcov is a static instrumentation tool for x86_64 block coverage collection[13][14]. The primary aim of bcov is to improve the efficiency of instrumentation and to accurately scale to real-world applications.

Design and Implementation

Bcov works directly on x86_64 ELF binaries without compiler support. It implements a trampoline-base approach to insert probes in specific locations to trace basic block coverage. Some specific solutions are: probe placement and pruning based on superblock (SB) dominator graph (DG) analysis, efficient transparent detour placement, and precise control flow graph (CFG) analysis via sliced microexecution.

The instrumentation workflow includes three phases: module and function-level analysis, probe location identification and code and data sizes requirement estimation based on instrumentation policy, and the patching phase that introduces the new code and data to the target binary.

One of the major focus areas of bcov is efficient instrumentation, which includes both minimizing the number and overhead of probes. To minimize the number of probes, bcov first performs superblock dominator graph (SB-DG) analysis. The SB-DG analysis describes a function's CFG in such a way that a single probe can reflect the coverage of multiple blocks by using dominance frontiers, where the requirements for probes vary depending on the policy chosen. This is used to implement two different coverage policies which serve two common use cases of coverage information:

- In the “leaf-node” policy, the goal is to be able to reflect 100% block coverage with the minimal number of probes.
- The “any-node” policy reflects the standard notion of being able to describe whether any given basic block was covered or not in a given execution. See the figure below (fig. 4.1) from the authors' paper for an illustration.

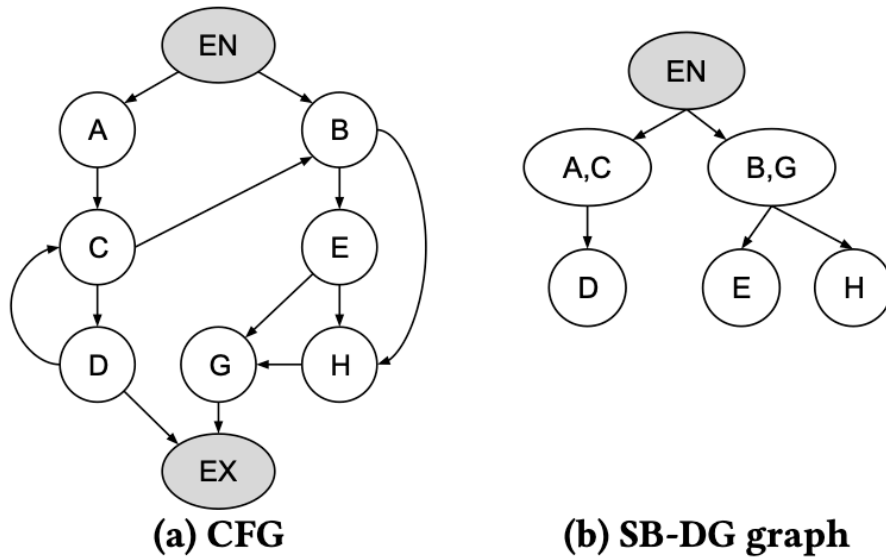


Figure 3: An example CFG and its corresponding SB-DG. First, pre-dominator and post-dominator trees are constructed and merged in a dominator graph (DG). SCCs in DG represent nodes in SB-DG. In the *leaf-node* policy, only leaf nodes in SB-DG, namely, D, E, and H, need to be probed. In the *any-node* policy, either A or C need to be additionally probed. EN and EX are *virtual* nodes commonly used to simplify dominance analysis.

Figure 4.1: The authors' example description of the SB-DG and two policies

This probe selection process is supplemented by bcov's detour placement strategy, which minimizes overhead by selecting where to place instrumentation based on the basic blocks within a given SB. The authors developed a series of heuristics to rank instruction types based on the amount of overhead caused by inserting a detour. For example, a long call instruction of 5 bytes is more easily patched than a 2-byte jump instruction. The use of trampolines combined with this flexibility to choose instructions within a given superblock reduce the amount of changes required to statically add coverage instrumentation. Figures [4.2, 4.3, 4.4, 4.5, 4.6, and 4.7] show three different instrumentation placements based on bcov's strategies.

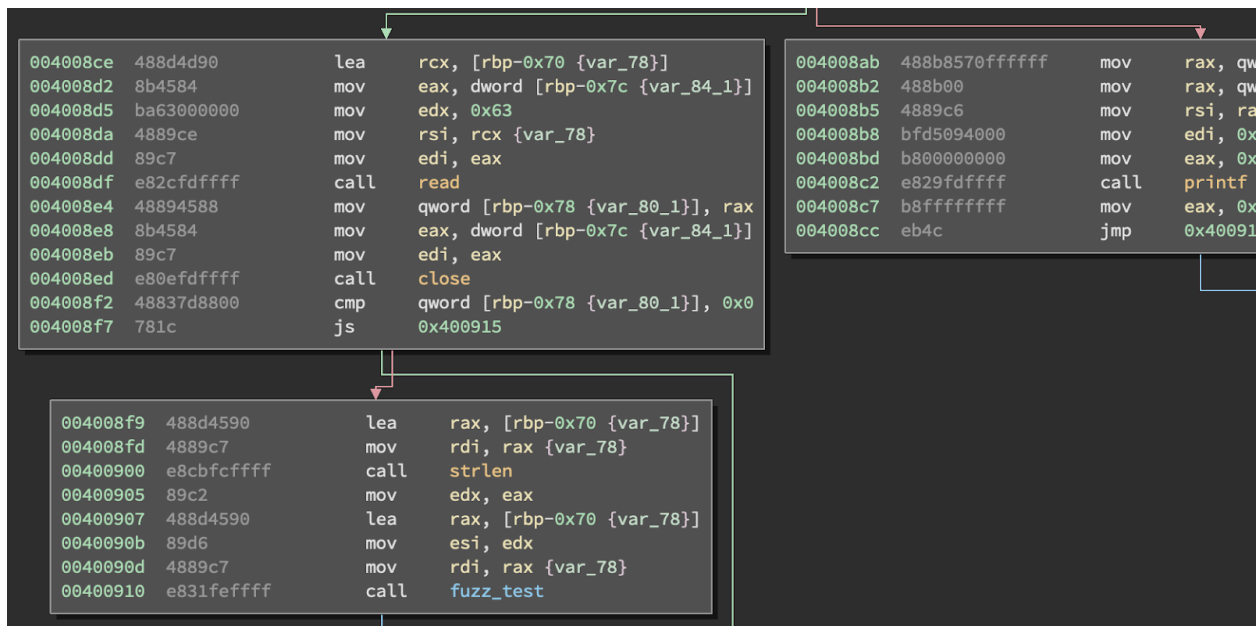


Figure 4.2: Before probe placement via function hooking (no overhead)

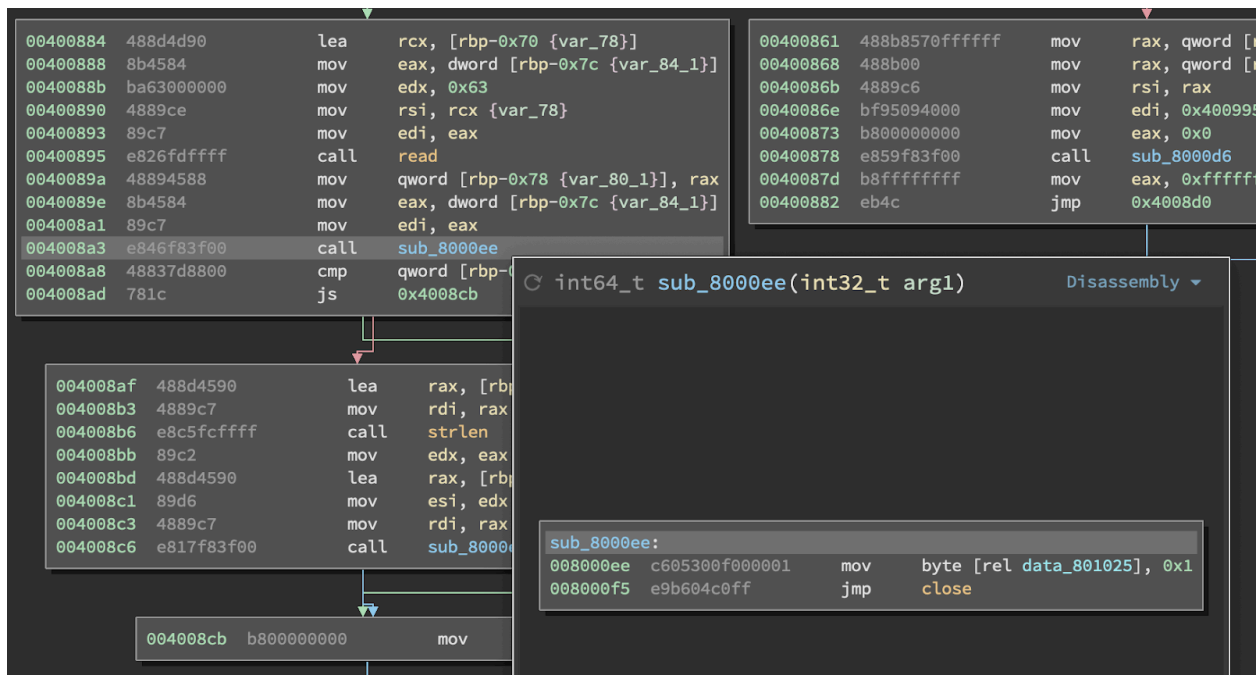


Figure 4.3: After probe placement via function hooking (no overhead)

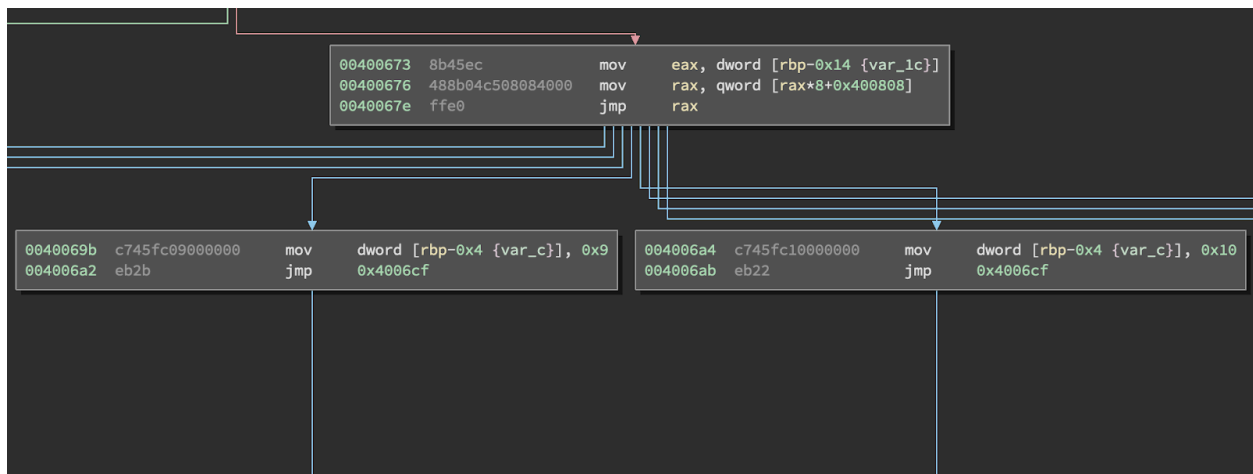


Figure 4.4: Before probe placement by replacing jump table destinations (no overhead)

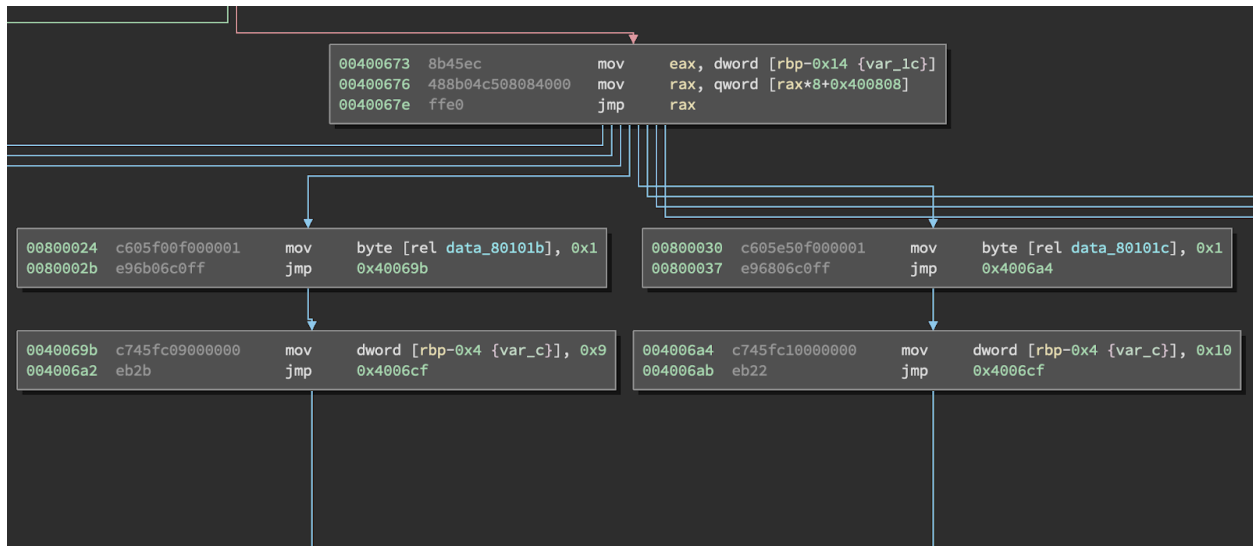


Figure 4.5: After probe placement by replacing jump table destinations (no overhead)

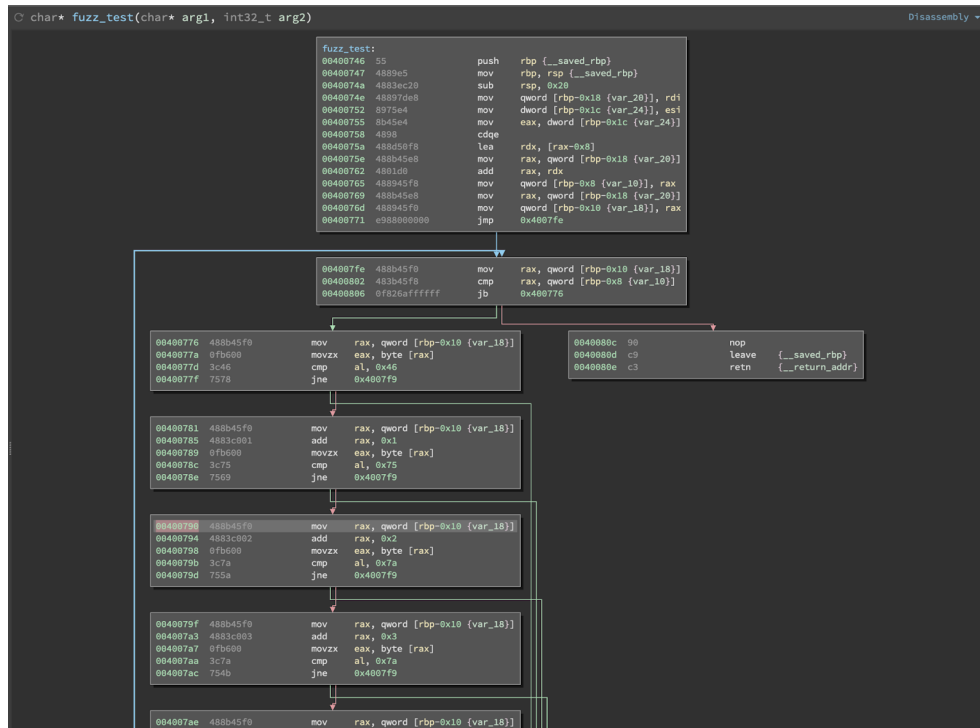


Figure 4.6: Before inline detours (more intrusive)

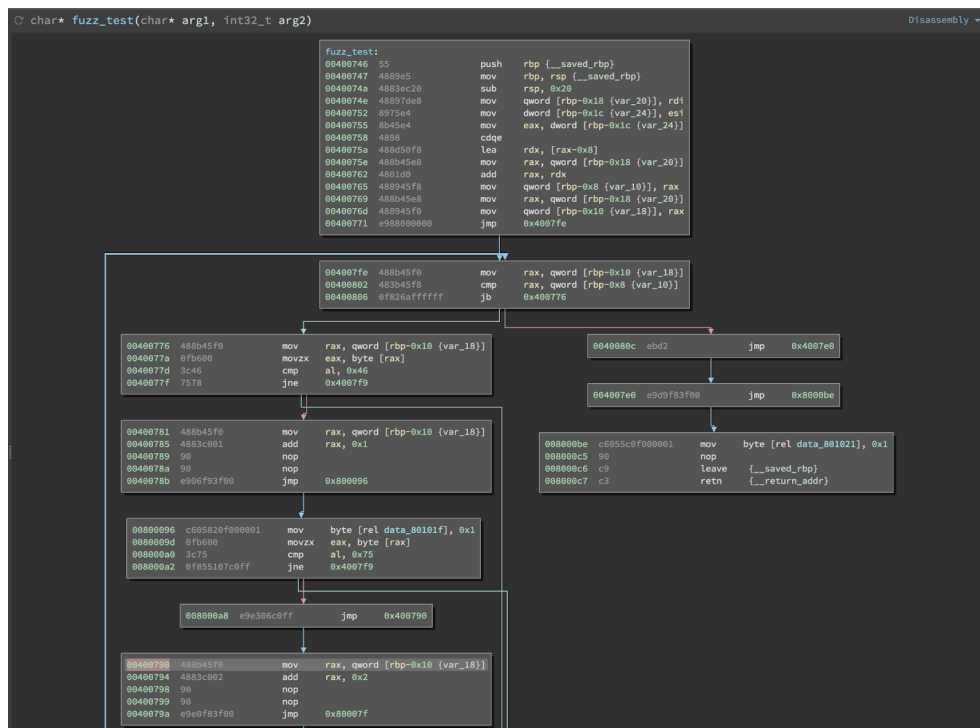


Figure 4.7: After inline detours (more intrusive)

During the patching process, bcov adds both a code segment for trampolines and a data section containing space for the coverage data. Each probe location is represented by a single byte in memory; if the associated block is covered during execution, the value is set to 1 (in the figures above: the added code and data addresses begin with 0x008 instead of the normal 0x004). While this approach allows easy compression and merging of coverage information, it requires using a library that can either be linked against or injected at runtime via LD_PRELOAD.

For general CFG reconstruction, the recovery of indirect control transfer targets is generally undecidable. However, bcov offers an approach that approximates the problem for jump tables with reasonable performance and accuracy. The proposed solution of “sliced microexecution” includes backwards slicing to identify the index variable followed by emulation of the given program slice. The implementation first attempts to test hypotheses that determine whether the approach is applicable, such as identifying a constant base address and if the jump table is constrained by bound conditions: either with direct index checks or arithmetic operations that constrain the index values. If the slice can be reduced to a code fragment where the index is the only changing variable, then the fragment can be emulated with different index values to determine jump targets as well as verify the bound condition.

The authors evaluated their implementation for speed, transparency, and accuracy against 8 different open source programs/libraries, using two different versions of clang and gcc and multiple compiler optimization and build settings. The resulting dataset included 95 binaries with some having sizes >20MB, and they report finishing instrumenting the largest binary in about 30 seconds, suggesting that the approach scales reasonably for real-world targets. For runtime performance they compared their implementation to the dynamic binary instrumentation (DBI) frameworks DynamoRIO and Pin, finding their implementation to be many times faster (14% overhead vs 7x and 29x), which is to be expected given the optimization of bcov as compared to the broader capabilities of a DBI framework. To test transparency, the authors ran the full test suites for each of the instrumented targets and found no regressions. Finally the authors tested the coverage accuracy against DynamoRIO and found very similar results, with bcov showing precision and recall of 99.97% and 99.95% respectively.

Use Cases and Limitations

Bcov’s design is explicitly focused on enabling the collection of basic block coverage information on binaries without requiring recompilation. The obvious use cases are for examining coverage of existing tests and to enable feedback for coverage-guided fuzzers. For these purposes, bcov performs well but does have some limitations.

First, bcov requires “well-formed” targets in order to perform function and CFG recovery. The authors mention that linker symbols and call-frame information can be a source of well-formed function definitions, but these are not guaranteed to be present. Further, they use a definition of “well-formed functions” that could be violated by obfuscated or malicious code. These limitations are at least easily identified early in the process, as bcov will fail to produce an instrumented binary, though the user feedback in such a case is not

descriptive.

In addition, bcov uses a runtime library to dump coverage information injected via LD_PRELOAD in the static rewriting case, which offers some advantages as well as limitations. The runtime library is designed to allow dumping of coverage upon receipt of a user signal, which enables “online” coverage collection. However, since the runtime library is injected into the target process, it must execute before the process exits. This means that crashes due to a SIGSEGV or other exceptional conditions, the runtime library must be able to wrest back execution or else no coverage will be recorded. So currently bcov cannot record coverage information during crashing behavior. Because crashes are a part of the fuzzing process, this could be a limitation.

Overall, the approach is sound, and it appears effective for its intended use case of block coverage collection. While it does not provide hit-counted edge coverage that many modern fuzzers use, block coverage analysis is still a useful tool and bcov is optimized for this use case.

4.2 ColdSnap

Reference Link	https://github.com/defparam/Coldsnap
Target Type	Linux Application Binaries
Host Operating System	Linux, MacOS
Target Operating System	Linux, MacOS
Host Architecture	x86 (32, 64), ARM, PPC, MIPS
Target Architecture	x86 (32, 64), ARM, PPC, MIPS
Initial Release	12/20
License Type	Open Source (MIT)
Maintenance	Last Commit, December 2020

Overview

Coldsnap is a lightweight, simple, snapshot fuzzing test harness that's implemented in Python. Coldsnap uses ptrace to:

- Gain introspection into the target "artifact under test" (AUT)
- Enable snapshot creation and management
- Generate code coverage information

Snapshot fuzzing is a technique used by test harnesses to optimize fuzz testing performance. On the back end, this test harness can be integrated with different mutational fuzzers to generate a large corpus of test data allied with code coverage information.

First researchers identify “snapshot save” and “snapshot restore” execution points in the target artifact under test (AUT) via static analysis/reverse engineering. These two execution points define the specific area of interest in the target AUT. The test harness is then configured with this information and AUT test execution begins. When the “snapshot save” point is reached, the test harness takes a snapshot of the process state of the AUT and saves it off. The AUT is then allowed to continue execution and process the fuzzed data. Once the target AUT has reached the “snapshot restore” point, the results of the computation are saved. This includes meta-data related to code coverage, memory taint analysis, fuzzed input corpus, and crash dumps. The test harness then restores the saved AUT process snapshot, mutates the input data further and starts the test again. By defining the area of interest in an AUT and re-executing those instructions repeatedly, significant efficiency gains are obtained.

Design and Implementation

The GitHub repo for ColdSnap includes two source code files: `target.c` and `coldsnap.py`.

The ‘C’ source file, “`target.c`” is a sample software (AUT) that has two intentionally crafted defects. The AUT parses command line input and, if the input matches a certain sequence or characters, the AUT crashes with a segmentation fault. The main execution loop has a function “`startf()`”, that marks the `snapshot_save` point. The `snapshot_restore` point is represented by a “`endf()`” function at the end of the execution loop. Between the two functions, is a call to “`process()`”—this is where the AUT does its processing. This source file is compiled to “`target`”, which is loaded and fuzzed by ColdSnap. When source code is unavailable, the snapshot save and restore points and code/symbol locations of interest have to be identified using reverse engineering.

`coldsnap.py` is a python-based implementation of the snapshot fuzzer. It has been developed as a proof of concept to demonstrate the use of a lightweight, `ptrace`-based framework. Because Coldsnap is only intended as a proof of concept, this article will focus on the core functionality rather than implementation details.

When ColdSnap is run with the name of the AUT, it creates a child process for the AUT, using `python.ptrace()`, and attaches to it. The `ptrace` module is used to set/delete breakpoints and read/write registers and memory in the process space. Once the AUT is loaded by the fuzzer, it enumerates the various memory segments that have been mapped into the AUT process space, by reading the `/proc/<pid>/maps` directory. The text/code segment is identified, and the Unix “`nm`” utility locates the symbol addresses of “`startf()`” and “`endf()`”. Next, the data segment is identified along with the payload’s address. Breakpoints are created at these locations using `ptrace` so that when execution breaks, we can save the initial snapshot and subsequently restore this saved snapshot. Execution is then started until the breakpoint at “`startf()`” is hit. Additional breakpoints are created for deriving code coverage data. In the example AUT, these breakpoints are created at every instruction – however, when fuzzing complex applications with many code paths, the code coverage breakpoints are created at the beginning of basic blocks of interest, as identified during the RE process. At this point, a snapshot of the process is created and stored away.

After setup, ColdSnap is ready to mutate the payload and feed it into the AUT. The tool first creates a corpus of fuzzing seeds guided by code coverage. Initially, this corpus is empty and the initial payloads are mutated based on whatever has been defined in the AUT. If a certain payload mutation encounters a coverage breakpoint, it's added to the corpus of fuzzer seeds. Each fuzzing run involves picking a random fuzzing seed from the corpus, mutating it further and feeding it to the AUT. The AUT is run until:

- A crash occurs and is trapped by ptrace. When this happens, the responsible fuzzer seed is noted and added to the corpus, along with crash location information.
- A coverage breakpoint is encountered. In this case, the fuzzer seed is added to the corpus along with code location data, the breakpoint is deleted, and execution continues.
- Execution reaches the "snapshot restore" breakpoint. If there are remaining code paths, (i.e., we still have some coverage breakpoints that haven't been encountered) restore the snapshot and restart the fuzzing loop. Otherwise, the AUT terminates.

Use Cases and Limitations

As mentioned previously, Coldsnap uses ptrace to introspect and control execution of the target AUT. In turn, it can only be used on Unix-like platforms to fuzz application-layer target artifacts. Coldsnap is best viewed as a prototype used to demonstrate the viability of lightweight and bespoke test harnesses.

Generally, code-coverage guided snapshot fuzzers have high performance. Coldsnap is written in Python, which makes it slower than a purely native implementation.

4.3 Format Fuzzer

Reference Link	https://github.com/uds-se/FormatFuzzer
Target Type	Binary
Host Operating System	Linux; MacOS
Target Operating System	Linux; MacOS; BSD; etc.
Host Architecture	N/A
Target Architecture	N/A
Initial Release	10/2021
License Type	Open Source (MIT)
Maintenance	Maintained by UDS-SE (Saarland University, Germany)

Overview

Fuzzers generate large amounts of invalid inputs that lead to uninteresting results. This is particularly true when fuzzing a program that expects highly structured input, such as an image binary or network packet. This wastes time and computer resources. FormatFuzzer uses templating to mitigate this problem and create high-performance fuzzers targeted at specific formats.

Design and Implementation

FormatFuzzer carries out structured fuzzing through the use of binary template files. Given a template specifying a data format, it will generate an executable generator. These generators create fuzzer inputs that can perform much better, for format-specific binaries, than

format-agnostic/mutation-based fuzzers. While FormatFuzzer can operate independently from the binary being tested, it can also integrate with AFL++ to add format awareness and potentially increase coverage during fuzzing.

FormatFuzzer has two other functions: a standalone parser, and a generator/mutator pair for specific binary formats.

The parser can be used to mutate inputs or test template accuracy. It is possible to save the choices the parser makes into a decision file. These decision files can then be modified to further increase mutation during binary format generation. For example, parsing a gif into a decision file, then using that decision file to generate a gif, will result in creating a copy of the original gif binary. By simply mutating that decision file, you can generate new files in valid gif format. This can increase efficiency compared to a fuzzer that mutates the entire binary structure, which can lead to inputs that are too malformed to lead to new coverage.

Templates allow even quicker fuzzing/parsing for specific formats. FormatFuzzer's binary templates are generated from the 010 editor [15]. FormatFuzzer will parse files into a hierarchical structure. Users may also create templates manually if the file format doesn't exist. These templates are in the format of C structures. Below is an example excerpt of a PNG binary template.

```
1 typedef struct {
2     byte btRed;
3     byte btGreen;
4     byte btBlue;
5 } PNG_PALETTE_PIXEL;
6
7 struct PNG_CHUNK_PLTE (int32 chunkLen) {
8     PNG_PALETTE_PIXEL plteChunkData[chunkLen/3];
9 };
```

Figure 4.8: PNG binary template (extract)

Use Cases and Limitations

The computing cost to generate formatted binaries during fuzzing is very low and can decrease the total time needed to fuzz while increasing code coverage. [16] Unfortunately, not all the 010 Editor's formats are included with the FormatFuzzer release. From the github's readme: "Out of the box, FormatFuzzer produces formats such as AVI, BMP, GIF, JPG, MIDI, MP3, MP4, PCAP, PNG, WAV, and ZIP; and we keep on extending this list every week[17] ." Another limitation is that while the existing binary templates work well for parsing, the binary generation requires further refinement. While generally easy to do, this increases overhead. From the article, novice engineers took a few days to update most formats, but some complex formats took over a week [16]. Because of this, this tool is more effective when used for fuzzing a system that accepts a limited number of formatted binary inputs.

Another possible drawback of using format-aware fuzzers is that, by taking away the freedom to mutate certain parts of a binary, some bugs could be missed that might have been found by a format-agnostic fuzzer. The increased performance likely outweighs the risk. It can be further mitigated by careful design of the templates or by carrying out a separate format-agnostic fuzzing run in parallel, ideally with shared seeds.

4.4 FuzzBuilder

Reference Link	https://github.com/hksecurity/FuzzBuilder
Target Type	32-Bit C/C++ Linux Libraries
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (64)
Target Architecture	x86 (32)
Initial Release	12/2019
License Type	Open-Source (No licence specified)
Maintenance	Not currently maintained - last updated 2020

Overview

FuzzBuilder is a greybox fuzzing tool for 32-bit Linux C/C++ libraries. It will generate entry-point executables and input seeds for many well-known fuzzers using pre-existing unit tests.

Design and Implementation

FuzzBuilder's approach can be broken down into two main steps. First, the tool instruments unit tests to capture the data passing through API function parameters. This allows FuzzBuilder to generate high coverage seeds. The second step is to analyze these unit test functions to create high coverage API function sequences for executable generation. To accomplish this, FuzzBuilder employs LLVM-6.0 to create IR bitcode, which it will then operate on and build as necessary.

While most of this is automated, the base library API functions must be first specified in a configuration file. The configuration file includes a base function that demonstrates sequence patterns that are used by FuzzBuilder to generate a fuzzing executable. Users can also indicate function parameters and buffer sizes as well as specify the files that contain the unit tests along with an identifier prefix for the unit test functions (fig. 4.9). This matching string may contain the '*' wildcard if a postfix is needed depending on the naming conventions of the library [18].

```
{
    "targets" : [ ["XML_Parse", 2, 3] ],
    "tests" : [ "test_" ],
    "files" : [ "runtests.bc" ]
}
```

Figure 4.9: XML_Parse.conf, a simple configuration file specifying a single base function and unit test function prefix [19].

Depending on the testing framework, this can be simple for software that has easy-to-consume naming conventions. If this isn't the case, then each test function will need to be specified explicitly. The user will then compile LLVM bitcode for the target and run FuzzBuilder's seed command to analyze and instrument the targeted unit tests (figures 4.10 and 4.11).

```
root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat# afl-clang -emit-llvm
-DHAVE_CONFIG_H -I. -I.. -DHAVE_EXPAT_CONFIG_H -m32 -g -Wall -Wmissing-prototypes -Wstrict-prototypes -fexceptions -fno-strict-aliasing -c lib/xmlparse.c -fPIC -DPIC
afl-cc 2.57b by <lcamtuf@google.com>
root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat# $FuzzBuilder/build/fuzzbuilder seed $FuzzBuilder/exp/expat/seed.conf
[D] == Config ==
[D] TARGET #1 XML_Parse 23
[D] FILE #1 xmlparse.bc
[D] ==
[D] == Loader ==
[D] Loaded Module #1 xmlparse.bc
[D] ==
[I] Identified Target Functions #1 XML_Parse
[I] Collect Instrumented at XML_Parse
[I] xmlparse.bc was modified to xmlparse.bc.mod.bc
root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat#
```

Figure 4.10: Emitting the LLVM IR for xmlparse.c and using FuzzBuilder's seed command to analyze emitted bitcode for seed generation.

```

root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat# afl-clang -DHAVE_CONFIG_H
-I. -I.. -DHAVE_EXPAT_CONFIG_H -m32 -g -Wall -Wmissing-prototypes -Wstrict-prototypes -f
exceptions -fno-strict-aliasing -c xmlparse.bc.mod.bc -fPIC -DPIC -o xmlparse.o
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 3338 locations (32-bit, non-hardened mode, ratio 100%).

```

Figure 4.11: Instrumenting and building unit test bitcode for seed collection.

Once the unit tests are instrumented, they must be run again to collect input data, which will then be used as fuzzing seeds (figures 4.12 and 4.13).

```

make check-TESTS
make[3]: Entering directory '/FuzzBuilder/exp/expat/source/libexpat/expat/tests'
make[4]: Entering directory '/FuzzBuilder/exp/expat/source/libexpat/expat/tests'
PASS: runtests
PASS: runtestsp
=====
Testsuite summary for expat 2.2.6
=====
# TOTAL: 2
# PASS: 2
# SKIP: 0
# XFAIL: 0
# FAIL: 0
# XPASS: 0
# ERROR: 0
=====

```

Figure 4.12: Rerunning instrumented unit tests to collect seed data.

```

root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat# python $FuzzBuilder/scrip
t/seed_maker.py fuzzbuilder.collect seed_fb
[0/17]
[1/17]
[2/17]
[3/17]
[4/17]
[5/17]
[6/17]
[7/17]
[8/17]
[9/17]
[10/17]
[11/17]
[12/17]
[13/17]
[14/17]
[15/17]
[16/17]
root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat#

```

Figure 4.13: Collecting the seeds from the output of unit test instrumentation.

Finally, the executables can be generated. First, FuzzBuilder will extract entry and test functions from the LLVM bitcode. It will then insert an interface for loading fuzzing input into

memory, along with the necessary instructions to support that interface. Next, it will iterate across all the test functions and check to see if they are necessary. If so, FuzzBuilder will modify them to accept the loaded fuzzing input; otherwise it will remove them (figures 4.14 and 4.15). [18]

```

1: procedure FUZZBUILDER(functions)
2:   tests, entry  $\leftarrow$  preprocess(functions)
3:   entry  $\leftarrow$  insert_interface(entry)
4:   for all test  $\in$  tests do
5:     if is_necessary(test) then
6:       test  $\leftarrow$  insert_operands(test)
7:     else
8:       test  $\leftarrow$  remove_test(test)
9:     end if
10:  end for
11:  modify(entry, tests)

```

Figure 4.14: The process FuzzBuilder uses to generate an executable from unit tests [18].

```

root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat/tests# afl-clang -emit-llvm -DHAVE_CONFIG_H -I. -I.. -I../lib -DHAVE_EXPAT_CONFIG_H -m32 -g -Wall -Wmissing-prototypes -Wstrict-prototypes -fexceptions -fno-strict-aliasing -c runtests.c
afl-cc 2.57b by <lcmtuf@google.com>
root@72117f2a0b03:/FuzzBuilder/exp/expat/source/libexpat/expat/tests# $FuzzBuilder/build/fuzzbuilder exec $FuzzBuilder/exp/expat/XML_Parse.conf
[D] == Config ==
[D] TARGET #1 XML_Parse 23
[D] TEST #1 test_
[D] FILE #1 runtests.bc
[D] ==
[D] == Loader ==
[D] Loaded Module #1 runtests.bc
[D] ==
[D] Test Function #1 test_wfc undeclared_entity_with_external_subset_standalone
[D] Test Function #2 test_entity_with_external_subset_unless_standalone
[D] Test Function #3 test_wfc_no_recursive_entity_refs
[D] Test Function #4 test_ext_entity_set_encoding
[D] Test Function #5 test_alloc_nested_groups

```

Figure 4.15: Generating executables for library functions based on unit tests.

Finally, the generated executables may be run with most fuzzers. AFL is used in the example below (fig. 4.16).

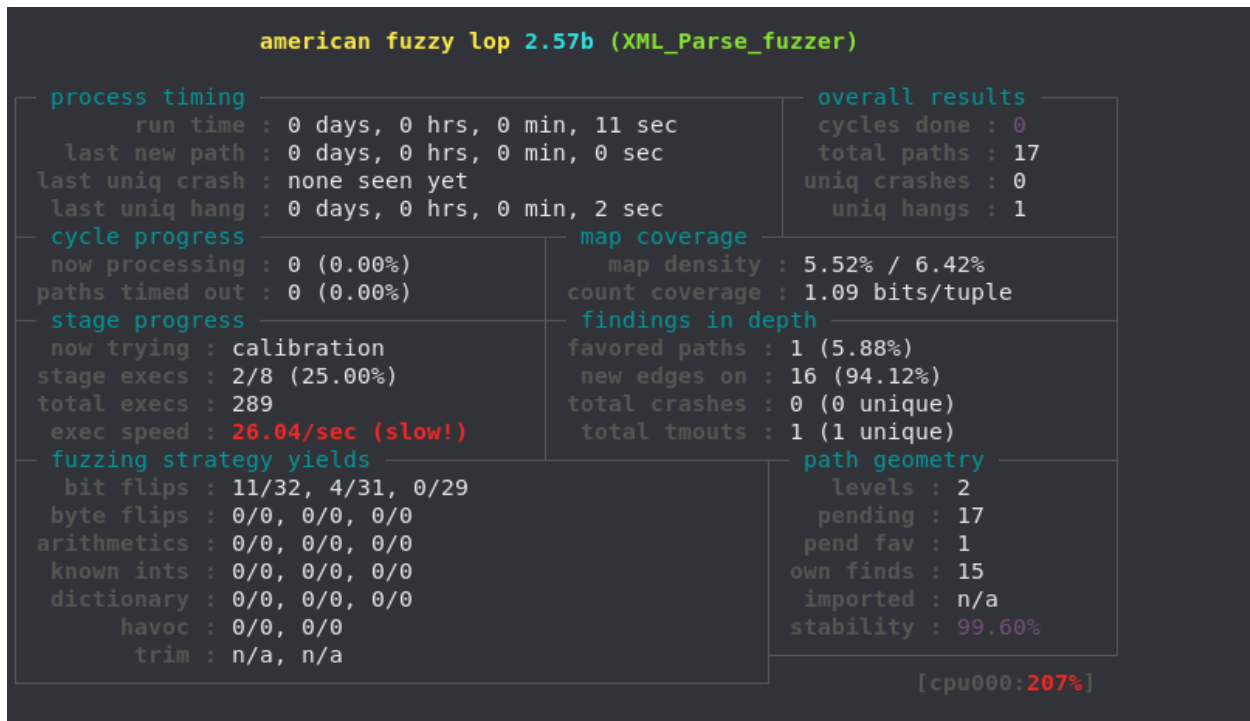


Figure 4.16: Fuzzing the generated executables.

Use Cases and Limitations

FuzzBuilder has a large number of shortcomings. First, the environment is difficult to set up, so users may have better luck using the provided Docker container. Second, the target source must provide unit tests for the target library. These unit tests should preferably use a popular testing framework like Google test with conventional function names. Otherwise, these test functions must be specified in a configuration file, which can be cumbersome considering the size of real-world libraries. In some cases, the base function for a unit test may not even match the format necessary for FuzzBuilder to analyze it. This is the case with libpng, where data is provided to the unit test via a file opened within the test function [20]. Third, additional configuration files are also necessary for providing information about target function prototypes (that is, function names and parameters.) Last, and most limiting, is that FuzzBuilder will only generate executables for 32-bit libraries.

The README for the project only supplies enough information to reproduce the experimental environment presented in the paper [19]. Specifics on necessary details, such as afl-clang compilation flags, are not provided. Although the concept for this tool is very interesting, the current version is too limited to be of broad use.

4.5 Fuzzolic

Reference Link	https://github.com/season-lab/fuzzolic
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86_64
Target Architecture	x86_64
Initial Release	27 Dec 2019
License Type	Open Source (GPL-2.0)
Maintenance	2 contributors (Sapienza University of Rome), last commit 4 Nov 2021

Overview

Fuzzolic is a concolic framework designed to support coverage-guided fuzzing [21][22]. It is built on top of QEMU. A component of Fuzzolic is an approximate solver named Fuzzy-SAT that provides an alternative to traditional SMT solvers. The work is comparable to QSYM and SymQEMU in that it:

1. Uses concolic execution under emulation or dynamic binary instrumentation.
2. Provides new improvements in optimizing symbolic execution.
3. Is designed for use in conjunction with a fuzzer such as AFL++.

Design and Implementation

Fuzzolic includes several main components: a QEMU-based tracer, a solver frontend, and a solver backend (which can be Z3 or the authors' mutation-based solver, Fuzzy-SAT). The framework is designed to trace an input as the target program executes, and produce new inputs by tracking symbolic state. The components are designed such that they may be run standalone: Fuzzy-SAT can be invoked to solve SMT queries; Fuzzolic can be used to produce new inputs from a single seed and target; or the system can be run as a worker node in a hybrid fuzzing configuration alongside AFL++ (figure 4.17).

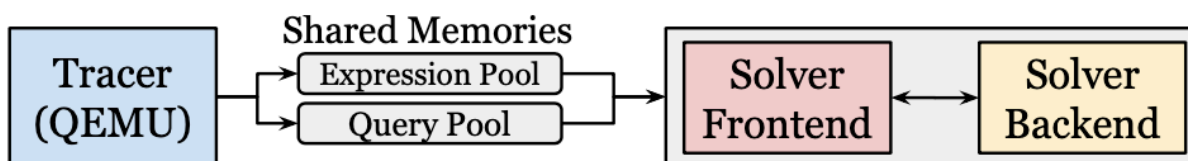


Figure 2: Internal architecture of FUZZOLIC: a QEMU-based component executes the binary application, building symbolic expressions and queries within two shared memories (*Expression Pool* and *Query Pool*, respectively), while a solver frontend reads from the two shared memories, optimizing the expressions and submitting the queries to the solver backend.

Figure 4.17: Fuzzolic's architecture as documented in the Fuzzolic paper [21]

Fuzzolic's QEMU-based component is responsible for emulating the target and recording symbolic state. This approach does not require source code, nor is it tied to a particular architecture. The authors also leverage JIT compilation and basic blocks caching to increase the efficiency of their tracer. This is further augmented by implementing three different analysis modes (figure 4.18) depending on the state of execution and symbolic input. This allows Fuzzolic to cache blocks in either a concrete or symbolic cache. Fuzzolic can also skip the production of branch queries within standard library functions, such as malloc.

	ANALYSIS MODE		
	A	B	C
Expr. builder	✗	✓	✓
Branch queries	✗	✗	✓
Overhead	low	medium	high
JIT cache	concrete	symbolic	symbolic
Use scenario	program startup and API models	uninteresting code	interesting code

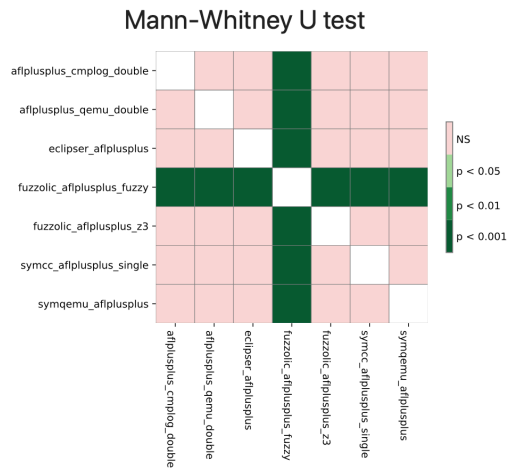
Figure 4.18: Fuzzolic's three different analysis modes

The tracing and solving components are run in separate processes. The tracer generates symbolic expressions and queries during execution of the target program, and the solving component focuses on receiving, optimizing, and solving the queries. The solving component is designed to be able to swap backends, and the authors enable the use of both Z3 and their own solver, Fuzzy-SAT.

Fuzzy-SAT is designed to mirror the interface of an SMT solver while functioning in a completely different manner. Fuzzy-SAT uses an analysis process to observe how the input is used and generate intelligent mutations in an attempt to solve the query at hand. The analysis phase observes the execution and collected expressions to determine groups of input bytes and attempts to develop knowledge of how the bytes are used. Based on the input groups identified, the reasoning phase attempts to exploit these relationships to generate intelligent mutations. For example, if an expression limits an input group to a given range, Fuzzy-SAT can attempt to brute-force values within that range interval.

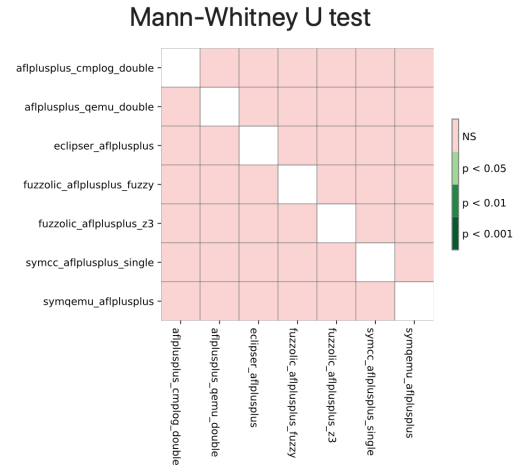
By leveraging the identification of input groups and multiple mutation strategies, Fuzzy-SAT analyzes both what bytes are used and how they are used. This approach has similar characteristics to the research projects Angora, Eclipser, and JFS, but its integration into a framework like Fuzzolic is unique. In practice this yields a solver that behaves like Z3 but with very different performance characteristics.

In practice, Fuzzolic is an effective VR tool. Experimental results in the authors' original paper, as well as independent metrics from Google's FuzzBench demonstrate high performance. The paper's evaluation tested four different fuzzers (Fuzzolic with Fuzzy-SAT, AFL++, Eclipse, and QSYM) across 8 benchmark targets, and showed that Fuzzolic generated more code coverage in 6 out of 8 targets. Google's FuzzBench compares the results of multiple fuzzers across different target benchmarks. Fuzzolic showed comparable performance or significant improvement over other fuzzers for a number of targets. This is illustrated in figures 4.19 and 4.20.



The table summarizes the p values of pairwise Mann-Whitney U tests. Green cells indicate that the reached coverage distribution of a given fuzzer pair is significantly different.

Figure 4.19: Mann-Whitney U test results for libjpeg_turbo



The table summarizes the p values of pairwise Mann-Whitney U tests. Green cells indicate that the reached coverage distribution of a given fuzzer pair is significantly different.

Figure 4.20: Mann-Whitney U test results for openssl_x509

Use Cases and Limitations

Fuzzzolic is distributed both as a docker container and source and provides an example script for common use cases. While some users might want to test Fuzzy-SAT on it's own, the most common use case is in combination with a fuzzer. The primary wrapper script provides an "AFL mode" for this, which launches two AFL++ instances to generate inputs via fuzzing (figure 4.21). The Fuzzzolic script reads from the input queue of the one instance, generates new inputs, and writes those new inputs into the queue of the other instance. This ensures these new inputs will eventually imported by AFL++ for mutation.

```

ubuntu@b7821b54796a:~/fuzzzolic$ ./fuzzzolic/run_afl_fuzzzolic.py \
> --address-reasoning --optimistic-solving --fuzzy \
> --timeout 90000 -o workdir-fuzzy \
> -i /host/example/input/ -- \
> /host/example/bin-plain/stdin_example
Running: /home/ubuntu/fuzzzolic/utils/AFLplusplus/afl-fuzz -c 0 -M afl-master -o workdir-fuzzy -i /host/example
/input/ -Q -- /host/example/bin-plain/stdin_example
Waiting 30 seconds before starting slave.
Running: /home/ubuntu/fuzzzolic/utils/AFLplusplus/afl-fuzz -S afl-slave -o workdir-fuzzy -i /host/example/input
/ -Q /host/example/bin-plain/stdin_example
Waiting 30 seconds before starting fuzzzolic.
Running: /home/ubuntu/fuzzzolic/fuzzzolic/fuzzzolic.py -f -p -r -t 90000 -a workdir-fuzzy/afl-slave/ -i workdir-f
uzzy/afl-slave/queue/ -o workdir-fuzzy/fuzzzolic -- /host/example/bin-plain/stdin_example
Configuration file for /host/example/bin-plain/stdin_example is missing. Using default configuration.

Running directory: /home/ubuntu/fuzzzolic/workdir-fuzzy/fuzzzolic/fuzzzolic-00000
Using Fuzzy-SAT solver
[+] Keeping test_case_0_0.dat
Run took 2.7 secs

Running directory: /home/ubuntu/fuzzzolic/workdir-fuzzy/fuzzzolic/fuzzzolic-00001
Using Fuzzy-SAT solver
[+] Keeping test_case_40_0.dat
[-] Discarding test_case_43_666.dat
[+] Keeping test_case_2_0.dat
[+] Keeping test_case_16_0.dat
[+] Keeping test_case_56_666.dat
[-] Discarding test_case_44_666.dat
[+] Keeping test_case_17_0.dat
[+] Keeping test_case_1_0.dat
[+] Keeping test_case_48_666.dat
[-] Discarding test_case_14_0.dat
[-] Discarding test_case_52_0.dat
[-] Discarding test_case_50_666.dat

```

Figure 4.21: Invoking Fuzzzolic wrapper in AFL mode.

The authors of Fuzzzolic have put some effort into the usability of the tool. A Dockerhub container distribution and the wrapper script make it relatively easy to get started. The most recent version of the code on GitHub handles both targets that take input from stdin and from a file and works as advertised on uninstrumented targets. While the error messages are sometimes challenging to understand, the debugging flags were sufficient to troubleshoot basic problems.

4.6 Go-Fuzz

Reference Link	https://go.dev/proposal/+master/design/draft-fuzzing.md https://github.com/golang/go
Target Type	Binary
Host Operating System	Linux, Windows, MacOS
Target Operating System	AIX, Android, DragonFly BSD, FreeBSD, Illumos, Linux, macOS/iOS (Darwin), NetBSD, OpenBSD, Plan 9, Solaris, and Windows
Host Architecture	x86_64
Target Architecture	x86_64, i386, ARM, ARM64, mips64, mips64le, mips, mipsle, ppc64, ppc64le, riscv64, s390x, wasm
Initial Release	Forthcoming stable release Q1 2022; beta release evaluated
License Type	Open Source (LGPL-3.0)
Maintenance	Regular commits and releases with a large developer community (approximately 1,700 contributors).

Overview

Golang (Go), is a procedural programming language initially developed at Google for writing microservices and released publicly in 2009 [23]. Due to its execution speed, cross-compilation support, and feature-rich standard library it has become a popular choice for software developers targeting other applications, including security researchers. The Go

development community recognized the importance of fuzzing to the early identification of bugs and security of systems written in the language, and several independent projects emerged to add source code fuzzing capabilities [24] [25]. However, these projects add additional friction to the workflow for writing and running fuzz test cases, lack some critical features (e.g., code that uses Go C language bindings) and require project-dependent packages to use, so usage and quality varies from system to system. The Go development team has sought to address this by bringing first-class support for fuzz testing into the standard library and tool chain [26].

Design and Implementation

Go already provides built-in support for writing unit test cases that can be automated with no extra tooling or dependencies. [27]. In order to make fuzzing as convenient to use as possible, all fuzzing data types and methods are included as part of this standard library package and therefore can be included as new methods in existing unit test cases. The build toolchain supports the “go test” CLI command that will, in addition to building all the code in the project, build and run all files ending in “_test.go”. By the language specification, unit tests are included in these files and should be a one-to-one mapping of program code to test code (e.g. foo.go, foo_test.go), as appropriate for the project. When built using the “-fuzz” flag, the target executable contains fuzzing coverage instrumentation that is used by the engine to inject mutated inputs.

Fuzz tests rely on a seed corpus of inputs used by the coverage-guided fuzzing engine. The corpus is optional, but the engine will more efficiently find errors when seed inputs are provided to guide coverage [27]. Next, the mutator will use the seed corpus to begin randomly modifying input bytes, converting the result into language types required by the test. The generated corpus is created and maintained by the fuzzing engine and will grow as new inputs are found.

Figure 1 shows a partial listing of a test for the parsing of a new standard library function for an IP data type [28]. For the developer, the test is simple to write and no prior knowledge of fuzzing concepts is required. In this example the corpus is contained in the source code of the test, however it’s possible to define it in a directory on disk that will be searched by the fuzzing engine at run time.

The function `f.Fuzz(t *testing.T, s string)` is the actual test that will be invoked by the fuzzing engine (Go defines the type `testing.f` for use in creating and running fuzz tests, and it’s only valid for use inside `*_test.go` source). The type of arguments passed to `f.Add()` are identical to the argument(s) passed to `f.Fuzz()`. After defining the set of initial inputs in the corpus variable, the `FuzzParse()` function adds each to the seed corpus for the fuzzing engine (at `f.Add()`). The fuzzing engine will start by using each of the provided inputs in the corpus. Then, it will begin mutating those inputs and pass each mutated string as an argument to the `ParseAddr()` function. Inputs that result in a crash will be reported to the developer and automatically added to the file corpus on disk for inclusion in future test runs.

```
var corpus = []string{
```

```

// Basic zero IPv4 address.
"0.0.0.0",
// Basic non-zero IPv4 address.
"192.168.140.255",
// IPv4 address in windows-style "print all the digits" form.
"010.000.015.001",
// IPv4 address with a silly amount of leading zeros.
"000001.00000002.00000003.00000004",
// 4-in-6 with octet with leading zero
 "::ffff:1.2.03.4",
// Basic zero IPv6 address.
 "::",
// Localhost IPv6.
 "::1",
// Fully expanded IPv6 address.
"fd7a:115c:a1e0:ab12:4843:cd96:626b:430b",
// IPv6 with elided fields in the middle.
"fd7a:115c::626b:430b",
<...more test inputs...>
}

func FuzzParse(f *testing.F) {
    for _, seed := range corpus {
        f.Add(seed)
    }

    f.Fuzz(func(t *testing.T, s string) {
        ip, _ := ParseAddr(s)
        checkStringParseRoundTrip(t, ip, ParseAddr)
        checkEncoding(t, ip)

        // Check that we match the net's IP parser, modulo zones.
        if !strings.Contains(s, "%") {
            stdip := net.ParseIP(s)
            if !ip.IsValid() != (stdip == nil) {
                t.Logf("ip=%q stdip=%q", ip, stdip)
                t.Fatal("ParseAddr zero != net.ParseIP nil")
            }
        }
    })
}

```

Listing 4.1: Partial listing of fuzz test for `netip.ParseAddr()` function in the Go standard library [28]

The test is invoked via the Go toolchain:

```
go test -fuzz <regex matching the name of test(s)>
```

At the time of this evaluation, all code for the feature was fully merged into the development branch [29]. The next release of Go is 1.18, due in early 2022, will be the first stable release

to include support. In order to use the fuzzing framework, we used the latest stable version of Go (1.17.5) to build the development branch on a Debian 11 VM as follows:

```
curl -L https://go.dev/dl/go1.17.5.linux-amd64.tar.gz -O
tar -C /usr/local -xvf go1.17.5.linux-amd64.tar.gz
go install golang.org/dl/gotip@latest
gotip download
<building>
gotip version
go version devel go1.18-766f89b Fri Dec 10 19:26:50 2021 +0000 linux/amd64
```

Listing 4.2: Building the development branch

Use Cases and Limitations

The envisioned use case for this feature is for all systems written in Go to begin including fuzzing tests as part of the development workflow of the project. Similar to the use case for including unit test scaffolding in the Go toolchain, by including fuzzing capabilities the community expects to see a significant uptake in fuzz testing performed by developers, thereby increasing security and reliability of all Go applications. This is particularly important in Go packages that are imported by a large number of other projects. The Go development team will track the usage of the fuzzing capabilities in public repositories to measure adoption rates.

The mutation engine currently does not support “struct” types. The constituent field types in a struct must have corpus data and fuzzing code written for them individually [24]. A future release will add support for dynamically marshalling data into types in a struct. This version also does not support writing custom generators for the mutator, which is another feature identified for future inclusion.

Another challenge is that corpus data can grow very large, and therefore is not desirable to include in the git repository with the source code. Current recommendations are to only keep the corpus data locally on a developer or build machine, or to create a separate repository exclusively for corpus data for a given project in order to share among development team members.

4.7 Gramatron

Reference Link	https://github.com/hexhive/gramatron
Target Type	Source (C/C++/Objective C) QEMU Mode: Binary
Host Operating System	Linux; BSD With Constraints: macOS, Solaris (Any OS supported by AFL++)
Target Operating System	(Any OS supported by AFL++) Linux; BSD With Constraints: macOS, Solaris
Host Architecture	(Any architecture supported by AFL++) Primary: x86 (32, 64); With Modification: ARM (32, 64); PPC (32, 64); MIPS (32, 64); etc. Linux; MacOS
Target Architecture	x86(32, 64) QEMU Mode: QEMU Supported Architectures
Initial Release	07/2021
License Type	Open Source (Apache 2.0)
Maintenance	Maintained by Prashast Srivastava (Purdue University) and Mathias Payer (EPFL)

Overview

Gramatron is a grammar-based coverage-aware fuzzer built on top of AFL++. It was presented in "Gramatron: Effective Grammar-Aware Fuzzing" at ISSTA 21 in July 2021 [30]. The tool uses grammar automata instead of parse trees to traverse the input space; the authors of the paper claim that this allows Gramatron to more aggressively mutate existing inputs, reduce biases in the generated inputs, and reduce the input representation size

compared to other grammar-aware fuzzers. The results showed improvements over two variants of Nautilus [31], a similar tool. In the paper, the tool was used on three popular interpreters for a 10-day fuzzing campaign where it helped uncover 10 new vulnerabilities, including one named CVE [32]. The Gramatron source code is available on Github [33]. A pre-built Docker image is also available. Gramatron has also been incorporated into the current "3.15a" development branch of AFL++ [34].

Design and Implementation

Gramatron works in two stages. In the first "preprocessing" stage, Gramatron converts the input grammar (a context-free grammar) into a grammar automaton. In the second "fuzzing" stage, the grammar automaton is used to explore the input space for the fuzzing target. Mutation operations together with coverage-based feedback are used to modify a random automaton walk.

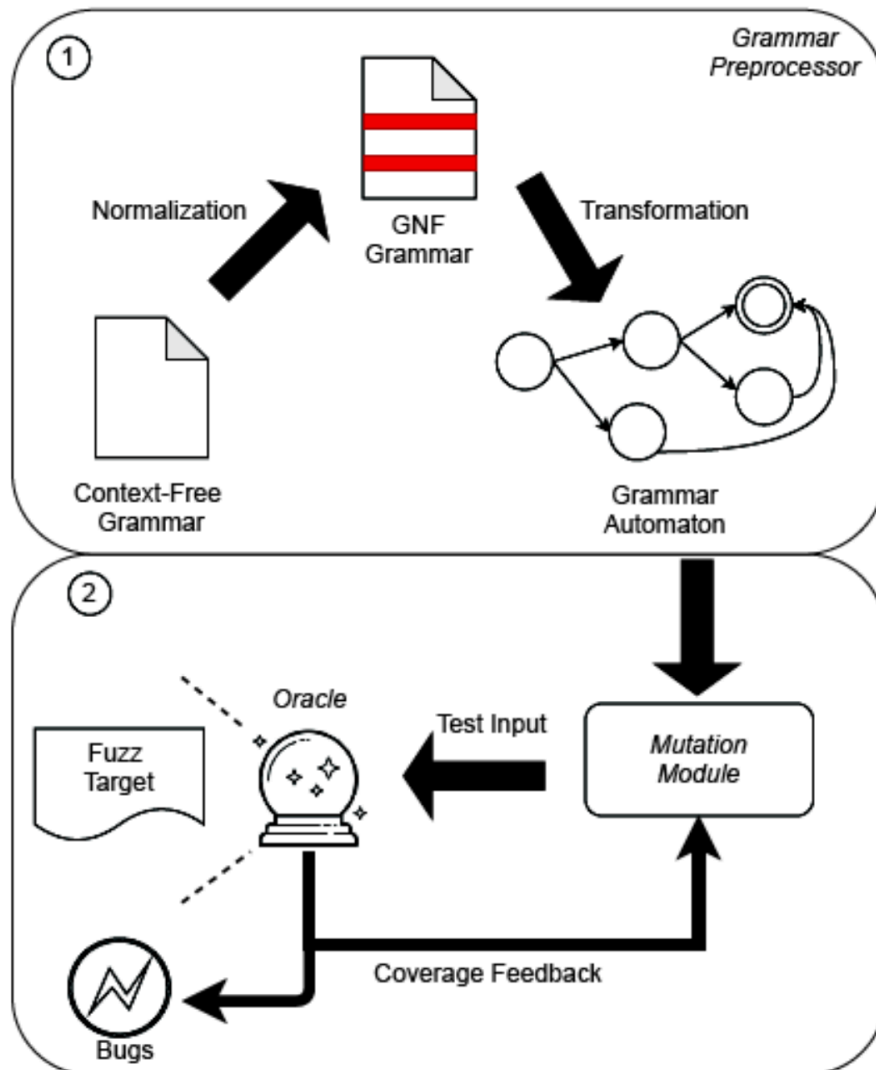


Figure 4.22: Overview of Gramatron

The grammar automaton is generated in two steps. First, the input grammar is converted into Greibach normal form (GNF). A context-free grammar is in GNF if its production rules have the form:

$$A \rightarrow t \ A_1 \ \dots \ A_k \ (*)$$

where t is a terminal symbol and the A 's are non-terminals; every context-free grammar can be converted into GNF [35].

program	→	'<?php' phpBlock '?>'
phpBlock	→	stmt
		stmt phpBlock
stmt	→	callStmt
		retStmt
callStmt	→	func '() ;'
retStmt	→	'return ;'
func	→	'rand '
		'mt_rand '

Figure 4.23: Subset of PHP grammar [30]

program	→	'<?php' phpBlock C
phpBlock	→	'rand ' A
		'mt_rand ' A
		'rand ' A phpBlock
		'mt_rand ' A phpBlock
		'return ;' phpBlock
		'return ;'
A	→	'() ;'
C	→	'?>'

Figure 4.24: Same grammar in GNF [30]

Next, the GNF grammar is converted to a finite state automaton (FSA). This is done by constructing the pushdown automaton corresponding to the grammar while using a stack limit so that there are only a finite number of states. The states correspond to stacks of non-terminal symbols (up to a configurable maximum depth), and each transition corresponds to popping the top element A on the stack and replacing it with A_1, \dots, A_k while emitting t , using a production rule (*) corresponding to the top element A .

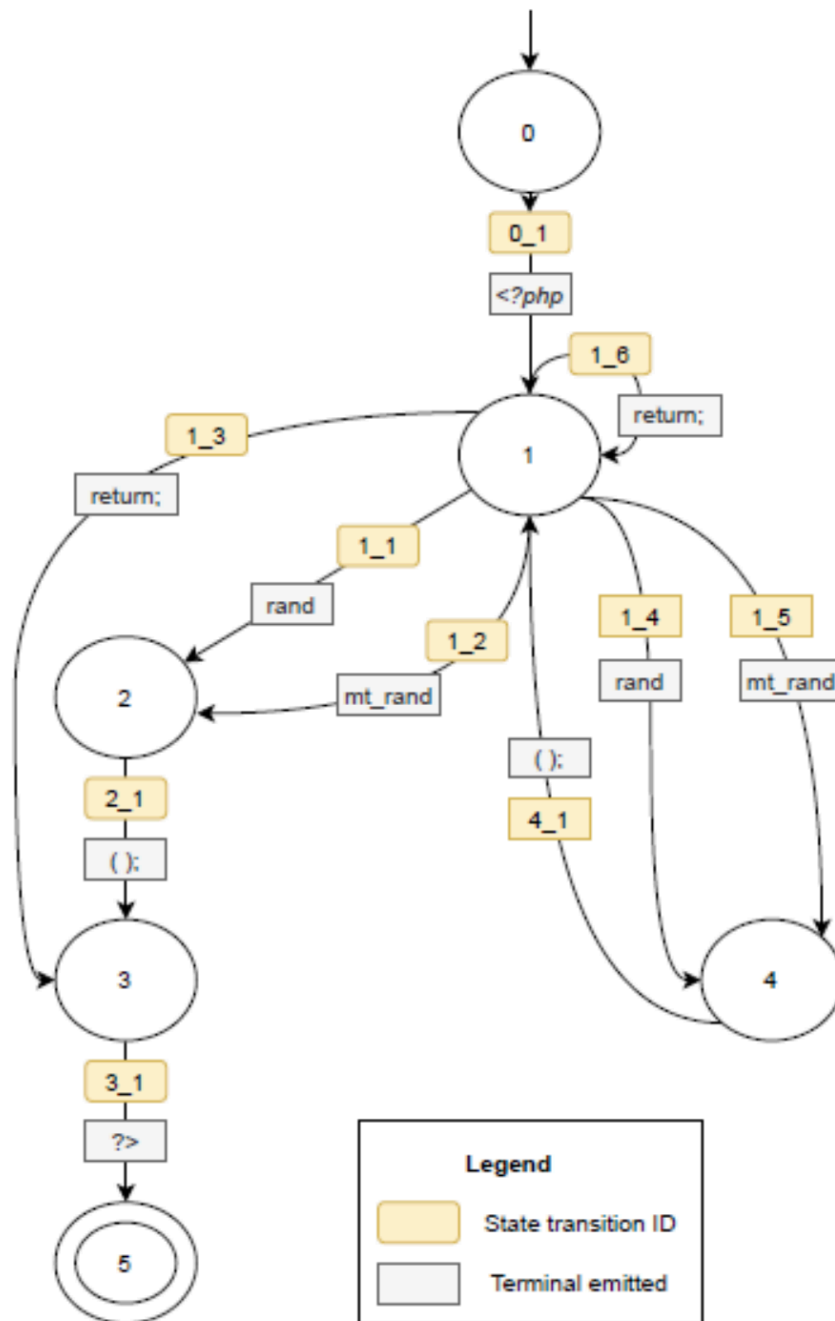


Figure 4.25: FSA for grammar in figure 4.24 [30]

In the fuzzing stage, Gramatron generates 100 seed inputs by performing random walks over the grammar automaton. It then repeatedly performs fuzz iterations, each of which consists of four steps:

1. Choose a seed from the queue.

2. Pass the seed through each of a set of mutation operators.
3. Test generated mutants on fuzz target.
4. Select candidates for further testing based on coverage feedback.

To avoid getting stuck, Gramatron generates a new candidate for each iteration. Gramatron uses three types of mutation operators:

- **Splicing** – “combining two inputs while preserving syntax.” This is done by taking two automaton walks *W1* and *W2*, picking a particular state *S* in *W1*, and replacing after *S* by a fitting subwalk from *W2* that starts at the same state *S*.
- **Random mutation** – “pick a random non-leaf non-terminal node and create a new subtree.” This is done by taking an automaton walk, stopping at an intermediate state *S*, and then performing a random walk starting from that point until a final state is reached.
- **Random recursive** – “find recursive production rules and unroll up to *n* times.” This is done by taking an automaton walk *W*, traversing it once to log recursive features, and then replicating those features (subwalks)[30].

Implementation The grammar preprocessor, which converts the grammar production rules into a grammar automaton, is implemented in Python. The input generator and mutator are implemented in C as a custom mutator for a modified version of AFL++. Our experience with using the Docker container is documented below. The docker container was run on a x86-64 Dell laptop host running Ubuntu 18.04.6 LTS. Some tweaks were necessary but overall it was easy to run the tool. Specifically, we started by running the sample test fuzzing run described in the accompanying README file. That involved building the target from source. Afterwards, we modified the procedure to verify that Gramatron still worked against a pre-compiled binary target by using AFL’s QEMU mode.

```
docker pull prashast94/gramatron:latest
// image hash: bfb280a0cef7711e5b901bfaaa50afe8c381fccb05510d93e7ffe9bf15839ff2
docker run --security-opt seccomp=unconfined -it prashast94/gramatron:latest /bin/bash
```

Next we prepare the target. For the sample campaign, we run “create sample script,” which downloads the target source code and compiles it using afl-clang-fast. At this point we also need to generate an automaton from an input grammar. For this run we used the pre-built automaton that comes with the Docker container. The input grammar file `source.json` (1195 lines, 25853 bytes) starts like this:

```
root@83472e836918:~/gramatron_src/gramfuzz-mutator# head ~/grammars/ruby/source.json
{
  "ARGS": [
    "VAR",
    "VAR ' ', ' ARGS",
    " ' ' "
```

```

],
"IDENTIFIER": [
    "'abcdef0123456789ABCDEF' ",
    "'abcdefghijklmnopqrstuvwxyz' ",
    "'abort' ",

```

while the automaton file (69263 bytes) is a packed JSON file which when pretty-printed starts off like this:

```

{
  "final_state": "6",
  "pda": {
    "15": [
      [
        "15_1",
        "1",
        "a"
      ],
      [
        "15_2",
        "1",
        "b"
      ],
    ],

```

Fuzzing the target We run the "run_campaign.sh" script to start the fuzzing run:

```

./run_campaign.sh ~/grammars/ruby/source_automata.json test_output "/tmp/mruby/bin/
mruby @@"=

```

This simple script first prepares an AFL++ input directory with 100 dummy "a" seeds. It then runs afl-fuzz, using the AFL_CUSTOM_MUTATOR_ONLY and AFL_CUSTOM_MUTATOR_LIBRARY to mutate inputs using the Gramatron custom mutator gramfuzz-mutator.so. At this point, afl-fuzz crashed with the following error:

```

[-] Hmm, your system is configured to send core dump notifications to an
external utility. This will cause issues: there will be an extended delay
between stumbling upon a crash and having this information relayed to the
fuzzer via the standard waitpid() API.
If you're just testing, set 'AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1'.

```

To avoid having crashes misinterpreted as timeouts, please log in as root and temporarily modify /proc/sys/kernel/core_pattern, like so:

```

echo core >/proc/sys/kernel/core_pattern

```

```

[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
    Location : check_crash_handling(), src/afl-fuzz-init.c:1701

```

```

It was necessary to change this kernel setting on the host:
user@ubu0731:~$ cat /proc/sys/kernel/core_pattern
|/usr/share/apport/apport %p %s %c %d %P %E
user@ubu0731:~$ sudo echo core >/proc/sys/kernel/core_pattern
user@ubu0731:~$ cat /proc/sys/kernel/core_pattern
core
user@ubu0731:~$

```

Trying again, we encountered another error:

```

[-] Whoops, your system uses on-demand CPU frequency scaling, adjusted
    between 781 and 4003 MHz. Unfortunately, the scaling algorithm in the
    kernel is imperfect and can miss the short-lived processes spawned by
    afl-fuzz. To keep things moving, run these commands as root:

```

```

cd /sys/devices/system/cpu
echo performance | tee cpu*/cpufreq/scaling_governor

```

You can later go back to the original state by replacing 'performance' with 'ondemand' or 'powersave'. If you don't want to change the settings, set AFL_SKIP_CPUFREQ to make afl-fuzz skip this check - but expect some performance drop.

```

[-] PROGRAM ABORT : Suboptimal CPU scaling governor
    Location : check_cpu_governor(), src/afl-fuzz-init.c:1810

```

Indeed we need to select a different CPU scaling governor:

```

user@ubu0731:~$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor | uniq
powersave
user@ubu0731:~$ sudo bash -c 'for x in `ls /sys/devices/system/cpu/cpu*/cpufreq/
    scaling_governor`; do cat $x; done'
user@ubu0731:~$

```

Moving on, we tried running the "run_campaign.sh" script and ran into a third and final error:

```

Read length:621[*] Spinning up the fork server...
[+] All right - fork server is up.
[*] Target map size: 15680

```



```
./run_campaign.sh: line 34: 1549 Illegal instruction (core dumped)
root@83472e836918:~/gramatron_src/gramfuzz-mutator#
```

It turned out that the afl-fuzz binary in the Docker container was compiled for a processor that supports an AVX512 extension missing from our host processor. Fortunately we were able to recompile afl-fuzz inside the container:

```
cd /root/gramatron_src/afl-gf
make clean
make
```

This time we the "run_campaign.sh" script ran successfully:

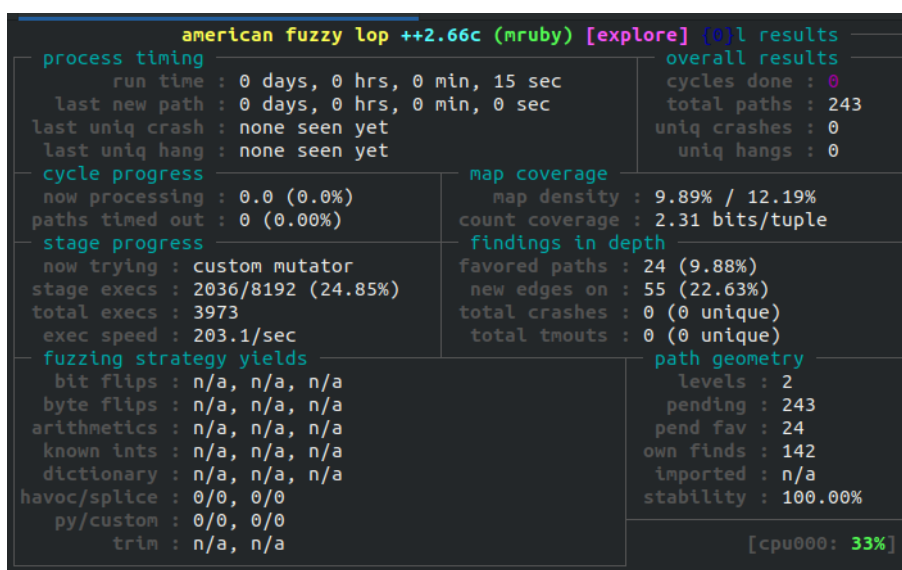


Figure 4.26: Successful AFL GUI

Meanwhile, the seed files in the input directory (/tmp/inputs) have been replaced with inputs generated with from the grammar:

```
root@83472e836918:/tmp/inputs# cat 002
a=false . object(d,c)
break d
a=a.mrb_random_srand(c,a,b)
d=byteslice .NODE_ZSUPER(b)
root@83472e836918:/tmp/inputs#
```

We killed this job after letting it run for several minutes without finding a crash.

Fuzzing a pre-compiled target Next we used Gramatron to fuzz a pre-compiled binary target by running afl-fuzz in QEMU mode. First we recompiled the mruby target with plain GCC:

```
cd /tmp/mruby
make clean
make
```

Next we installed some required libraries and ran a "build_qemu_support.sh" script to add QEMU mode support to afl-fuzz:

```
cd ~/gramatron_src/afl-gf/qemu_mode
apt install -y flex libtool-bin libglib2.0-dev libpixmap-1-dev
./build_qemu_support.sh
```

In order for the "build_qemu_support.sh" script to work in the Docker container, it was necessary to add the "--no-check-certificate" flag in the following line of the script:

```
wget --no-check-certificate -c -O "$ARCHIVE" -- "$QEMU_URL" && OK=1
```

We then added the -Q flag to the afl-fuzz invocation in "run_campaign.sh":

```
$FUZZ_MAIN -m none -Q -a $AUTOMATON -i $INPUT_DIR -o $OUTPUT_DIR -- $RUNCMD
```

Now, running "run_campaign.sh" (with the same arguments as before) works:


```

Running the following recreates the source_automaton.json automaton file from the
    source.json grammar file
root@83472e836918:~/gramatron_src/gramfuzz-mutator/preprocess# ./prep_automaton.sh ~/
    grammars/ruby/source.json PROGRAM
File:source.json
Name:source
python3 construct_automata.py --gf source.json
0
[X] Actual Number of states: 19
[X] No unexpanded rules , absolute FSA formed
Copying source_automata.json to /root/grammars/ruby

```

When we did this with the provided JavaScript (sub)grammar file ~/grammars/js/source.json, the script hung with no explanation:

```

root@83472e836918:~/gramatron_src/gramfuzz-mutator/preprocess# ./prep_automaton.sh ~/
    grammars/js/source.json PROGRAM
File:source.json
Name:source
python3 construct_automata.py --gf source.json
It turned out that we had to specify a stack limit.
root@83472e836918:~/gramatron_src/gramfuzz-mutator/preprocess# time ./prep_automaton.
    sh ~/grammars/js/source.json PROGRAM 5
File:source.json
Name:source
python construct_automata.py --gf source.json --limit 5
[X] Operating in bounded stack mode
0
('[X] Actual Number of states:', 72)
('[X] Number of transitions:', 53620)
('[X] Original Number of states:', 82)
[X] Certain rules were not expanded due to stack size limit. Inexact approximation has
    been created and the disallowed rules have been put in source_disallowed.json
('[X] Number of unexpanded rules:', 2529)
Copying source_automata.json to /root/grammars/js

real    0m0.930s
user    0m0.906s
sys     0m0.023s
root@83472e836918:~/gramatron_src/gramfuzz-mutator/preprocess#

```

Lastly, we started with the recommended value of 5 and gradually increased it to see the effect on automaton size and generation time:

depth	time (seconds)	#states	#transitions	size (bytes)
5	.9	72	53620	1526012
6	5.6	205	368368	11048672
7	58.1	566	3870131	118622935

Figure 4.28: Automation size and generation time.

Use Cases and Limitations

As with all grammar-based fuzzers, Gramatron will only produce fuzzing inputs following the grammar. In turn, bugs that require inputs not produced by the grammar will not be triggered.

In the case of self-embedding grammars, the stack depth limit means that the grammar automaton generated by Gramatron will only generate a subset of the grammar. (A context-free grammar is self-embedding if it contains a derivation of the form $S \rightarrow^* u S v$ where S is a non-terminal symbol and u and v are non-empty strings of terminals.) Non-self-embedding grammars do not have this limitation. The user can generate more of the grammar by increasing the limit, at the cost of significantly increasing the size of the automaton as we have seen.

4.8 NyxNet

Reference Link	https://github.com/RUB-SysSec/nyx-net
Target Type	Binary
Host Operating System	Linux
Target Operating System	AIX, Android, DragonFly BSD, FreeBSD, Illumos, Linux, macOS/iOS (Darwin), NetBSD, OpenBSD, Plan 9, Solaris, and Windows
Host Architecture	x86_64
Target Architecture	x86_64, others as supported by the modified QEMU
Initial Release	Nov 14, 2021
License Type	Open Source (AGPL license)
Maintenance	The Nyx-Net repo is intended to capture the state described in the white paper. The original Nyx project receives regular updates.

Overview

Nyx-Net, a coverage guided fuzzer, is based on the fuzzer Nyx, which uses virtualization and snapshots to enhance the binary fuzzing experience. The -Net variation is focused on interrogating network services. Nyx-Net claims to improve “test throughput by up to 300x and coverage found by up to 70%.” The fuzzer has found previously unknown bugs in software such as Lighttpd, MySQL clients, and Firefox’s IPC mechanisms. According to the GitHub page, Nyx-Net is built upon some already existing technologies including kAFL, Red-queen, and it’s own predecessor, Nyx [36].

Design and Implementation

Nyx-Net has taken a different approach to advancing the state of the art. Other fuzzing solutions have focused on improvements to the fuzzing algorithms. The white paper suggests that even with algorithmic improvements the likelihood of finding bugs remains roughly the same on a first run. The Nyx-Net team is attempting to simply make target applications and systems more feasible to fuzz in the first place.

Nyx-Net uses a generative approach only allowing specification of input formats as sequences of typed function calls (or opcodes), forgoing the option of providing a seed. Nyx-Net's language allows for the definition of "opcodes" and "nodes" that are defined and chained together by the system. See figure 4.29 for an example. The example shows the construction of a data type named `d_bytes` and a definition for the type of data it should be, in this case unsigned 8 bit integers (unsigned char). The `n_con` variable is defined as a node that will create a new connection and return the handle `e_con`. Finally `n_pkt` is defined as an actual opcode that uses the node `n_con` to pass the data `d_bytes`.

```
d_bytes = s.data_vec("bytes", s.data_u8("u8"))
n_con = s.node_type("connection", outputs=[e_con])
n_pkt = s.node_type("pkt", borrows=[e_con], d_bytes)
```

Figure 4.29: Exploring provenance of a successful end state for a simple crackme

Nyx-Net utilizes snapshots of network services to jump-start the process of fuzzing a given location and circumvents the time sink of ensuring that the targeted application is in a known good state before fuzzing continues. Not doing this could cause misleading results as the test case that does crash the system may have been caused by a state from a previous test case leading to potential difficulties in crash triage. Nyx-Net claims to circumvent both of these problems with snapshots. "[...] by obtaining a snapshot of the system's state directly before executing the test case, we can reset the system to a deterministic state after each test" [37]. On its website, the originating project (Nyx) also expounds on how the use of snapshots and running everything with a KVM can help with fuzzing difficult applications such as shells [38]. It points out that shells have a tendency to be hard to fuzz because of other issues such as wiping file systems, running out of memory, fork-bombs (intentional or otherwise) and killing the parent process. All of these hinder the fuzzing of the target application with complications that can be more easily dealt with by running the target applications inside of a KVM.

Nyx-Net is controlled by a modified version of QEMU and executed by a modified version of KVM. Inside the QEMU Nyx runs what it calls an agent. The modified version of KVM uses modern CPU virtualization mechanisms to run the guest OS natively inside of the QEMU VM. This level of control over the VM allows for Nyx to quickly revert to previous snapshots

and interaction with the agent inside of the VM lets the underlying fuzzing system know exactly when the target is ready to be fuzzed again. This shaves time off of other fuzzers by eliminating the strict delays or the network connection start up time. This communication with the agents across VM boundaries is achieved by something called a hypercall. These hypercalls are similar to syscalls, but for VMs.

Nyx-Net claims some significant increases in efficiency in network based fuzzing applications which have been notoriously time consuming to fuzz in the past. Nyx-Net does this by emulating “significant fractions of the relevant functionality” [37]. Nyx-Net achieves this by hooking a multiple libc functions associated with network connectivity. Two methods to achieve this are provided: compiling the hooks directly into the target; or by using LD_PRELOAD. The file descriptors are tracked in an effort to maintain continuity of network connection paths upon snapshot resets. The system will also track file descriptors being passed to child processes from fork(). The read() / recv() functions are hooked and receive their data directly from the Nyx byte code generated test case.

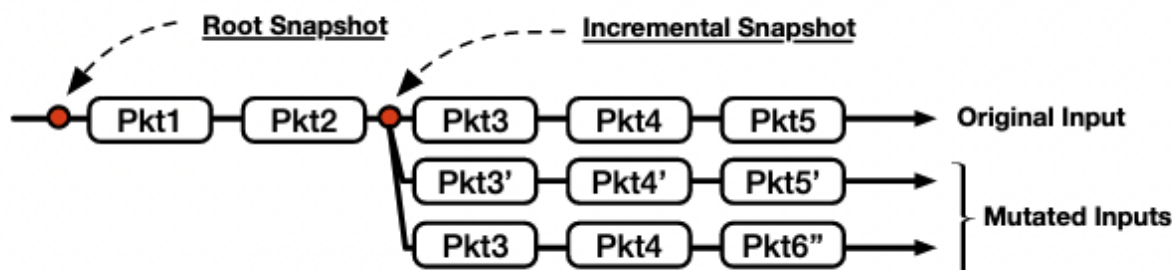


Figure 4.30: Using incremental snapshots to run mutated tests while skipping the common prefix consisting of packets one to three” [37]

Nyx-Net will also keep track of where the snapshots are in relation to the test case process. Figure 4.30 shows an example of how Nyx-Net keeps track of where to pick up the next test case and its relation to previous snapshots [37].

Use Cases and Limitations

Nyx-Net is best suited to fuzz networked applications that have time-consuming code paths. It ensures that individual input cases lead to specific crashes and that those crashes are not reliant on previous input.

Nyx-Net attempts to overcome the diminishing returns of algorithmic enhancements by taking direct control of the environment in which the target is run. Nyx-Net applies this approach to network services with the goal of increasing the reliability and reproducibility of the fuzzing and triage processes.

4.9 REVEN

Reference Link	https://www.tetrane.com/index.html#technology
Target Type	Windows x86 and x64 OS and binaries Linux x86 and x64 OS and binaries
Host Operating System	Debian 10 Linux
Target Operating System	Windows 7, 10, 11 Linux up to kernel version 4.19
Host Architecture	x86 (32 and 64)
Target Architecture	x86 (32 and 64)
Initial Release	Earliest mention is a 2014 blog. Free edition launched December 1, 2021.
License Type	Proprietary with free and paid versions
Maintenance	Maintained by Tetrane

Overview

REVEN is a timeless debugging and analysis platform[39]. It records full system execution traces that include CPU instructions, memory accesses, hardware events, fault handlers, I/O, and more. REVEN then creates a replayable session from these events that allows the user to transparently explore the trace across operating system boundaries such as user space, kernel space, and process isolation. The analyst interacts with REVEN through a web browser, but the tool includes a scriptable python API for automation.

While REVEN is a commercial tool, Tetrane recently released a free version. The free version was evaluated as part of the research for this article.

Design and Implementation

The REVEN toolkit captures and analyzes snapshots of virtual machines that are running the target OS or application. As such, the tool is divided into dedicated components: virtual machine manager and analysis engine. The virtual machine and project manager orchestrates the analyst's session: the virtual machine lifecycle, snapshots, and setup necessary for analysis. An analyst must first register a virtual machine with REVEN and generate a snapshot to act as the basis of collection. As of writing, the free edition supports only the Debian 9 virtual machine provided by Tetrane. The process of uploading the virtual machine and generating a snapshot is straightforward and there is ample documentation.

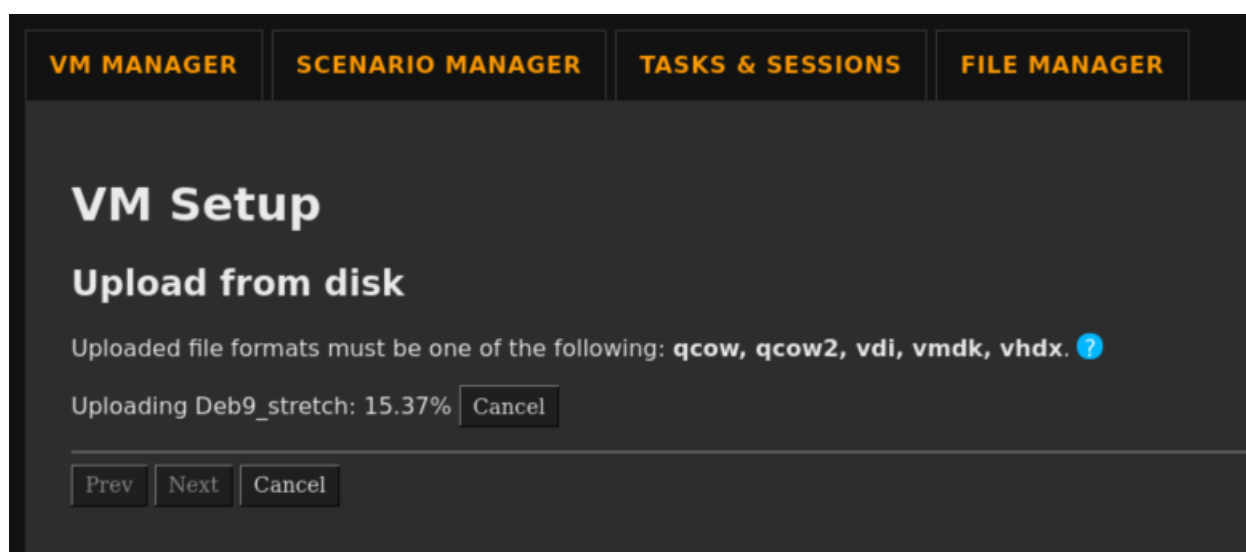


Figure 4.31: Uploading the Tetrane Debian 9 VM

VM Setup

Prepare the snapshot

The snapshot preparation is required to enable symbols in the REVEN trace. It will extract the filesystem of your VM.

Warning:

The extraction will require a free space about the same size as the final filesystem.

For a Windows 10 system, the filesystem is about 40GB large, meaning you will need about 40GB of free space to perform that operation.

For Linux system, this is usually far smaller, but depends a lot on the distribution.

⚠ This step is mandatory in the Free edition of REVEN.

Prepare

Task status: ☐ Cancel [\(more details\)](#)

Task log: ☐ Download log

```
[2021-12-17 16:21:22,489][INFO] Log started at 2021-12-17 16:21:22.489174-05:00
[2021-12-17 16:21:22,490][INFO] Log into /home/chess/Reven2/2.10.2/Logs/Tasks/4d20445c-9720-4b38-9f8c-41ca4f257530
[2021-12-17 16:21:22,667][INFO] Preparing snapshot 40de27b5-c8d1-411b-a9e8-9076fc3b222a
[2021-12-17 16:21:22,674][INFO] Extracting the filesystem...
[2021-12-17 16:21:22,685][INFO] Preparing the mount of the filesystem...
```

Figure 4.32: Generating an Initial Snapshot for Analysis

The analyst must then create a new scenario - a recording of the actions to be analyzed. This recording can then be replayed and analyzed as many times as necessary to explore the various CPU instructions, memory, traces, timelines, and more.

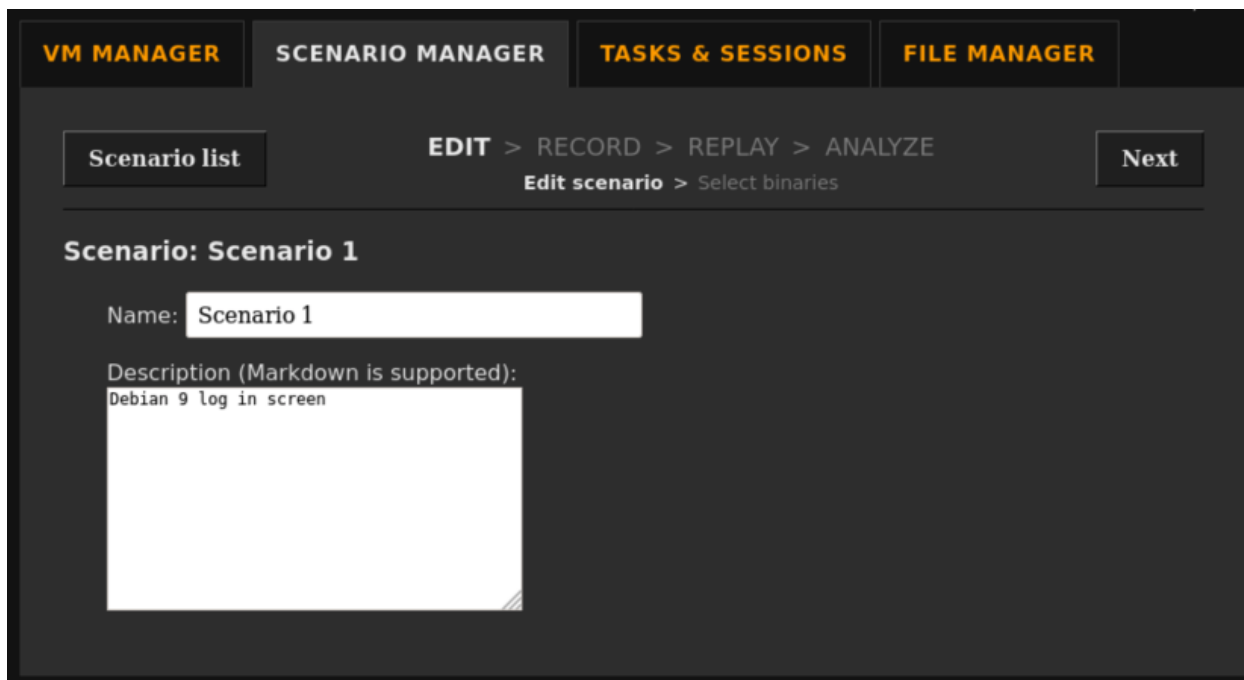


Figure 4.33: Generating an initial Scenario for record and replay

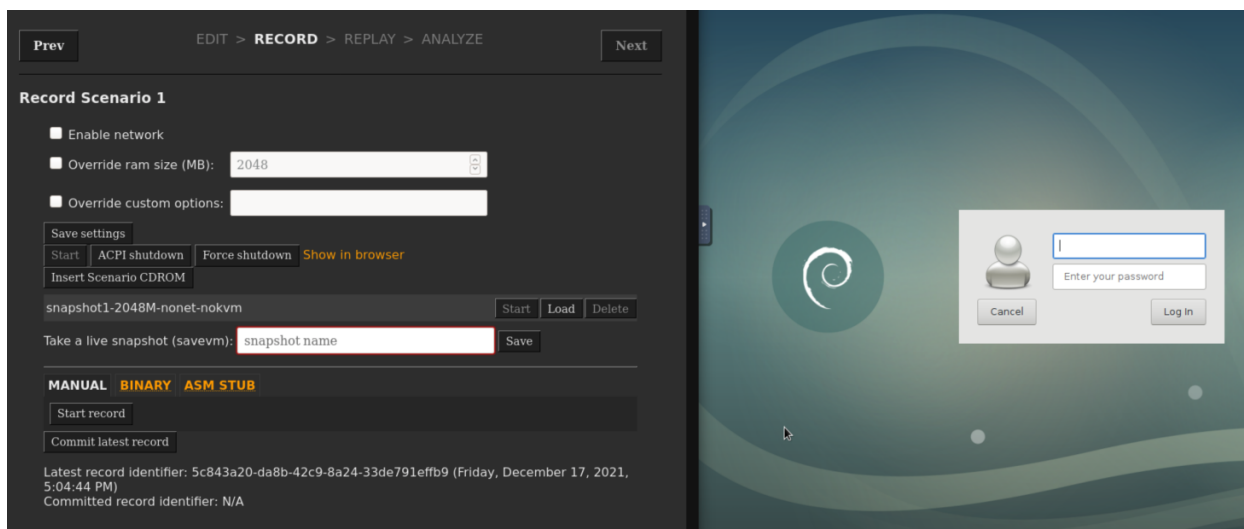


Figure 4.34: Recording actions within the VM

Once the scenario is generated it can be analyzed. This analysis window includes a variety of widgets with various purposes.

The REVEN GUI is the main method by which an analyst explores a scenario. There are a variety of widgets available to expedite analysis.

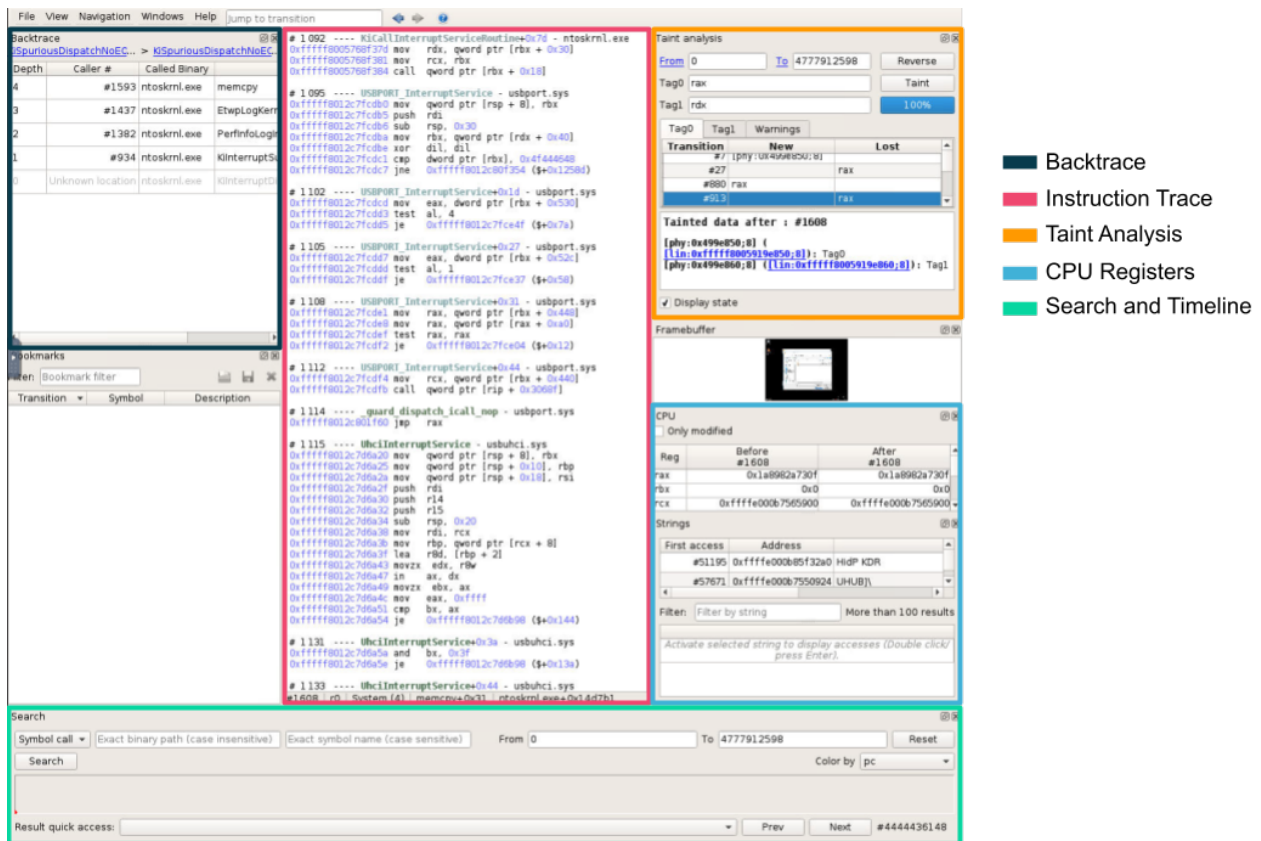


Figure 4.35: REVEN GUI for Exploring Scenarios

The Central Widget (red) contains the trace of CPU instructions executed by the VM while recording the scenario. It “includes kernel or user code while executing the binary and will indiscriminately jump from function to another, binary to another, and userland to kernel and back. Each instruction displayed is called a transition and represents a moment in time when the CPU did something.”

The CPU widget (blue) “contains content of registers before and after the selected transition is executed”. The Backtrace widget (black) “contains the current backtrace of calls leading to the active transition in the trace widget.” The backtrace will update based on the current active transition, as each transition may represent a different execution context. The Framebuffer Widget “displays frame buffer image at the active transition in the trace widget.”

The Timeline widget (green) “represents a linear time view of the execution trace. It is interactable and will update the other widgets. The trace is searchable and can find execution of symbols, binaries, or addresses. A specific binary name can be found with a specific symbol.”

An analyst can also explore the memory of the system via a hex dump via memory operands. On inspection of a memory operand the HexDump widget displays the “content of the requested memory range at the active transition” as well as an “access history view that

displays list of accesses to the area.” If the memory is not directly accessed it instead displays the next closest access. Exploring these memory accesses automatically adjusts the active transition.

The Taint widget (orange) “follows the data flow tracking backwards and forwards. The taint results correspond to a list of transitions when new data gets tainted or tainted data get untainted. Transitions where data is modified but not moved around are not presented in the taint results.”

Use Cases and Limitations

REVEN allows for full OS and application analysis. Its ability to provide a transparent view into user space applications through the kernel and back allows for deep introspection into system state. This is particularly useful for finding kernel exploits triggered through user space. Its replay system includes IO, greatly expediting analysis of complex applications such as web browsers for heap bugs. Additionally, REVEN integrates with IDA, WinDbg, Volatility, and Wireshark as well as providing a python API for automated analysis. There are also a variety of interactive tutorials that detail using REVEN to find modern CVEs.

REVEN runs on Debian 10 if installed directly, though it may run on other Linux operating systems via a Docker container. It can only target Windows and Linux systems on x86/x64 architectures, and the Linux kernel must be version 4.19 or earlier. The free edition can only record and replay a custom Debian 9 virtual machine. The professional version can target Windows 7, 10, 11, and a variety of Linux systems at or below 4.19. The license for the professional version at time of writing is 3,900 USD per year. Both the free and professional licenses only allow a single workflow - one recording and one replay. Finally, the management of virtual machines and their snapshots is storage intensive.

4.10 ZAFL

Reference Link	https://www.usenix.org/system/files/sec21-nagy.pdf https://git.zephyr-software.com/opensrc/zafl
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux (primary), Windows
Host Architecture	Linux
Target Architecture	x64 only
Initial Release	02/2021
License Type	Open Source (BSD 3-Clause)
Maintenance	Actively Maintained - Last updated 10/2021

Overview

ZAFL is a coverage-guided grey-box fuzzer [40]. ZAFL lifts a target binary into an intermediate representation and then applies static rewriting transformations to insert instrumentation.

The ZAFL authors note four main categories of transformations that coverage-guided fuzzers apply to improve performance. These are the same four areas to which ZAFL applies similar transformations:

- **Instrumentation pruning**, for instance AFL's instrumenting a percentage of basic blocks, or INSTRIM targeting only backwards/looping edges

- **Instrumentation downgrading**, such as CollAFL merging single-predecessor basic blocks
- **Sub-instruction profiling**, which multiple tools use to split up hard-to-bypass multi-byte checks like checksums into individual byte comparisons that enable the fuzzer to more easily detect progress
- **Extra-coverage behavior tracking**, prototyped in some LLVM implementations using additional context-sensitive information to distinguish traversal order among blocks.

ZAFL's goal, which they believe they have accomplished, is to apply these transformations consistently in real-world binaries, maintaining execution speed, and finding as many, or more crashes, as competing fuzzers.

Design and Implementation

The ZAFL authors note four primary pillars/requirements of their design and implementation:

- Criterion 1: Instrumentation added via static rewriting.. not emulation.. emulation is slow.
- Criterion 2: Instrumentation is invoked via inlining, not trampolining.. trampolining adds significant overhead with instruction control transfers out and back to the basic block.
- Criterion 3: Must facilitate register liveness tracking.. saving and restoring registers that weren't even touched is slow.
- Criterion 4: Support common binary formats and platforms including stripped binaries, C++, and Windows.

Name	Fuzzing Appearances	Fuzzing Overhead	Supports Xform	Instrumentation			Supported Programs			
				type	invoked	liveness	PIC & PDC	C & C++	stripped	PE32+
LLVM	[1, 6, 13, 18, 19, 31, 32, 47, 70, 75, 93]	18–32%	✓	static	inline	✓	N/A	✓	N/A	N/A
Intel PT	[7, 11, 20, 37, 75]	19–48%	✗	hardware	replay	✗	✓	✓	✓	✓
DynamoRIO	[37, 43, 73]	>1,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
PIN	[45, 49, 63, 68, 92]	>10,000%	✓	dynamic	inline	✓	✓	✓	✓	✓
QEMU	[23, 31, 91, 93]	>600%	✓	dynamic	inline	✓	✓	✓	✓	✓
Dyninst	[44, 55, 62, 76]	>500%	✓	static	tramp.	✗	✓	✓	✗	✗
RetroWrite	[26]	20–64%	✗	static	tramp.	✓	✗	✗	✗	✗

Table 2: A qualitative comparison of the leading coverage-tracing methodologies currently used in binary-only coverage-guided fuzzing, alongside compiler instrumentation (LLVM). No existing approaches are able to support compiler-quality transformation at compiler-level speed and generalizability.

Figure 4.36: Comparison of current tools across the previously defined criteria and transform approaches [40]

Note that ZAFL requires a binary lifter to an intermediate representation. They do not implement their own lifter from scratch to implement static rewriting. They evaluated the

popular LLVM lifter McSema but did not choose it due to performance issues. They instead leveraged a GCC Intermediate Representation (IR)-based lifter (Zipr).

ZAFL lifts the binary, then runs ZAX (the component of ZAFL where all the new instrumentation logic resides), then writes the intermediate representation of the binary back out for fuzzing. ZAX applies multiple transforms to the IR including:

- Performance Transformation
- Single Successor-based Pruning
- Dominator-based Pruning (if all paths to block B go through A then A dominates B)
- Instrumentation Downgrading
- Feedback Transformation
- Sub-instruction Profiling (which is also done in CmpCov)
- Context-sensitive Coverage (function-level annotations)

Also of note, when ZAX inserts instrumentation, it performs analysis to determine where that analysis is cheapest (such as avoiding areas that would require costly saving of EFLAGS).

ZAFL is focused on Linux targets, but also supports Windows.

Use Cases and Limitations

The primary use case for ZAFL is closed-source binaries where one would otherwise use DynInst or QEMU mode of AFL. “ZAFL enables fuzzers to average 26–131% more unique crashes, 48–203% more test cases, achieve 60–229% less overhead, and find crashes in instances where competing instrumenters find none.” [40]

ZAFL’s primary limitation is that it is not guaranteed to be able to apply its transforms on all targets, particularly heavily obfuscated binaries. As the ZAFL paper notes, ZAFL fails for obfuscated binaries such as Dropbox, Skype or Spotify.

Appendix

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

5.1 Resources

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

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

5.2 Tools Criteria

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

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

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

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

5.3 Techniques Criteria

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

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

5.4 Tool and Technique Categories

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

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

5.5 Static Analysis Technical Overview

5.5.1 Disassembly

An assembler converts a program from assembly language to machine code, and a disassembler performs the reverse: it converts machine code to assembly language. Since there is often a one-to-one correspondence between machine instructions and assembly instructions, this translation is much less complicated than decompilation. However, disassembly can pose challenges, especially with architectures like x86 which have variable length instructions. When overlapping sequences of bytes could themselves be valid instructions, one cannot just disassemble an instruction at random. Several approaches to disassembly address this challenge, including linear sweep (which disassembles instructions in the order they appear starting from the first instruction) and recursive descent (which disassembles instructions in the order of their control flow) [41]. Many popular disassemblers including IDA Pro [42] and Binary Ninja [43] use the latter technique.

Disassembling machine code is often the first step in binary analysis. There are currently a variety of disassemblers available, ranging from simple command line utilities to proprietary platforms with capabilities far beyond basic disassembly. A simple example is `objdump` [44], a standard tool on Linux operating systems, which given a target binary, will output its disassembly. Tools like debuggers often rely on more sophisticated disassembly frameworks like Capstone [45] which has features complementary to its core disassembler and is designed to be used via an API. The disassembly framework Miasm [46], which is a tool included in the second version of this report, can be used similarly to Capstone.

In contrast to frameworks, disassembly platforms are designed primarily for humans to analyze disassembled code through a graphical user interface (GUI). These are often sophisticated user applications which offer a significant range of features beyond disassembling code. For example, many of these applications have built-in APIs that can be used as frameworks for custom, automated analyses. Several of these tools were discussed in the first version of this report: IDA Pro [42], Ghidra [47] and Binary Ninja [43].

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

sembling [48], claims that at the time “no existing tool is able to disassemble executable binaries into assembly code that can be correctly assembled back in a fully automated manner, even for simple programs. Actually, in many cases, the resulting disassembled code is far from a state that an assembler accepts, which is hard to fix even by manual effort. This has become a severe obstacle [48, 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 [48, p. 1]. Since 2015, this technique has been improved, notably by the creators of angr who built a reassembling tool called Ramblr [49]. More recently, the tool DDisasm [50] was introduced. (DDisasm was discussed in the first version of this report.)

Static Binary Rewriting and Static Instrumentation Binary rewriting modifies a binary executable without needing to change the source code and recompile. One use case is for binary instrumentation, which is often thought of as a dynamic technique. While many dynamic binary instrumentation (DBI) techniques exist, there are also methods for statically instrumenting binaries. Many of these rely on reassembling or relinking the binary. Retrowrite [41], a tool designed to statically instrument binaries for dynamic analysis like fuzzing and memory checking, also uses a reassembleable disassembly technique that builds on previous research. The tool LIEF [51], discussed in the first version of this report, allows the user to statically hook into a binary, or statically modify it in a variety of ways.

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

Intermediate Representations are commonly used in compilers; a familiar example being LLVM [52], the IR used in the Clang compiler [53]. LLVM is helpful as an example not just because it is well known, but because it demonstrates the range of features a well-designed IR can offer. The instruction set and type system for LLVM is language independent, which means there are no high-level types and attributes. This allows LLVM to be ported to many architectures. Although the type system is low-level, by providing type information, LLVM enables a target program to be optimized through various analyses [54]. Unlike machine code however, LLVM is designed to be human readable [54].

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

These traits are not specific to LLVM but are attributes of many IRs discussed in this report. Clang's compiler works by translating source code languages to LLVM, performing optimizations in this form, and then translating the LLVM bitcode to many possible architectures[55]. The Ghidra decompiler does something similar but in reverse: a binary program is first lifted (converted to a higher-level representation) to an IR called P-Code [56], 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 [57].

Ghidra uses SSA form in its decompilation. However, unlike LLVM, P-Code is not in SSA form by default [57]. Other IRs also have an SSA and non-SSA form, for instance, Binary Ninja's IRs offer the ability to toggle between non-SSA and SSA form [43].

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

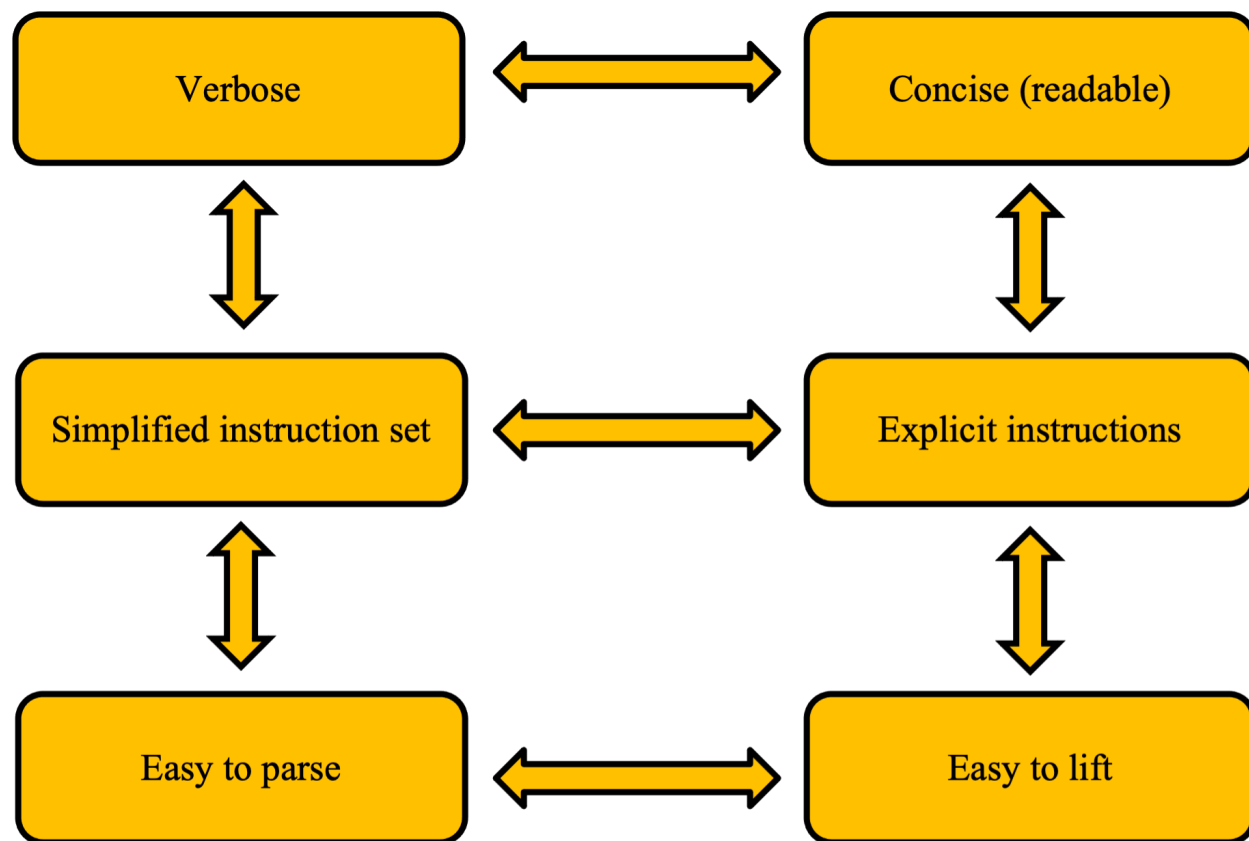


Figure 5.1: Tradeoffs of IRs, Pt. 1 [58, p. 29] The double arrows imply that emphasizing one makes the other more difficult.

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

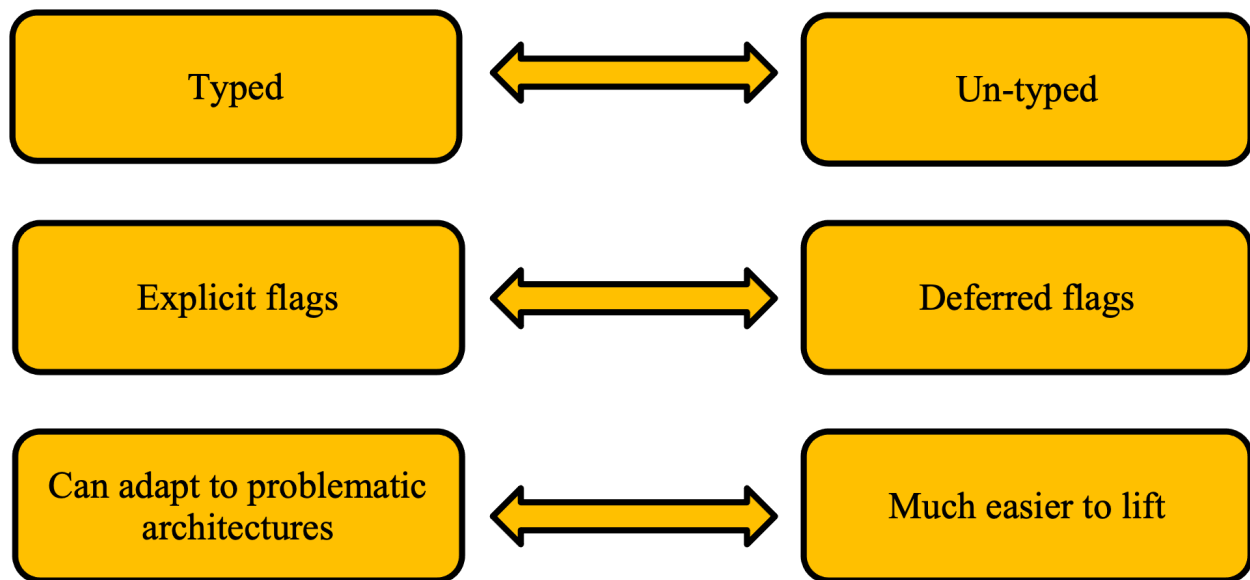


Figure 5.2: Tradeoffs of IRs, Pt. 2 [58, p. 30] The double arrows imply that emphasizing one makes the other more difficult.

utilize it for their own analyses. IDA Pro only recently documented their API [59], 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 [60]. Their third level serves as a decompilation level.

Other IR frameworks discussed in the second version of this report can be used in the same manner, but each offer their own set of features. Binary Analysis Platform (BAP) [61] is a framework designed for program analysis which is built around the BAP Intermediate Language (BIL), which has a formally defined grammar [61]. Miasm has an expression-based IR that facilitates tracking memory and registry values. Miasm also has a JIT engine for emulation and has built in support for symbolic execution [46].

Certain IRs are tailored for specific use cases. For example, Fuzzilli, a fuzzer which targets Javascript JIT engines, uses a custom IR called FuzzIL. Seeds are constructed and mutated in FuzzIL then translated into Javascript before being fed into the engine [62]. This approach has the benefit of being able to theoretically explore all possible patterns given enough computing power, unlike a JIT fuzzer working from hardcoded Javascript samples.

In contrast some tools in this report use existing IRs rather than creating their own. The symbolic execution tool angr uses Vex, which is the IR implemented by the memory debugger Valgrind [63]. WinAFL, a version of the AFL fuzzer for the Windows operating system, uses DynamoRIO, a dynamic binary instrumentation engine with its own IR [64].

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 [65], which is discussed in the previous version of this report, is an IR designed to convert between different IRs. It is also the IR used in DDisasm.

5.5.2 Decompilation

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

Whereas compiler theory has been a popular area of computer science for decades, its reverse has received far less attention. In 1994 Christina Ciafuentes published her PhD thesis on the subject, Reverse Compilation Techniques [66]. 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 [67]. 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 [68]. In March 2020, Vector35 released a decompiler for their tool, Binary Ninja. Binary Ninja not only exposes its IRs to the user, but makes them a fundamental part of its design, with this new decompilation acting as a third layer in their three-tiered IR system. Their decompilation is available in both SSA and non SSA form.

5.5.3 Static Vulnerability Discovery

There are a number of tools and techniques intended to statically discover vulnerabilities. Many are designed for source code, including tools such as Coverity [69], CodeSonar [70], and CodeQL [71]. These use a number of static analysis algorithms to find possible vulnerabilities and common vulnerability patterns in a code base. Additionally, there are program analysis techniques designed to statically identify vulnerabilities in binary code, such as graph-based vulnerability discovery and value-set analysis (VSA) [63, p. 5].

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

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

branches: a basic block is a sequence of instructions that contains no jumps, except at the entry and exit. A control flow graph (CFG) models a program as a graph in which the basic blocks of the program are represented as nodes, and the jumps, or branches, are represented as edges. A CFG is instrumental to many forms of static program analysis and vulnerability discovery. Recovering one is done by disassembling the program and identifying the basic blocks and the jumps between them (both direct and indirect) [63, 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 [41]. One attribute of many IRs is that their lifters will recover variable and type information and include it in the IR form. This is also necessary for decompilation.

- **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 [41].
- **Value Set Analysis (VSA):** VSA is a form of static analysis which attempts to track values and references throughout a binary [63]. This analysis has a variety of uses, including identifying indirect jumps or find vulnerabilities such as out of bound accesses.
- **Graph-based vulnerability discovery:** This form applies graph analysis to a CFG to identify vulnerabilities [63].
- **Symbolic Execution:** Symbolic execution replaces program inputs with symbolic values, and then symbolically executes over the program. Symbolic execution be done either entirely statically or in conjunction with dynamic analysis. See page in the dynamic analysis section for a more in-depth discussion.
- **Abstract Interpretation, data-flow analysis, etc.:** There are many types of formal static analysis which apply mathematical approaches to program analysis. These include abstract interpretation and data-flow analysis. The tools BAP has implemented support these forms of analysis [72].

5.6 Dynamic Analysis Technical Overview

5.6.1 Debuggers

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

and dynamic techniques (e.g., the popular debugger GDB uses a disassembler and the tracing utility ptrace [73]) to implement its functionality.

Recordable, replayable debugging is one of the most powerful additions to modern debugging. This allows a user to record a program execution and then replay while debugging the process. In addition to forward debugger actions like step and continue, replayable debugging allows the user to step backwards and continue backwards, etc. TTD, a tool discussed in a previous edition, allows for replayable debugging from within the Windows debugger Windbg [74]. The tool rr [75], which was discussed in the first version of this report, enables recordable replayable debugging on Linux.

5.6.2 Dynamic Binary Instrumentation (DBI)

DBI, which underlies many dynamic binary analysis techniques, entails modifying the binary, either before or during execution, often by hooking into the binary at specific points and injecting code. DBI frameworks implement custom instrumentation which the user can access through an API to perform dynamic analysis. These include Intel Pin [76] and DynamoRIO [77], which underlie many of the tools discussed in these reports. Both can be used to drive the Windows fuzzer WinAFL [64], and the dynamic binary analysis tool Triton is built around Intel Pin [78]. DBI frameworks are implemented in a variety of ways. Intel Pin works by intercepting instructions before they are executed and recompiling them into a similar, but Intel Pin-controlled instruction, which is then executed [76]. 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 [77]. Other DBI options are less granular and intrusive and rely on hooking into the program through dynamically loaded libraries (e.g., this is how the tool Frida [79] operates).

5.6.3 Dynamic Fuzzing Instrumentation

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

5.6.4 Memory Checking

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

information. Several tools exist to do this, such as Valgrind [80], Dr. Memory (a part of the DynamoRIO framework) and LLVM's Sanitizer Suite which includes Address Sanitizer (ASAN) [52].

5.6.5 Dynamic Taint Analysis

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

5.6.6 Symbolic and Concolic Execution

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

Symbolic execution can be performed “dynamically,” and this is called dynamic symbolic execution, or DSE. However, throughout literature on symbolic execution there are generally two competing definitions of DSE. The first kind of DSE refers to any form of symbolic execution which “explores programs and generates formulas on a per-path basis [81]”. 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 [81, p. 3]. In the paper (State of) The Art of War: Offensive Techniques in Binary Analysis [63], 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. [63, 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 [63].

The other common definition of DSE and concolic execution refers solely to symbolic execution which is "driven by a specific concrete execution [82, 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 [83], [82, p. 5-6]. After doing this, additional paths can be explored by negating one (or more) of the collected branch conditions for the path of the concrete input, and then solve for the new path with these negated conditions using an SMT solver in order to generate a new input [82, 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 [84] (a fuzzer discussed in the first version of this report) is an example of hybrid fuzzing.

There are many tools for symbolic execution, including Triton and Miasm. The tool angr [63] (discussed in the first version of this report) is one of the most popular, publicly available tools, and uses emulation to perform symbolic execution.

While symbolic execution does provide powerful insights into program semantics, it is greatly limited by space and time complexity issues. Path explosion is one of the challenges in symbolic execution: unbounded loops might result in exponentially many new paths. Symbolic execution is also hindered by the memory needed to store a growing number of path constraints. It is also difficult to apply to real world systems, because system calls and library calls can be hard to manage with symbolic values, among other environmental concerns [82]. Additionally, constraint solving is often a difficult and time-consuming task. As such, symbolic execution is in many cases not a feasible option or must be constrained to a small area of the program.

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

Bibliography

- [1] D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. M. Mazurek, "An observational investigation of reverse engineers' processes," in *USENIX Security Symposium*, 2020. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational>
- [2] A. R. Bryant, "Understanding how reverse engineers make sense of programs from assembly language representations," Ph.D. dissertation, Air Force Institute of Technology, 2012.
- [3] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *IEEE Symposium on Security and Privacy (SP)*, 2018.
- [4] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *ACM Automated Software Engineering (ASE)*, 2016.
- [5] L. Battle and J. Heer, "Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau," *Computer Graphics Forum (Proc. EuroVis)*, 2019. [Online]. Available: <http://idl.cs.washington.edu/papers/exploratory-visual-analysis>
- [6] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *IEEE Symposium on Visual Languages*, 1996.
- [7] J. P. Leasure. (2021) Ghidraal. [Online]. Available: <https://github.com/jpleasu/ghidraal>
- [8] Oracle. (2021) Get started with graalvm. [Online]. Available: <https://www.graalvm.org/22.0/docs/getting-started/>
- [9] CERTCC. (2021) Fuzzolic. [Online]. Available: <https://github.com/CERTCC/kaiju>
- [10] T. of Bits. (2021) Mui. [Online]. Available: <https://github.com/trailofbits/MUI>
- [11] ——. (2021) Manticore. [Online]. Available: <https://github.com/trailofbits/manticore>
- [12] V. 35. (2018) angr_plugin.py. [Online]. Available: https://github.com/Vector35/binaryninja-api/blob/dev/python/examples/angr_plugin.py
- [13] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Efficient Binary-Level Coverage Analysis," in *ACM Joint European Software Engineering Conference and Symposium on the*

Foundations of Software Engineering - ESEC/FSE'20. Virtual Event, USA: ACM Press, nov 2020, pp. 1153–1164.

- [14] B. Khadra. (2021) bcov. [Online]. Available: <https://github.com/abenkhadra/bcov>
- [15] S. Software. (2021) 010 editor. [Online]. Available: <https://www.sweetscape.com/010editor/>
- [16] R. Dutra, R. Gopinath, and A. Zeller, “Formatfuzzer: Effective fuzzing of binary file formats,” *CoRR*, vol. abs/2109.11277, 2021. [Online]. Available: <https://arxiv.org/abs/2109.11277>
- [17] usd se. (2021) Formatfuzzer. [Online]. Available: <https://github.com/uds-se/FormatFuzzer>
- [18] J. Jang and H. K. Kim, “Fuzzbuilder: Automated building greybox fuzzing environment for c/c++ library,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 627–637. [Online]. Available: <https://doi.org/10.1145/3359789.3359846>
- [19] hksecurity. (2020) Fuzzbuilder. [Online]. Available: <https://github.com/hksecurity/FuzzBuilder>
- [20] glennrp. (2021) libpng. [Online]. Available: <https://github.com/glennrp/libpng>
- [21] Season-Lab. (2021) Fuzzolic. [Online]. Available: <https://github.com/season-lab/fuzzolic>
- [22] L. Borzacchiello, E. Coppa, and C. Demetrescu, “FUZZOLIC: mixing fuzzing and concolic execution,” *Computers & Security*, 2021.
- [23] Google. (2020) Using go at google. [Online]. Available: <https://go.dev/solutions/google/>
- [24] dvyukov. (2021) go-fuzz: randomized tstring for go. [Online]. Available: <https://github.com/dvyukov/go-fuzz>
- [25] Google. (2021) Fuzz testing for go. [Online]. Available: <https://github.com/google/gofuzz>
- [26] K. Hockman. (2021) Design draft: First class fuzzing. [Online]. Available: https://go.googlesource.com/proposal/+/_master/design/draft-fuzzing.md
- [27] golang.org. (2021) Testing package documentation. [Online]. Available: <https://pkg.go.dev/testing@master#hdr-Fuzzing>
- [28] capnspacelab. (2021) Pull request #50108. [Online]. Available: <https://github.com/golang/go/pull/50108/commits/6f94f3fc2ef31ab5ccf231038879c4af976cd1f0>
- [29] K. Hockman. (2021) Proposal #44551. [Online]. Available: <https://github.com/golang/go/issues/44551>

- [30] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 244–256. [Online]. Available: <https://doi.org/10.1145/3460319.3464814>
- [31] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.
- [32] MITRE. (2020) Cve-2020-15866. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15866>
- [33] AFLplusplus. (2021) Gramatron. [Online]. Available: <https://github.com/hexhive/gramatron>
- [34] HexHive. (2021) Afl, change log for version 3.15a-dev. [Online]. Available: <https://aflplusplus/docs/changelog/#version-315a-dev>
- [35] (2021) Greibach normal form. [Online]. Available: https://en.wikipedia.org/wiki/Greibach_normal_form
- [36] RUB-SysSec. (2021) Nyx-net: Network fuzzing with incremental snapshots. [Online]. Available: <https://github.com/RUB-SysSec/nyx-net>
- [37] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," *CoRR*, vol. abs/2111.03013, 2021. [Online]. Available: <https://arxiv.org/abs/2111.03013>
- [38] S. Schumilo and C. Aschermann. (2021) To boldly fuzz what no other tool can fuzz. [Online]. Available: <https://nyx-fuzz.com/>
- [39] TETRANE. (2021) Reven free edition 2.10.2 user documentation. [Online]. Available: <https://doc.tetrane.com/free/latest/Index.html>
- [40] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/nagy>
- [41] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1497–1511. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00009>
- [42] Hex-Rays. (2021) Ida pro. [Online]. Available: <https://hex-rays.com/ida-pro/>
- [43] V. 35. (2019) Binary ninja. [Online]. Available: <https://binary.ninja/>
- [44] I. Free Software Foundation. (2019) Gnu binutils. [Online]. Available: <https://www.gnu.org/software/binutils/>

- [45] N. A. Quynh. (2013) Capstone engine. [Online]. Available: <https://www.capstone-engine.org/>
- [46] C. I. Security. (2019) Miasm. [Online]. Available: <https://github.com/cea-sec/miasm>
- [47] N. S. Agency. (2021) Ghidra. [Online]. Available: <https://ghidra-sre.org/>
- [48] S. Wang, P. Wang, and D. Wu, “Reassembleable disassembling,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15. USA: USENIX Association, 2015, p. 627–642.
- [49] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, “Ramblr: Making reassembly great again,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/ramblr-making-reassembly-great-again/>
- [50] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1075–1092. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya>
- [51] Quarkslab. LIEF. [Online]. Available: <https://lief.quarkslab.com/>
- [52] LLVM Foundation. The llvm compiler infrastructure. [Online]. Available: <https://llvm.org/>
- [53] ——. Clang: a C language family frontend for llvm. [Online]. Available: <https://clang.llvm.org/>
- [54] ——. Llvm language reference manual. [Online]. Available: <https://llvm.org/docs/LangRef.html>
- [55] ——. “clang” CFE internals manual. [Online]. Available: <https://clang.llvm.org/docs/InternalsManual.html>
- [56] National Security Agency. Ghidra software reverse engineering framework. [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra>
- [57] A. Bulazel. (2019) Working with ghidra’s p-code to identify vulnerable function calls. [Online]. Available: <https://www.riverloopsecurity.com/blog/2019/05/pcode/>
- [58] P. LaFosse and J. Weins, “Modern binary analysis with il’s,” presented at the BlueHat Seattle, 2019.
- [59] R. Rolles. (2018) Hex-rays microcode api vs. obfuscating compiler. [Online]. Available: <https://hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/>
- [60] V. 35. Binary ninja intermediate language series, part 1: Low level il. [Online]. Available: <https://docs.binary.ninja/dev/bnil-lil.html>
- [61] CMU Cylab. (2015) Carnegie mellon university binary analysis platform (cmu bap). [Online]. Available: <https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf>

- [62] S. Groß, “Fuzzil: Coverage guided fuzzing for javascript engines,” Master’s thesis, KIT, Karlsruhe, 2018.
- [63] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.
- [64] Google Project Zero. (2019) WinAFL. [Online]. Available: <https://github.com/googleprojectzero/win afl>
- [65] GrammaTech. (2020) GTIRB. [Online]. Available: <https://github.com/GrammaTech/gtirb>
- [66] C. Cifuentes, “Reverse compilation techniques,” Ph.D. dissertation, QUT, Brisbane, 1994.
- [67] I. Guilfanov, “Keynote: The story of IDA Pro,” presented at, CODE BLUE, Tokyo, Dec 2014.
- [68] L. H. Newman. (2019) The NSA makes ghidra, a powerful cybersecurity tool, open source. [Online]. Available: <https://www.wired.com/story/nsa-ghidra-open-source-tool/>
- [69] Synopsys. Coverity scan static analysis. [Online]. Available: <https://scan.coverity.com/>
- [70] GrammaTech. Codesonar. [Online]. Available: <https://www.grammatech.com/codesonar-cc>
- [71] Semmle. CodeQL. [Online]. Available: <https://semmlle.com/codeql>
- [72] I. Gotovchits. (2019) [ANN] BAP 2.0 Release. [Online]. Available: <https://discuss.ocaml.org/t/ann-bap-2-0-release/4719>
- [73] Free Software Foundation, Inc. GDB: The GNU project debugger. [Online]. Available: <https://www.gnu.org/software/gdb/>
- [74] Microsoft. (2020) Debugging using windbg preview. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview>
- [75] Mozilla. rr. [Online]. Available: <https://rr-project.org/>
- [76] Intel. (2021) Pin - a dynamic binary instrumentation tool. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>
- [77] D. Bruening. DynamoRIO. [Online]. Available: <https://dynamorio.org/>
- [78] Quarkslab. Triton - a DBA framework. [Online]. Available: <https://triton.quarkslab.com/>
- [79] O. A. V. Ravnås. Frida. [Online]. Available: <https://frida.re/>
- [80] T. V. Developers. Valgrind. [Online]. Available: <https://valgrind.org/>

- [81] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser, "Veritesting challenges in symbolic execution of java," *SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, p. 1–5, Jan. 2018. [Online]. Available: <https://doi.org/10.1145/3149485.3149491>
- [82] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018. [Online]. Available: <https://doi.org/10.1145/3182657>
- [83] CEA IT Security. (2017) Playing with dynamic symbolic execution. [Online]. Available: https://miasm.re/blog/2017/10/05/playing_with_dynamic_symbolic_execution.html
- [84] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/yun>